HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

JassBot mit Machine Learning

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2017

Autoren: Andreas Eder, Pascal Kistler

Betreuer: Prof. Dr. Markus Stolze (IFS)

Experte: Thomas Kälin (bbv Software Services AG)

Gegenleser: Prof. Dr. Daniel Patrick Politze (IPEK)

Abstract

Ausgangslage

Im Frühling 2017 veranstaltete die Firma Zühlke Engineering AG einen Wettbewerb in der Jassvariante Schieber. Das Besondere daran war, dass die Teilnehmenden eigene Programme (Bots) entwickeln mussten, die für sie jassen konnten. Der Veranstalter Zühlke Engineering AG stellte einen Server zur Verfügung, der das Kommunikationsprotokoll und die Spielregeln vorgab.
Am Wettkampftag wurde der beste Bot im Jassturnier auserkoren. Als Turnierabschluss durfte der Bot des Siegerteams gegen ein menschliches Turnierteam antreten. Dieses letzte Spiel zeigte, dass auch der beste Bot des Turniers chancenlos gegen das menschliche Team ist.
Der hier vorliegenden Forschungsarbeit liegt jener Jasswettbewerb zu Grunde. Ziel dieser Forschungsarbeit war es einen Bot zu entwickeln, welcher nicht nach vorgegebenen heuristischen Instruktionen spielt, sondern den Schieber Jass selber erlernen kann. Bekannt ist diese Idee von Projekten, wie AlphaGo, AlphaGo Zero und AlphaZero, welche von Google für das Spiel Go umgesetzt wurden.

Vorgehen/Technologien

Zur Lösung dieser Fragestellung wurde entschieden, wie bei AlphaGo, mit neuronalen Netzwerken und zwei verschiedenen Lernstrategien zu arbeiten. Für jede Strategie wurde ein Prototyp mit den Machine Learning Bibliotheken TensorFlow und Keras in Python entwickelt. Zur Erforschung einer idealen Konfiguration wurden verschiedene Aufbauarten von neuronalen Netzwerken mit der Strategie Supervised Learning geprüft. Die daraus resultierten Ergebnisse wurden anschliessend in ein Netzwerk mit Reinforcement Learning übernommen, um dieses mit verbesserter Konfiguration zu trainieren.
Die Daten von gespielten Jasspartien für das Supervised Learning wurden von Swisslos zur Verfügung gestellt. Als Lernpartner für das Reinforcement Learning konnte der siegende Bot aus dem Zühlke Jasswettbewerb verwendet werden.

Ergebnis

Daraus entstand für beide Lernstrategien je ein Prototyp, welcher auf dem Niveau eines Gelegenheitsjassers spielen kann. Im Turnier gegen den siegenden Bot aus dem Zühlke Jasswettbewerb verlor der beste Prototyp jedoch noch die meisten Spiele. Diese Prototypen könnten mit weiteren Optimierungen der Parameter sowie einer längeren Trainingszeit mit mehr Trainingsdaten noch verbessert werden. Das nächste interessante Ziel wäre die Entwicklung eines Prototyps, in welchem die beiden Lernstrategien kombiniert werden.

Aufgabenstellung Bachelorarbeit Abteilung I, HS 2017 Andreas Eder, Pascal Kistler

JassBot mit Machine Learning

- Wie ein Computer selbständig jassen lernt

1. Betreuer

Betreuer dieser Arbeit ist
Prof. Dr. Markus Stolze mstolze@hsr.ch

Co-Referent dieser Arbeit ist
Prof. Dr. Daniel Patrick Politze

Externer Experte dieser Arbeit ist
Thomas Kälin

2. Ausgangslage

Im ersten Halbjahr 2017 wurde von der Firma Zühlke eine Jass-Challenge als Wettbewerb durchgeführt. Dabei hatten die Teilnehmer die Aufgabe ein Programm (JassBot) zu entwickeln, dass gegen andere JassBots im Schieber Jass gewinnt. Der Wettbewerb war dabei so aufgebaut, dass jeweils zwei Instanzen des JassBots des einen Teams gegen zwei Instanzen des JassBots eines anderen Teams spielten. Eine direkte Kommunikation der Instanzen war dabei verboten.

Im Wettbewerb der Firma Zühlke trat der JassBot des Gewinner-Team anschliessend gegen die ehemalige Jass-Sendung-Moderatorin Monika Fasnacht an. Dieses Spiel verlor der JassBot.

Alle Programme die an der Zühlke Jass-Challenge teilnahmen nutzten heuristische Verfahren. Das aktuell erfolgreichste Programm für Go (AlphaGo) nutzt dagegen Machine-Learning Techniken.

3. Ziele der Arbeit

Ziel dieser Arbeit ist zu analysieren inwieweit sich Machine-Learning Techniken wie sie bei AlphaGo genutzt wurden sich auf ein JassBot Programm übertragen lässt. Es soll analysiert werden ob es möglich ist auf Basis dieser Techniken einen JassBot zu entwickeln der dem aktuellen Gewinner-Programm der JassBot Challenge überlegen ist und womöglich auch gegen menschliche Spieler sehr gut abschneidet.

Im Team des gewinnenden JassBot war ein HSR Student involviert (Daniel Latzer). Freundlicherweise darf dieses Programm als Benchmark für diese Arbeit genutzt werden.

Das Projektteam führt eine Forschungsarbeit im Auftrag von Prof. Dr. Markus Stolze durch. Diese beinhaltet die Aufgabe einen JassBot mittels Machine Learning zu entwickeln. Dies bedeutet, dass sich das Projektteam mit der Thematik des Machine Learning auseinandersetzt, sich darin einarbeitet und das neu erworbene Wissen einsetzen und anwenden kann. Der JassBot soll so konstruiert werden, dass das Programm das Spiel sowie wichtige Entscheidungsstrategien selber lernt. Es darf aber zusätzlich mit Heuristiken gearbeitet werden.

4. Vereinbarte Rahmenbedingungen

Der JassBot soll gegen das Interface des Zühlke Jass-Challenge-Servers entwickelt werden. Entsprechend soll der JassBot mir einer zweiten Instanz von sich selbst gegen den Referenz JassBot antreten können. Bei dieser Arbeit steht die Auseinandersetzung und die Entwicklung mit Hilfe von Machine Learning im Vordergrund, daher wird kein rein heuristischer Lösungsansatz verfolgt.

Die Studierenden können sich Hilfe von Samuel Kurat (Mitarbeiter IFS) und Hannes Badertscher (Mitarbeiter ICOM) holen. Es besteht auch die Zusage, dass der NVIDIA Cluster der Abteilung E für diese Arbeit genutzt werden darf.

5. Erwartetes Ergebnis

Der Inhalt dieser Machbarkeitsstudie soll einen Einblick in die Thematik Machine Learning geben und als Show-Case für diese Techniken genutzt werden können.

4. Dokumentation

Die Experimente und Resultate sind sinnvoll zu dokumentieren. Für den abschliessend entwickelten JassBot ist eine sinnvolle Dokumentation der genutzten Machine Learning Techniken abzuliefern. Die zentralen Elemente des Codes (Algorithmen) sind so zu dokumentieren, dass die genutzten Machine Learning Techniken nachvollziehbar sind.

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die Dokumentation (inkl. Source-Code) ist vollständig entsprechend den Instruktionen des Studiengangs abzugeben. Zudem ist ein Download-Link für Prof. Stolze und weitere Exemplare nach Absprache mit dem Co-Referenten und dem Experten bereitzustellen.

Zudem ist eine kurze Projektresultatdokumentation im öffentlichen Wiki von Prof. M. Stolze zu erstellen.

5. Weitere Regeln und Termine

Im Weiteren gelten die allgemeinen Regeln zu Bachelor und Studienarbeiten
„Abläufe und Regelungen Studien- und Bachelorarbeiten im Studiengang Informatik" (HSR Intranet)
https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html)

Der Terminplan ist hier ersichtlich (HSR Intranet)
https://www.hsr.ch/Termine-Bachelor-und-Studiena.5142.0.html

6. Rechte

Die resultierende Software und Dokumentation soll möglichst als Open-Source Software publiziert werden, so dass sie ohne Einschränkungen sowohl von den Studenten, wie auch von der HSR weiter genutzt und erweitert werden kann. Es ist daher bei der Verwendung von Libraries darauf zu achten, dass möglichst keine Libraries mit viraler Open-Source Lizenz (z.B. GNU) genutzt werden.

7. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert. Entsprechend sollten ca. 350h Arbeit für die Bachelorarbeit aufgewendet werden. Dies entspricht ungefähr 25h pro Woche (auf 14 Wochen) und damit ca. 3 Tage Arbeit pro Woche pro Student.

Für die Beurteilung ist der HSR-Betreuer verantwortlich. Die Bewertung der Arbeit erfolgt entsprechend der verteilten Kriterienliste.

Diese definitive Aufgabenstellung wurde am Mittwoch 17.10.17 beschlossen.

Rapperswil, Mittwoch 17.10.17
Prof. Dr. Markus Stolze, Institut für Software, Hochschule für Technik Rapperswil

Inhaltsverzeichnis

1 Management Summary

Ausgangslage

Die Firma Zühlke Engineering AG führte im Frühling 2017 einen Schieber Jasswettbewerb der besonderen Art durch. Dabei traten Computerprogramme, sogenannte Bots, im Turnier gegeneinander an. Diese wurden von den Teilnehmenden des Wettbewerbes im Vorfeld entwickelt.
Der Siegerbot wurde anschliessend von einem menschlichen Team herausgefordert. In diesem Duell unterlag der Bot jedoch den Menschen.
Durch diesen Wettkampf inspiriert, entstand die Idee zur folgenden Forschungsarbeit. Es soll Tradition in Form des Schieber Jasses mit modernen Technologien der Informatik verbunden werden. Dabei werden Konzepte aus dem Gebiet des Machine Learnings verwendet. Durch neue Errungenschaften der letzten zwei Jahrzehnten in der Hardware- und Softwareentwicklung konnten vermehrt bis anhin unlösbare Ideen realisiert werden.
Erfolgreiche Projekte aus diesem Bereich sind AlphaGo und AlphaZero von Google, welche für das Spiel Go und für zwei Varianten des Schachs entwickelt wurden. In dieser Forschungsarbeit soll daher mit ähnlichen Überlegungen ermittelt werden, ob ein JassBot entwickelt werden kann, der das Jassen selbst erlernen kann.

Vorgehen

Nach der Analyse von ähnlichen Fragestellungen wurde der bevorzugte Lösungsansatz mit einem neuronalen Netzwerk verfolgt. Dafür wurden zwei verschiedene Lernstrategien angewandt. Die erste, das Supervised Learning, eignet sich besser für die Erforschung der optimalen Parameter des Netzwerkes. Mit der zweiten Strategie, dem Reinforcement Learning, kann mit den ermittelten Parametern ein selbst lernendes Netzwerk mit besserem Lernerfolg entwickelt werden.
Da die Trainingsmethode Supervised Learning aufbereitete Daten von gespielten Jassspielen benötigt, wurde ein grosses schweizer Online-Jassportal angefragt. Dieses stellte Daten von 5 000 Jasspartien für das Training zur Verfügung.

Technologien

Beim Einsatz von neuronalen Netzwerken im Bereich Deep Learning werden für eine effiziente Arbeitsweise hohe Anforderungen an die Entwicklungsumgebung gestellt. Für die Entwicklung der Prototypen wurden folgende bekannte Technologien aus dem Machine Learning Bereich eingesetzt:

- Hardware (Nvidia DGX-1 Supercomputer mit 8 GPUs)
- Programmiersprache (Python)
- Entwicklungsbibliotheken (TensorFlow, Keras)
- Visualisierungswerkzeug für Messungen des Lernerfolges von neuronalen Netzwerken (TensorBoard von TensorFlow)

Ergebnisse

Für beide Lernstrategien konnte je ein Prototyp entwickelt werden, welcher bereits auf dem Niveau eines Gelegenheitsspielers jassen kann. Die entwickelten Prototypen können durch die Optimierung von Parametern und durch längere Trainingszeiten noch verbessert werden. Ein weiteres interessantes Ziel ist die Entwicklung eines Prototyps, welcher die beiden Lernstrategien kombiniert. Dieser könnte in der Lage sein, gegen einen erfahrenen Jassspieler zu gewinnen.

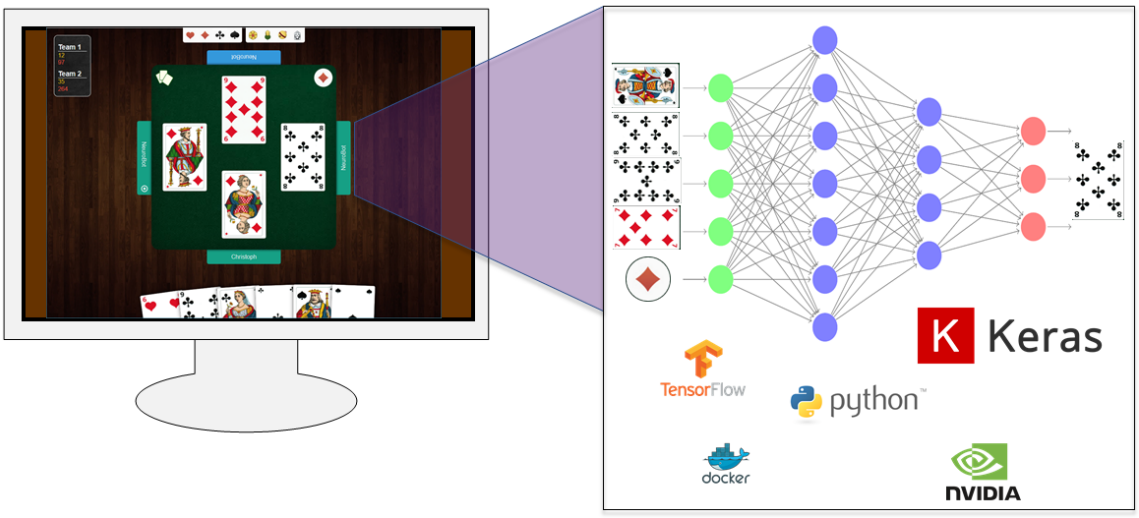


Abbildung 1.0.1: JassBot mit neuronalem Netzwerk beim Schieber Jass

2 Technischer Bericht

2.1 Einleitung

Der technische Bericht dieser Forschungsarbeit ist für Ingenieure konzipiert, welche sich mit dem Themenbereich des Machine Learnings auseinandersetzen wollen und den Entwicklungsprozess verstehen möchten, wie diese Technologie für die Lösungsfindung einer komplexen Problemstellung eingesetzt werden kann.

Für das bessere Verständnis dieser Arbeit werden im Kapitel 2.2 die verschiedenen Komponenten der Aufgabenstellung detaillierter erläutert und im Kapitel 2.4 in das Thema des Machine Learnings eingeführt. Die nachfolgenden Kapitel zeigen die Herangehensweise mittels Vorstudien, Analyse und der Entwicklung des Prototyps auf. Am Ende werden die Ergebnisse und Erkenntnisse der Experimente des Projekts nochmals zusammengefasst. Die in diesem Bericht verwendeten technischen Begrifflichkeiten aus dem Themenbereich Machine Learning werden in ihrer ursprünglichen Sprache belassen, wodurch die Assoziation mit oft verwendeter Literatur in diesem Gebiet vereinfacht wird.

Bei der Entwicklung des Prototyps und dessen Experimente wurden für die einzelnen Parameter des Netzwerkes Standardwerte verwendet, welche im Machine Learning Bereich gebraucht werden. Diese wurden dann unterschiedlich an die Ziele des Experiments und die Jassproblematik angepasst. Anschliessend wurden einzelne Konfigurationsparameter geändert, mit der ersten Lernstrategie geprüft und ausgewertet worden. Diejenigen Änderungen, die gute Resultate hervorbrachten, wurden in das Netzwerk für die zweite Lernstrategie überführt. Die Netzwerke der beiden Lernstrategien wurden stets trainiert und schliesslich auch dazu verwendet um Jassturniere durchzuführen.

> Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation
>
> *Chapter 11, Deep Learning Book[9]*

2.2 Ausgangslage

2.2.1 Jass und seine Varianten

Jass ist ein schweizweit beliebtes Kartenspiel für 4 Spieler. Jassen ist aber nicht gleich Jassen, weil unterschiedliche Spielvarianten und Blätter (Kartensets) existieren. Bei den Blättern sind das Deutsche/Deutschschweizer und das Französische Blatt die Bekanntesten, weshalb diese in der Tabelle 2.2.1 gegenübergestellt werden.

Englisch	**Französisches Blatt**	**Deutschschweizer Blatt**
Hearts	Herz	Rosen
Diamonds	Karo/Ecke	Schellen
Clubs	Kreuz	Eichel
Spades	Pik/Schaufeln	Schilten

Tabelle 2.2.1: Übersicht der verschiedenen Kartensets

Bei den Spielvarianten werden Differenzler, Schieber, Coiffeur, Molotow und weitere unterschieden, wobei sich diese in den Regeln und dem Punktezählverfahren unterscheiden. Der Schieber ist dabei die bekannteste Spielvariante, welche auch wieder in verschiedene Arten gespielt werden kann. Dabei wird entweder eine Farbe als Trumpf gewählt oder eine Spielrichtung, wie UNDEUFE, OBEABE, Slalom oder Mitte angegeben. Auf die Erklärung der detaillierten Spielregeln wird hier verzichtet, da diese auf unzähligen Webseiten[24], in Spielanleitungen und Büchern[14] nachgelesen werden können.

Falls Spielregeln in dieser Forschungsarbeit zu einer Entscheidung beitragen, so wird diese an der entsprechenden Stelle kurz aufgegriffen. Folgend wird der Ablauf eines Schieber Jassspieles aufgeführt, wodurch sogleich einige Begriffe eingeführt werden.

Beschreibung eines Schieber Jassspiels nach Swisslos[24]:
Die vier Spieler platzieren sich im Kreis, wobei diejenigen, welche sich gegenüber sitzen im selben Team sind. Der erste Spieler ist in der ersten Runde der Dealer und verteilt die 36 Karten gleichmässig und vollständig an alle Spieler.
Die Person rechts vom Dealer (Vorhand) bestimmt die Spielart (Trumpffarbe, UNDEUFE oder OBEABE). Falls der Spieler aus seinen Handkarten keinen guten Trumpf bestimmen kann, darf er auch SCHIEBE, was dazu führt, dass er die Entscheidung

an seinen Spielpartner weiterreicht, welcher dann einen Trumpf entscheiden muss. Die Person, die Vorhand hat, muss in jedem Fall die erste Karte ausspielen, danach folgt ein Spieler nach dem anderen im Gegenuhrzeigersinn. Liegen vier Karten auf dem Tisch, wird ermittelt, welcher Spieler die beste Karte gelegt hat und damit den Stich gewinnt. Der Gewinner des Stichs nimmt die Karten vom Tisch zu sich und muss die nächste Karte auf den leeren Tisch legen. Dies wiederholt sich neun Mal bis alle Spieler ihre Karten gelegt haben. Am Ende einer solchen Runde werden die Punkte pro Team zusammengezählt und notiert.
Das Amt des Dealers verschiebt sich eine Position nach rechts, wodurch nun das zweite Team die Karten verteilen muss und wiederum die Person rechts vom neuen Dealer den Trumpf wählen muss. Es werden so viele Runden gespielt, bis ein Team die vor dem Spiel vereinbarten Zielpunktzahl erreicht und somit das Spiel gewonnen hat.

2.2.2 Vorgängerprodukt

Die Idee dieser Forschungsarbeit beruht auf einem Wettbewerb, welcher die Firma Zühlke im Frühling 2017 durchgeführt hatte. Damals war die Aufgabe, einen JassBot zu entwickeln, welcher im Team gegen andere JassBots spielen kann. Dafür hatte die Firma Zühlke einen Jasssver entwickelt, auf welchem die Bots die Jassvariante Schieber spielen mussten. Das Ziel war, einen Bot zu entwickeln, welcher besser als die anderen Teams ist. Am Schluss musste der beste Bot gegen ein menschliches Team antreten, welchem er jedoch unterlag.
Der siegende Bot des Wettbewerbs wurde von einem Team entwickelt, in welchem auch ein HSR Student involviert war. Die Entwickler verfolgten dabei einen heuristischen, das heisst einen regelbasierten Lösungsansatz in der Programmiersprache Java. Dieser Bot stand dem Projektteam der vorliegenden Forschungsarbeit zur Verfügung, wodurch ein relativer Referenzwert gegen einen maschinellen Gegner geschaffen werden konnte.
Der neu zu entwickelnde JassBot soll ebenfalls auf dem öffentlich zugänglichen Jassserver[1] von Zühlke spielen können, weshalb gegen diese Schnittstelle programmiert wird. Die Auseinandersetzung mit dem Jassserver und den verfügbaren Jassclients ist im Kapitel 2.5.5 dokumentiert.

[1] https://github.com/webplatformz/challenge

2.3 Analyse der Ausgangslage

Wie aus der Aufgabenstellung ersichtlich wird, wurde zu Beginn der Forschungsarbeit noch nicht genau definiert, in welcher Form die künstliche Intelligenz des Bots existieren soll. Es gibt eine Vielzahl von möglichen Lösungsansätzen im Bereich des Machine Learning, wenn es darum geht, ein Problem zu lösen. Allerdings eignen sich nicht alle Lösungsansätze für jede Problemstellung gleich gut. Dafür muss das Problem identifiziert und klassifiziert werden.

2.3.1 Analyse der Fragestellung

Die Fragestellung beinhaltet die Entwicklung eines Bots für das Spiel Schieber Jass auf der Grundlage des Machine Learnings. Hierbei bietet es sich an, zuerst die Problemdomäne Schieber Jass genauer unter die Lupe zu nehmen.
Die Domäne wird durch den Jassserver des Jasswettbewerbs von Zühlke begrenzt. Die dort verwendete Spielvariante ist der Schieber, welcher mit vier Spielern und dem französischen oder deutschen Blatt auf 2 500 Punkte gespielt wird. Allgemein kann auf dem Server nicht gewiesen[25] werden und im Turniermodus zählen alle Trumpfarten einfach.[6]
Ein Spieler muss stets aus mehreren Karten eine auswählen, die er spielen kann. Zu Beginn kommen oftmals mehrere Karten in Frage, aus denen er sich für eine entscheiden muss. Bei dieser Entscheidung muss ein Spieler nicht bloss die Regeln berücksichtigen. Er muss auch noch abschätzen können, welche Karte in der aktuellen Situation in Bezug auf das Erreichen von möglichst vielen Punkten am sinnvollsten erscheint. Zudem muss er einschätzten können, wie der Partner unterstützt und der Gegner gestört werden kann oder ob sogar mal eine Karte geopfert werden muss für den besten Ausgang eines Spieles. Diese und noch einige weitere Aspekte müssen bei einer solchen Abschätzung der Entscheidung miteingerechnet werden.

Aus der Perspektive der Spieltheorie wird ein Spiel, bei welchem nicht allen Spielern immer alle Informationen über den Gesamtzustand des Spiels bekannt sind, ein Imperfect Information Game[19] genannt.

Mathematische Betrachtungsweise

Das Projektteam hat die Problemdomäne zuerst aus der mathematischen Sicht betrachtet, wobei sich folgende Überlegungen ergaben:
Jeder der vier Spieler erhält in jeder Runde neun Karten. Die Karten können von einem Spieler in beliebiger Reihenfolge gelegt werden, solange die Regeln eingehalten werden. Zuerst müssen immer Karten gelegt werden, welche dieselbe Farbe haben, wie die Karte, mit welcher der Stich eröffnet wurde, bevor andere gelegt werden können. Als Ausnahme gelten die Trumpfkarten. So legt jeder Spieler bei

einem Stich immer eine Karte ab, wodurch sich neun Stiche für eine Runde ergeben.

Mathematisch können wir die Voraussetzung betrachten, dass jeder Spieler 9 von 36 Karten in einer beliebigen Spielreihenfolge, abhängig von bereits gespielten Karten, ausspielt. Dies wird in der Kombinatorik als **Variation ohne Wiederholung** betrachtet. Das bedeutet, dass es sich um eine Auswahl ohne Wiederholung handelt, jedoch unter Berücksichtigung der Reihenfolge. Somit ergibt sich folgende Berechnung:

$$\frac{n!}{(n-k)!} = \frac{36!}{(36-9)!} = 34\,162\,713\,446\,400 \tag{2.1}$$

Dieses Resultat sagt aus, dass es 34 162 713 446 400 mögliche Varianten gibt, wie ein Spieler neuen beliebige Karten ablegen kann, was ungefähr 3.4 $\times$ 10^{13} oder mehr als 34 Billionen entspricht. Bei dieser Betrachtung werden jedoch die Regeln vernachlässigt.
Die Berechnung unter 2.1 zeigt auf, dass es eine Vielzahl an möglichen Varianten gibt, die zur Auswahl stehen. Wenn nun noch einige Varianten abgezogen werden, die aufgrund des Regelwerkes verboten sind, wird die Zahl der möglichen Varianten immer noch bei etwa 30 Billionen sein.

Die zweite Aufgabe, die Wahl eines Trumpfes, die ein Spieler mindestens jede vierte Runde zu lösen, muss ebenfalls analysiert werden. So wird ersichtlich, dass diese Aktion zwar weniger komplex ist, aber trotzdem immer noch eine grosse Auswahl an Möglichkeiten besitzt.
Hierbei wird die Überlegung angestellt, dass ein Spieler beliebige 9 von 36 Karten erhält, dabei spielt die Reihenfolge der Karten jedoch keine Rolle, weil der Spieler aus all seinen Handkarten eine Aktion von sieben möglichen Aktionen wählen muss. Dabei handelt es sich um die Wahl einer Trumpffarbe, OBEABE, UNDEUFE oder SCHIEBE, also die Aktion die Entscheidung seinem Partner zu überlassen. Diese Anzahl Situationen kann ebenfalls mit Hilfe der Kombinatorik berechnet werden. Hierbei wird die **Kombination ohne Wiederholung und ohne Betrachtung der Reihenfolge** der gewählten Karten beachtet. Somit ergibt sich folgende Berechnung:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{36}{9} = \frac{36!}{9!(36-9)!} = 94\,143\,280 \tag{2.2}$$

Gemäss der Berechnung unter 2.2 existieren 94 143 280 verschiedene Kartenkombinationen, die ein Spieler in seiner Hand halten kann. Aus all diesen Kombinationen muss er sich für eine von sieben Aktionen entscheiden. Falls jedoch sein Partner bereits geschoben hat, so kann er lediglich nur noch aus sechs Aktionen auswählen, da er nicht mehr schieben darf.

Technologie

Die Berechnungen unter 2.1 und 2.2, sowie die Überlegungen über die Komplexität des Spieles zur Klärung der Kartenwahl, versetzte das Projektteam in die Lage, für dieses Problem eine geeignete Lösung im Gebiet des Machine Learnings zu suchen. Alle Bereiche im Machine Learning basieren grundsätzlich auf statistischen Grundlagen. Dabei sind einige Algorithmen oder Methoden näher an der Grundlagenmathematik. Andere sind so komplex, dass bei denen die darunterliegende Mathematik bei der ersten Betrachtung gar nicht in die Statistik eingeordnet werden.

Bei der Wahl der Technologie wurden andere Lösungen von komplexen Spielproblemen analysiert. Dabei wurde vor allem die neuste Entwicklung im Bereich des Spiels Go betrachtet. Hier hatten Forscher von DeepMind, das zum Google Mutterkonzern Alphabet gehört, ihre Erkenntnisse[20] und ihren Algorithmus publiziert. Im Laufe der Arbeit wurden noch zwei Weiterentwicklungen publiziert. Im Gegensatz zu AlphaGo wurde bei AlphaGo Zero[21] auf das Training mit bereits gespielten Partien verzichtet und ein Netzwerk trainiert indem es von Beginn an mit sich selbst gespielt hatte. Später wurde der Algorithmus noch verallgemeinert und es entstand AlphaZero[18]. Dieser meisterte so nun zusätzlich zum Spiel Go noch zwei Schachvarianten.

AlphaGo verwendet ein neuronales Netzwerk in Kombination mit einem Monte Carlo Tree Search (MCTS) Algorithmus[4], um zu solch guten Ergebnisse zu gelangen. Bei AlphaGo wird der MCTS Algorithmus verwendet, um einen Zug auch in Abhängigkeit von allen zukünftigen Zügen bewerten zu können und so immer den bestmöglichen Zug auszuführen.

Technologieentscheid

Beim Jassproblem wird aus zeitlichen Gründen zuerst nur mit einem neuronalen Netzwerk ohne einen MCTS Algorithmus begonnen, da auch AlphaGo ohne einen MCTS Algorithmus bereits die bisher besten Programme für Go schlug. Im Gegensatz zu Go werden beim Jass maximal 9 Schritte vorausgeplant, da nach einer Runde die Karten wieder neu verteilt und die Strategie wieder neu gewählt werden muss. Es sollen die selben Lernstrategien (Supervised Learning und Reinforcement Learning) wie bei AlphaGo verwendet werden.

2.4 Begriffserklärung

In diesem Kapitel werden einige Begriffe eingeführt, die für das Verständnis dieser Arbeit wichtige sind. Die Begrifflichkeiten sind nach dem Ablauf von einem Neuronalen Netzwerke geordnet. Zuerst erfolgt eine allgemeine Einführung über künstliche neuronale Netzwerke und deren Aufbau im Kapitel 2.4.2. Danach folgen Begriffe bezüglich des Lernens und Optimierens eines Netzwerkes.

2.4.1 Künstliche Intelligenz

Der Begriff künstliche Intelligenz (KI, englisch *artificial intelligence*, AI) ist seit Mitte des letzten Jahrhunderts ein aufstrebendes Forschungs- und Entwicklungsgebiet. Da der Begriff und die damit verbundenen technischen und soziologischen Fragestellungen seit 60 Jahren auch in Literatur und Filmen ein beliebtes Thema sind, ist der generelle Begriff der künstlichen Intelligenz in der Bevölkerung bereits seit längerem etabliert.

Unter künstlicher Intelligenz werden diverse Technologien und Begriffe zusammengefasst, welche sich mit intelligentem Verhalten von Computern befassen. Dies kann auf unterschiedliche Weise erreicht werden. Zum einen gibt es komplexe Algorithmen, die ein scheinbar intelligentes Verhalten aufweisen, zum anderen aber auch Systeme, die neue Verhalten und neues Wissen lernen und dies weiterentwickeln können. Obwohl im Themengebiet der künstlichen Intelligenz bereits seit einigen Jahrzehnten geforscht wird, konnten die grössten praktischen Erfolge erst seit der Millenniumswende verzeichnet werden. Dieser Umstand beruht darauf, dass diese Systeme schnell sehr komplex wurden und grosse Mengen an gut lernbaren Daten benötigten. Die geforderte Leistung ist notwendig, da viele Berechnungen sowie grosse Datenmengen in einer sinnvollen, endlichen Zeit verarbeitet werden müssen. Die Beschaffung, Haltung und Analyse von derartig grossen Datenmengen, wie sie für ein System mit KI hilfreich sein kann, wurde erst durch neuere Entwicklungen wie grössere Speicherkapazitäten auf kleinerem Raum, der Verbreitung des Internets und der Leistungsfortschritte in der Hardwareentwicklung möglich.

Bei der Arbeit mit künstlicher Intelligenz stellt sich die Frage, welchen Weg eingeschlagen werden soll, da sich das Feld grob in die zwei Bereiche unterteilen lässt. Die regelbasierte Programmierung und das Maschinelle Lernen (Machine Learning). Die beiden Bereiche werden nachfolgend erläutert.

Regelbasierte Programmierung

Diese Art zur Umsetzung der künstlichen Intelligenz legte den Grundstein in diesem Forschungsgebiet. Hierbei werden Fakten und Regeln definiert. Ein Kontrollsystem, der Interpreter, identifiziert die benötigten Regeln und wendet diese an, um Anfragen zu beantworten. Des Weiteren ist er dafür verantwortlich, dass die Faktenbasis aktualisiert wird.
Eine der bekanntesten Sprachen, die nach diesem Prinzip arbeiten und für KI verwendet werden, ist Prolog. Das wohl bekannteste Beispiel dafür ist die Software Watson.

Machine Learning

Im Gegensatz zur klassischen Softwareentwicklung wird beim Machine Learning das Problem nicht durch eine wohldurchdachte Programmlogik gelöst, sondern es wird aufgrund von Trainingsdaten ein Modell, zum Beispiel ein neuronales Netzwerk, trainiert. Die Trainingsdaten werden zuvor im Umfeld des Problems erhoben und beinhalten die selben Parameter, welche auch das Modell benötigt, um seine Entscheidung zu treffen.
Im Gegensatz zu der regelbasierten Programmierung, soll beim Machine Learning nur ein Grundgerüst erstellt werden, welches durch das Verarbeiten von Daten so verändert wird, dass es am Ende die gewünschte Funktionalität abbildet. Der Kern des Machine Learning ist das sogenannte Modell, welches durch das Training immer mehr optimiert wird.
Die meisten Probleme in diesem Bereich fallen in eine der folgenden Problemkategorien[5][7]:

- **Regression:** Aufgrund der gegebenen Daten wird ein bestimmter Wert in einem kontinuierlichem Wertebereich vorhergesagt (zum Beispiel die Temperatur für morgen)
- **Klassifizierung:** Aufgrund der gegebenen Daten wird eine vorgegebene Kategorie gewählt (zum Beispiel ein Objekt in einem Bild erkennen)
- **Clustering:** Versteckte Muster werden in gegebenen Daten erkannt (zum Beispiel zur Identifizierung von unterschiedlichen Kundengruppen)
- **Assoziation:** Bestimmte Regeln werden aus den gegebenen Daten extrahiert (zum Beispiel eine Warenkorbanalyse: wer X kauft, kauft oft auch Y?)

Auch bei der Trainingsart gibt es Unterschiede. Regressions- und Klassifizierungsalgorithmen werden vorwiegend durch Supervised Learning trainiert, während die Clustering- und Assoziationsalgorithmen durch Unsupervised Learning trainiert werden.

Beim **Supervised Learning** wird die Voraussage des Algorithmus mit dem bekannten Ergebnis aus den Daten abgeglichen und aufgrund der Differenz wird der Algorithmus angepasst. Dies setzt einen gelabelten Datensatz voraus.

Im Gegensatz dazu ist beim **Unsupervised Learning** das erwartete Ergebnis unbekannt und die Daten sind nicht gelabelt. Der Algorithmus soll anhand der Daten, die er erhält, Muster oder Gruppierungen in den Daten erkennen.

Das **Reinforcement Learning** wird häufig bei Spielen eingesetzt, wie AlphaGo, AlphaGo Zero und AlphaZero zeigen. Der Vorteil beim Reinforcement Learning ist, dass keine zuvor aufgezeichnete Daten benötigt werden. Es kann dabei direkt während dem Spielen gelernt werden. Je öfters eine Situation durchlaufen wird, desto besser wird das Netzwerk in dieser Situation.

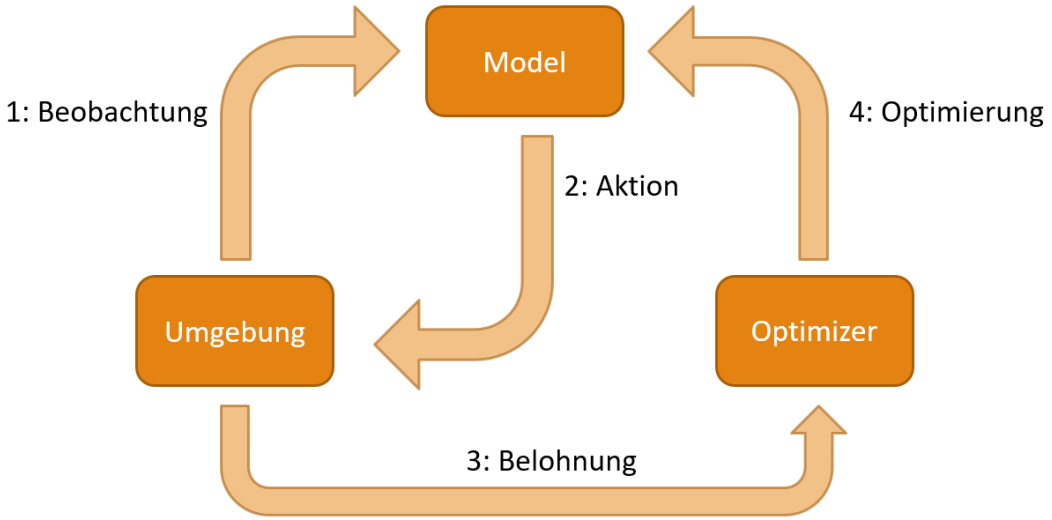


Abbildung 2.4.1: Schematische Darstellung Reinforcement Learning

Beim Reinforcement Learning interagiert das Netzwerk direkt mit der Umgebung, welche für jede Aktion, die das Netzwerk ausführt, einen Reward (Belohnung) zurückliefert. Mit Hilfe dieses Rewards kann bestummen werden, wie das Netzwerk aktualisiert werden muss, um in der selben Situation das nächste Mal eine bessere Entscheidung zu treffen.

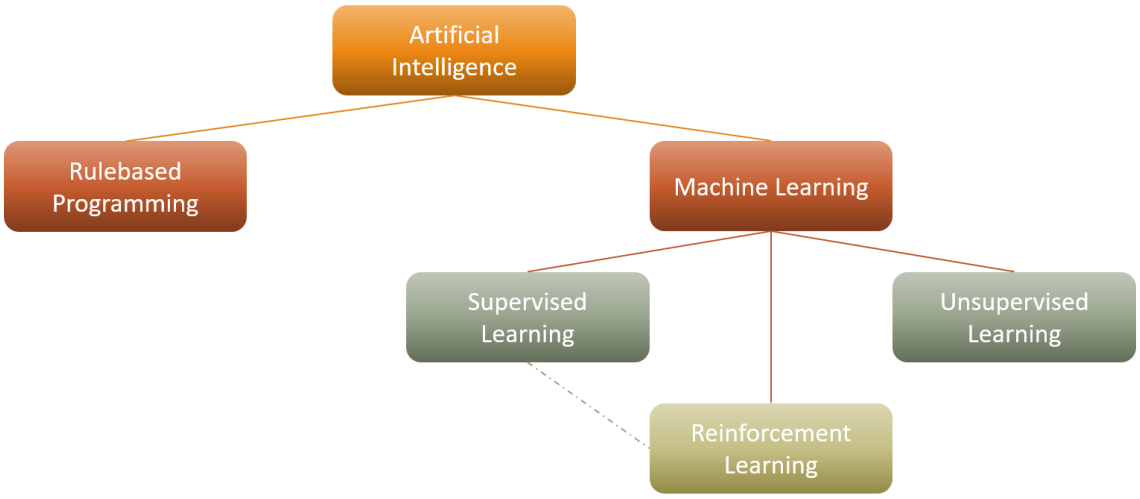


Abbildung 2.4.2: Übersicht zur Unterteilung der Künstlichen Intelligenz

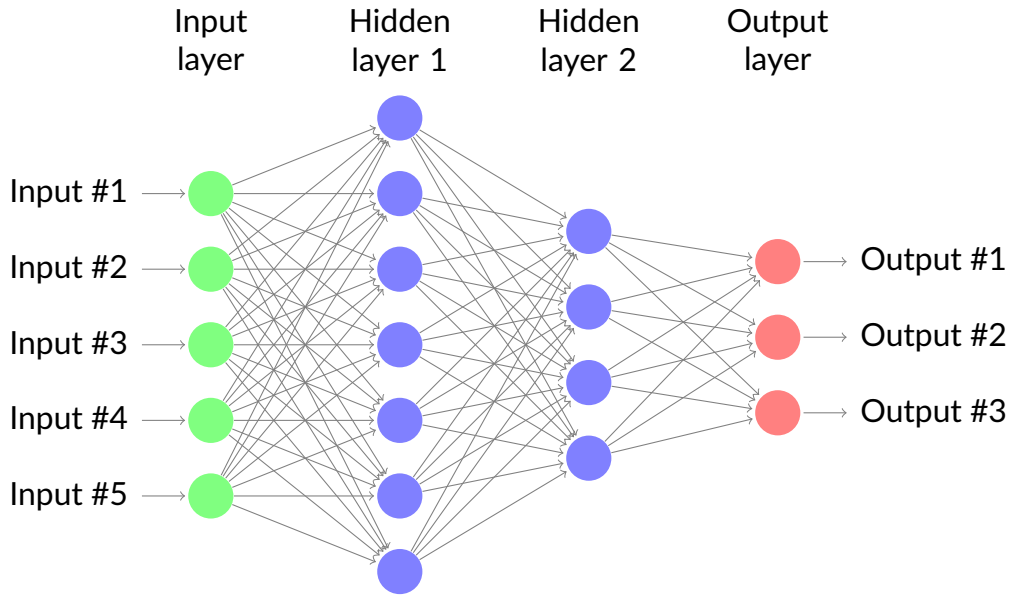
Deep Learning

Deep Learning ist eine Unterkategorie von neuronalen Netzwerken, welche wiederum eine Unterkategorie vom Bereich Machine Learning ist. Fachbücher und Artikel wie beispielsweise das **Deep Learning Book** (Introduction[9]) oder der Artikel **Scaling Learning Algorithms towards AI** (Kapitel 3[2]) über Machine Learning, neuronale Netzwerke und Deep Learning haben unterschiedliche Auffassungen und Definitionen, ab wann ein neuronales Netzwerk als Deep Learning Netzwerk bezeichnet wird. Es kann mathematisch bewiesen werden, dass jedes Problem mit einem einzigen Hidden Layer gelöst werden kann, wenn auf diesem Layer genügend Neuronen vorhanden sind.[23][10] Diese Überlegung könnte als Grundlage verwendet werden um eine mathematisch begründete Definition aufzustellen. Dabei würde jedes Netzwerk, welches einen Input Layer, nur einen Hidden Layer und einen Output Layer besitzt als einfaches neuronales Netzwerk bezeichnet. Von dieser Definition ausgehend kann dann abgeleitet werden, dass alle neuronalen Netzwerke mit mehr als einem Hidden Layer in den Bereich Deep Learning zählen würden. Oftmals wird jedoch zur Vereinfachung von neuronalen Netzwerken gesprochen, ohne eine genaue Unterscheidung zu machen.

2.4.2 Künstliche neuronale Netzwerke

Künstliche neuronale Netzwerke sind wie ihre natürlichen Vorbilder aus miteinander verbundenen Neuronen aufgebaut. Die Neuronen sind in mehreren Schichten, auch Layers genannt, angeordnet. Jedes Neuron eines Layers ist auf eine bestimmte Art mit Neuronen des nächsten Layers verbunden. Diese Verbindungen haben eine

individuelle Gewichtung, welche die Entscheidung beeinflusst. Durch das Trainieren des Netzwerkes werden diese Gewichtungen verändert. Das erlernte Wissen wird durch diese Gewichte ausgedrückt.
Solche Netzwerke werden durch verschiedene Eigenschaften von einander unterschieden. Unter anderem ist das die Art, wie Neuronen zwischen den Layern vernetzt sind, die Anzahl Neuronen pro Layer und die Anzahl Layer im Netzwerk.



2.4.3 Layers

Ein Netzwerk wird meistens in Schichten aufgebaut. Die erste Schicht ist immer der Input Layer. Dieser enthält die Eingangsvariablen für das Netzwerk. An den Input Layer schliessen sich ein oder mehrere Hidden Layers an oder direkt der Output Layer. Die letzte Schicht ist der Output Layer, an welchem dann das Resultat abgelesen werden kann.
Jeder Layer kann unterschiedlich viele Neuronen beinhalten, die zwischen den Schichten unterschiedlich miteinander verbunden sind. Die häufigste Form ist ein fully connected Layer. Bei diesem ist jedes Neuron mit jedem Neuron der vorherigen Schicht verbunden. Normalerweise sind Neuronen innerhalb des selben Layers nicht miteinander verbunden.

2.4.4 Neuronen

Ein Neuron in künstlichen neuronalen Netzwerken ist jenem im menschlichen Gehirn nachempfunden. Es bündelt die Ausgaben der mit ihm verbundenen Neuronen des vorherigen Layers. Diese Eingaben werden gewichtet, aufsummiert und in eine Aktivierungsfunktion gegeben. Diese Funktion bestimmt am Schluss den Wert, den das Neuron an das nächste weiter gibt.

2.4.5 Gewichte

Der Kern des neuronalen Netzwerks sind die Gewichte. Sie bestimmen bei jedem Neuron wie viel Gewicht jedem einzelnen ankommenden Signal gegeben wird. Zu Beginn sind alle Gewichte zufällig initialisiert. Das Ziel des Trainings ist nun, diese Gewichte so zu verändern, dass alle zusammen als neuronales Netzwerk die dem Problem zu Grunde liegende Funktion möglichst gut approximieren können.

2.4.6 Bias

Im Gegensatz zu den Gewichten ist das Bias eine konstante, welche pro Neuron angepasst werden kann. Das Bias wird im Gegensatz zu den Gewichten direkt mit den gewichteten Eingaben aufsummiert.

2.4.7 Learning Rate

Die Learning Rate bestimmt wie stark die Gewichte und Bias angepasst werden. Eine hohe Learning Rate bedeutet die Gewichte und Bias werden bei jedem Durchgang stark angepasst. Dies hat den Vorteil, dass das Netzwerk relativ schnell optimiert wird. Bei einer zu grossen Learning Rate kann es aber passieren, dass über das Minimum hinausgeschossen wird. Dies muss dann im nächsten Durchgang korrigiert werden. Da aber eine hohe Learning Rate gewählt wurde, wird jedoch wieder am Ziel vorbei korrigiert. Das heisst, mit einer grossen Learning Rate pendelt das Ergebnis immer um das Minimum herum, ohne es jemals zu erreichen. Im Gegensatz dazu bedeutet eine kleine Learning Rate natürlich eine längere Trainingszeit, bis das gewünschte Ergebnis erreicht wird. Somit ist das Wählen der Learning Rate ein Abwägen zwischen schneller Konvergenz und Erreichen des optimalen Minimums.

2.4.8 Aktivierungsfunktion

Neuronen im Gehirn geben ein Signal erst weiter, sobald die Eingangssignale einen gewissen Schwellwert überschritten haben. Diese Eigenschaft wird bei künstlichen neuronalen Netzwerken mit sogenannten Aktivierungsfunktionen abgebildet. Diese Funktionen sollten einige grundlegende Eigenschaften haben:

Sie sollten nicht linear sein, da eine Kombination aus verschiedenen linearen Funktionen immer durch eine einzige lineare Funktion abgebildet werden kann. Damit sind diese Funktionen für neuronale Netzwerke nicht geeignet.
Zudem sollten Aktivierungsfunktionen an jeder Stelle differenzierbar sein, damit für das Training das Gradientenverfahren angewendet werden kann. Zudem verhindert eine monotone Funktion lokale Minimas, welche verhindern können, dass eine Optimierungsfunktion das globale Minimum und somit die beste Approximation findet.

Sigmoid

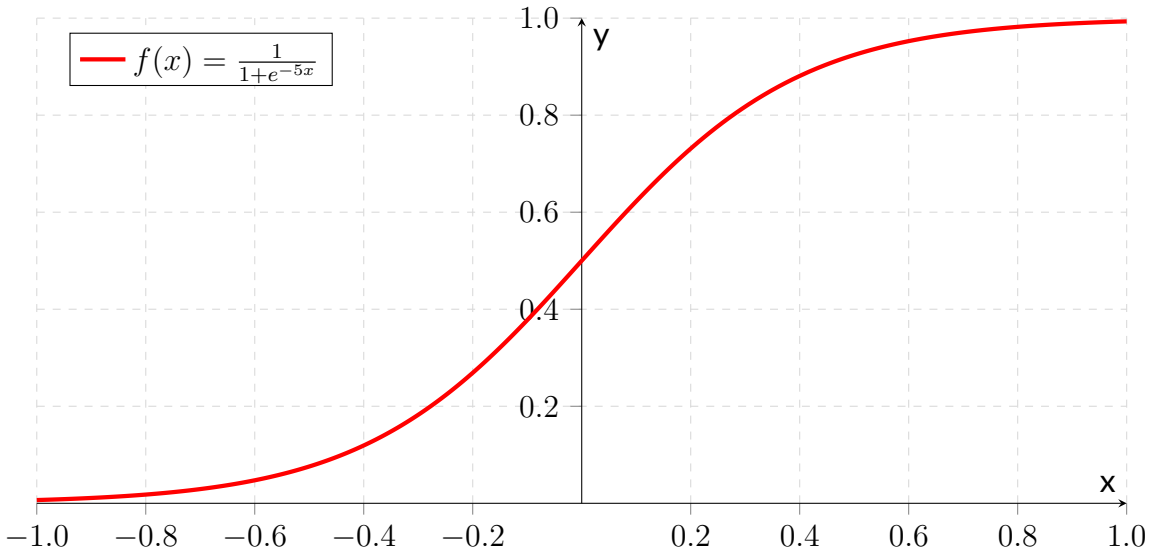


Abbildung 2.4.3: Sigmoid Funktion

Die Sigmoid Funktion (siehe Abb. 2.4.3) ist eine monotone Funktion, deren Ergebnisse immer zwischen 0 und 1 liegen. Diese Eigenschaft und dass die Funktion nicht linear ist, machen sie zu einer beliebten Aktivierungsfunktion.
Da beim Trainieren eines Netzwerkes die Ableitung der Aktivierungsfunktion eine wichtige Rolle spielt, kann bei der Sigmoid Funktion ein Problem auftreten, wenn x extrem wird, da sich dann die Ableitung gegen 0 verkleinert. Es tritt bei bestimmten Aktivierungsfunktionen in Kombination mit der gradientenbasierten Optimierung

auf. Dadurch kann es vorkommen, dass einzelne Layer langsamer trainiert werden als andere und dadurch die Lernfähigkeit des Netzwerkes verringert wird.
Das Problem des verschwindenden Gradienten ist ein Problem bei Aktivierungsfunktionen, die den Eingabewertebereich auf einen relativ kleinen Ausgabewertebereich (z.B. zwischen 0 und 1) reduzieren. Je mehr Layer mit einer solchen Aktivierungsfunktion das Netzwerk nun besitzt, desto grösser wird das Problem, da der Eingangswertebereich immer wieder auf einen noch kleineren Bereich reduziert wird. Dadurch beeinflusst auch eine grosse Änderung an den Eingabewerten die Ausgabe nur sehr minimal. Beim Trainieren ist somit eine Anpassung der Gewichte der hinteren Layer für den Optimierungsalgorithmus viel effektiver als Anpassungen an den vorderen Layern, wodurch schlussendlich die vorderen Layer kaum mehr Anpassungen an ihren Gewichten erfahren. Durch diese Einschränkung sinkt die Kapazität des Netzwerkes, da einige Layer kaum mehr für das zu lösende Problem optimiert werden können.

ReLU

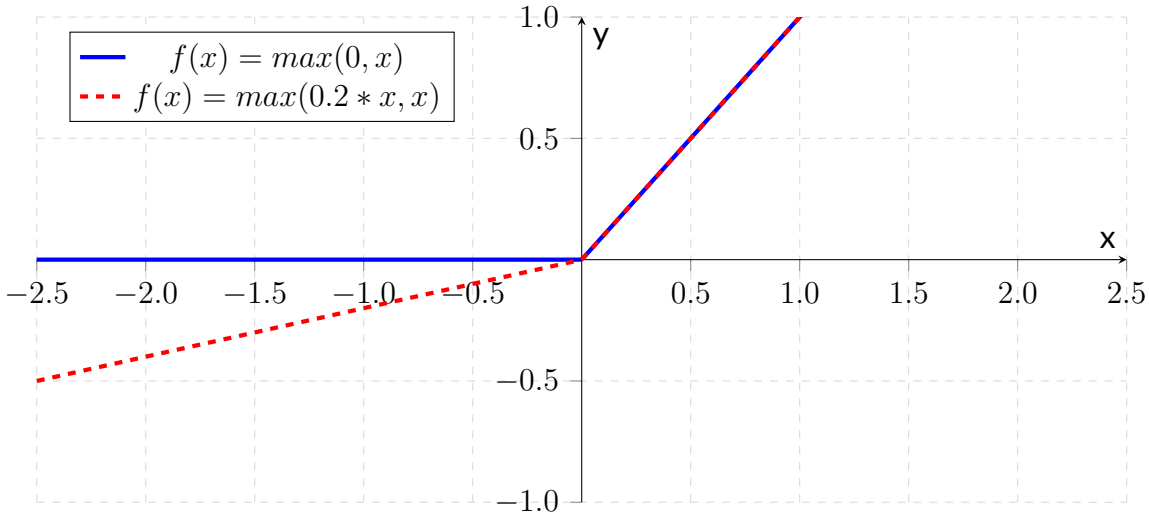


Abbildung 2.4.4: ReLU (blau), leaky ReLU (rot)

Eine **rectified linear unit** (blau in Abb.) ist eine abgeschnittene Identitätsfunktion, welche im negativen Bereich immer 0 ist. Da sie nun nicht mehr linear ist, kann sie gut als Aktivierungsfunktion verwendet werden. Auch das Problem des verschwindenden Gradienten ist durch die permanente Steigung von 1 gelöst. Dadurch wurden ReLUs als Aktivierungsfunktionen immer beliebter.
Ein Nachteil der ReLUs ist der negative Bereich. Durch die Art wie neuronale Netzwerke trainiert werden, wird ein Neuron, dessen Ergebnis der Aktivierungsfunktion

0 war, beim Training in diesem Fall nicht angepasst, da es in diesem Fall nichts zur Lösung beigetragen hat. So kann es vorkommen, dass zum Beispiel bei einer zu grossen Learning Rate grosse Teile des Netzwerkes absterben. Dieses Phänomen nennt man **dying ReLUs**.
Um dem entgegenzuwirken werden leaky ReLUs (rot) verwendet. In der Abbildung 2.4.4 ist zu erkennen, dass sich die beiden ReLU Funktionen auf der positiven Seite nicht unterscheiden. Nur im negativen Bereich wird für leaky ReLU eine kleine Steigung eingebaut, sodass negative Werte nicht genau 0 sind und das Neuron beim Training wieder berücksichtigt wird.

2.4.9 Lossfunktion

Damit nun ein Netzwerk überhaupt etwas lernen kann, muss ein aussagekräftiges Feedback berechnet werden. Ein gutes Feedback zeigt auf, ob und wie weit ein gegebenes Resultat vom tatsächlichen Resultat abweicht. Die Aufgabe einer Lossfunktion ist es, einen solchen Losswert zu berechnen. Dabei gibt es verschiedene Lossfunktionen für verschiedene Typen von Resultaten.
Die bekannteste Lossfunktion ist der mean squared error (MSE), welcher oft zu Beginn verwendet wird. Typischerweise wird bei Klassifizierungsproblemen zu einem fortgeschrittenen Zeitpunkt auf die Lossfunktion categorical crossentropy (dt. kategorische Kreuzentropie) gewechselt.[8]

2.4.10 Backpropagation

Nachdem durch die Lossfunktion bekannt ist, wie das Resultat vom Erwarteten abweicht, kann mittels **Backpropagation** berechnet werden, wie jedes einzelne Gewicht und Bias verändert werden soll, damit später in derselben Situation das Resultat näher am erwarteten Ergebnis liegt. Das heisst, dass Gewichte und Bias von Neuronen die nichts oder nur sehr wenig zum Endergebnis beigetragen haben, gar nicht oder nur minimal angepasst werden. Im Gegensatz dazu werden Gewichte und Bias von stark involvierten Neuronen stärker angepasst. Die Aktivität eines Neurons ist in jeder Situation unterschiedlich, dies bedeutet, dass in jedem Durchgang andere Neuronen mehr und weniger angepasst werden.
Diese Veränderung der Gewichte und Bias, welche mittels Backpropagation berechnet werden, heissen Gradienten. Dies kann sich vorgestellt werden, wie eine unebene Oberfläche, auf der das Minimum gefunden werden soll.

2.4.11 Optimierungsfunktionen

Bei der Optimierung werden die einzelnen Gewichte und Bias mit Hilfe der mit Backpropagation berechneten Gradienten angepasst.

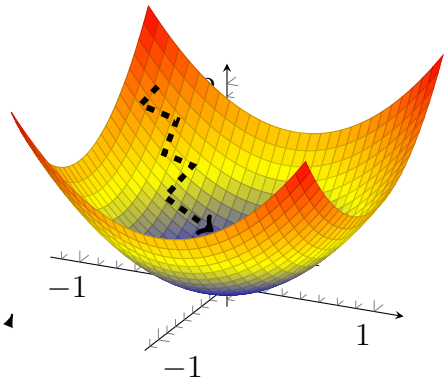


Abbildung 2.4.5: 3D Visualisierung eines Gradientenverfahrens

Auch hier gibt es verschiedene Optimierungsfunktionen (Deep Learning Book[9], Kapitel 8.3 und 8.5). Eine der Bekanntesten ist der stochastic gradient descent (SGD). Diese Funktion nutzt die Ableitung des Resultats zusammen mit der Lossfunktion und folgt dann der Neigung, um sich so über mehrere Iterationen immer mehr dem Minimum zu nähern.
Eine Alternative zum normalen SGD ist Adam[13]. Adam ist ein relativ neuer Algorithmus, der im Vergleich zum SGD eine effizientere Optimierung ermöglichen kann.

2.4.12 Credit assignment problem

Das **credit assignment problem** wird im Forschungsgebiet der neuronalen Netzwerke in zwei unterschiedlichen Bereichen verwendet.
Zum einen beschreibt es, auf das Netzwerk selbst bezogen, die Problematik wie Neuronen, welche unterschiedlich viel zum Endergebnis beigetragen, haben auch unterschiedlich optimiert werden müssen. Bei modernen Machine Learning Frameworks wie TensorFlow wird dieses Problem intern mit einem error backpropagation Algorithmus gelöst und steht deshalb im Kontext dieser Arbeit nicht im Fokus. Dabei werden die Verbindungen der Neuronen des künstlichen neuronalen Netzwerkes mit Hilfe des Losswertes rückwirkend angepasst.
Zum anderen tritt das credit assignment problem beim Reinforcement Learning auf, welches in Spielen sehr stark verbreitet ist. Dabei beschreibt es das Problem, wenn eine gewählte Aktion nicht sofort zu einer Bewertung führt, sondern diese erst nach

mehreren Aktionen berechnet werden kann. Hier stellt sich auch die Frage, welche Entscheidungen wie viel zur Lösung des Problems beigetragen haben, sowie welche gut oder schlecht waren. Eine Lösung ist das Temporal Difference Learning.

Temporal Difference Learning

Dabei wird zusätzlich zum aktuellen Reward auch der zukünftig erwartete Reward beim Lernen berücksichtigt und so eine gewisse Voraussicht des Netzwerkes trainiert. In der Praxis wird dazu die Ausgangslage und die daraus resultierende Entscheidung gespeichert. Später, sobald der Reward bekannt ist, wird dieser hinzugefügt und damit das Netzwerk trainiert.
Nun soll ein Netzwerk aber nicht nur den aktuellen Reward berücksichtigen, sondern eine Aktion wählen, welche auch in der Zukunft zu guten Ergebnissen führen wird. So sollte zum Beispiel eine Aktion mit niedrigerem sofortigem Reward gewählt werden, falls mit dieser Entscheidung in der Zukunft ein hoher Reward erwartet werden kann.
Um das Netzwerk darauf zu sensibilisieren, kommt das sogenannte Temporal Difference Learning zur Anwendung. Dabei wird zusätzlich zum aktuellen Reward noch versucht, die in der Zukunft erwarteten Rewards miteinzubeziehen. Die zukünftig erwarteten Rewards werden hierbei mit einem Discountfaktor multipliziert, der immer grösser wird, je weiter in der Zukunft eine Voraussage liegt. Mathematisch kann dies folgendermassen ausgedrückt werden:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.3}$$

In der Berechnung 2.3 ist R der berechnete Reward, welcher sich aus dem aktuellen Reward r und den addierten zukünftigen Rewards zusammensetzt. t steht für die Zeit und γ ist hierbei der Discountfaktor.

2.4.13 Training

Da nun die wichtigsten Begriffe erklärt wurden, folgt nun noch eine Erläuterung zum Training. Beim Trainig wird mit Daten und den oben beschriebenen Funktionen versucht, das Netzwerk zu optimieren, damit es am Schluss Vorhersagen mit einer hohen Genauigkeit (accuracy) machen kann.
Um solche Indikatoren zu messen und diese nicht zu verfälschen, werden die Daten vor dem Training in Trainingsdaten und Testdaten aufgeteilt. Mit den Trainingsdaten wird das Netzwerk nun trainiert und es versucht diese Daten möglichst gut zu approximieren. Die Testdaten werden nicht für das Training verwendet, sondern dienen dazu, die Performance des Netzwerkes mit unabhängigen Daten überprüfen zu können. Durch das Überprüfen mit den Testdaten können Indikatoren wie die Genauigkeit oder der Losswert ermittelt werden. Mit dem Training wird eine hohe

Genauigkeit und ein kleiner Losswert angestrebt.

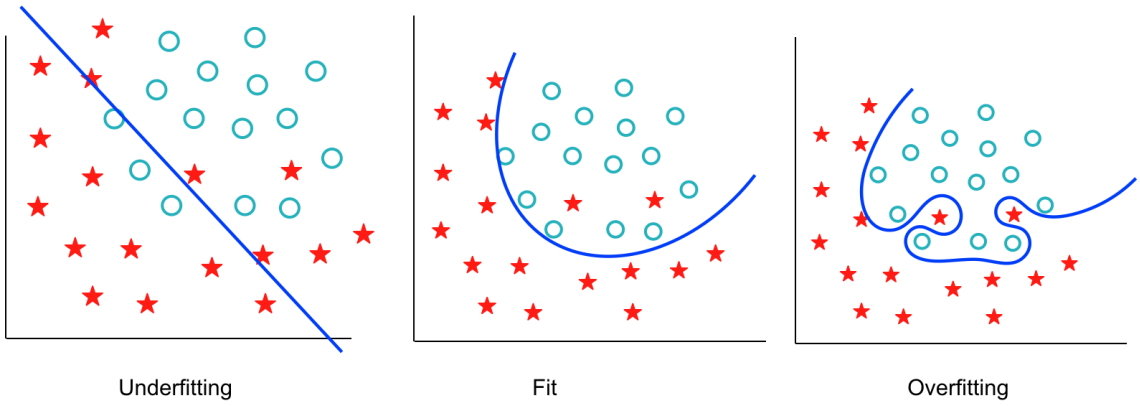


Abbildung 2.4.6: Under- und Overfitting[1]

Overfitting

Vor allem der Losswert kann dabei helfen ein Overfitting zu erkennen. Ist ein Netzwerk genug flexibel und mit genügend Daten optimiert worden, kann es vorkommen, dass sich das Netzwerk zu sehr den Trainingsdaten anpasst. Dadurch ist es im Training sehr genau, kann das gelernte jedoch schlecht auf neue Situationen anwenden. Das heisst, dass das Netzwerk zwar gute Werte liefert, aber ausserhalb der Trainingsdaten nicht verwendet werden kann. Ein solches Overfitting äussert sich dadurch, dass der Losswert im Training kontinuierlich sinkt, aber mit den Testdaten plötzlich wieder zu steigen beginnt. Overfitting kann ein hilfreicher Indikator sein, dass das Netzwerk die nötige Komplexität besitzt, um die Daten zu approximieren. Nachdem ein Overfitting beobachtet wurde, kann ein Snapshot des Netzwerk vor dem Overfitting gemacht werden und beispielsweise mit einer kleineren Learning Rate weiter trainiert werden, um noch bessere Resultate zu erreichen.

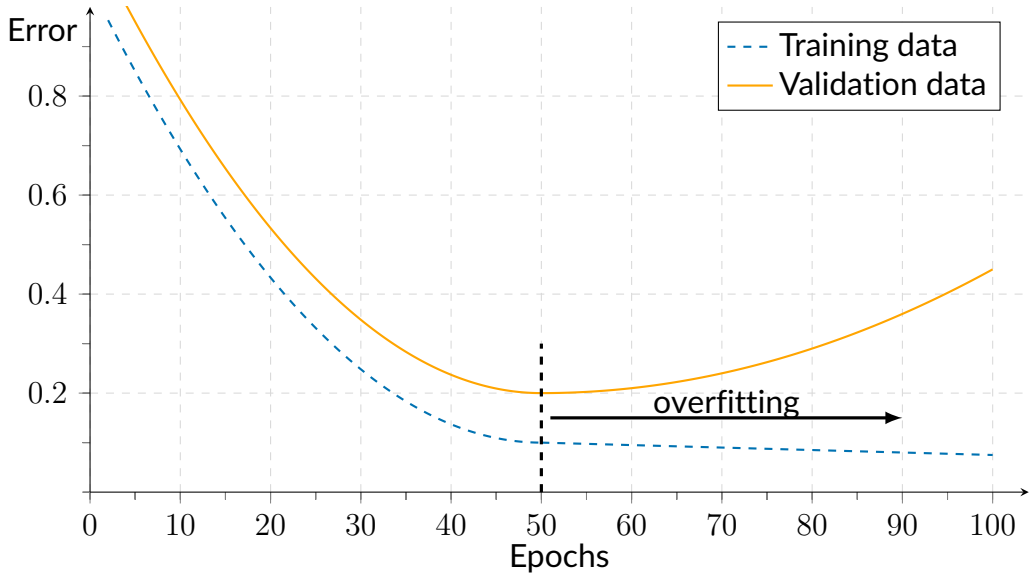


Abbildung 2.4.7: Overfitting

Underfitting

Als Underfitting wird ein Netzwerk bezeichnet, welches nicht genügend gross ist und nicht die nötige Flexibilität besitzt, um die Daten genügend zu approximieren. Dies bedeutet, das Netzwerk kann mit dieser Konfiguration die Realität nicht genügend genau abbilden und kann deshalb nicht verwendet werden. Um einem Underfitting entgegenzuwirken sollte das Netzwerk vergrössert werden. Je nach Situation sollten auch mehr Trainingsdaten verwendet werden.

2.5 Vorbereitende Arbeiten

Im folgenden Kapitel werden alle wichtigen Arbeiten aufgeführt, die zur Vorbereitung der Experimente für den Prototyp stattgefunden haben. Wie in der Tabelle 2.5.1 sichtbar, wurden mit Hilfe dieser Vorstudien gezielt Risiken angegangen, behandelt und behoben, welche in der Projektplanung evaluiert wurden.
In den einzelnen Vorstudien ist jeweils das Risiko aufgeführt, welches durch diese minimiert werden sollte. Nach den Vorstudien, im Kapitel 2.5.6, werden alle Risiken in einer Übersicht nochmals analysiert und neu bewertet.

Vorstudie	**Titel**	**Kurzbeschreibung**	**Abgedeckte Risiken**
1	Frozen Lake	Kennenlernen von TensorFlow (Kapitel 2.5.1)	Risiko 3, Risiko 7
2	Taxi	Machine Learning Spiel auf DGX Server in Betrieb nehmen, TensorBoard kennenlernen (Kapitel 2.5.2)	Risiko 3
3	Blackjack	Kennenlernen von Keras (Kapitel 2.5.3)	Risiko 3, Risiko 4
4	Qualität und Quantität der Daten	Daten von gespielten Jass-Partien beschaffen und analysieren (Kapitel 2.5.4)	Risiko 1, Risiko 2
5	Jass Server und Jass Clients	Inbetriebnahme von Jass Server und Jass Clients (Kapitel 2.5.5)	Risiko 9

Tabelle 2.5.1: Übersichtstabelle Vorstudien

In der Tabelle 2.5.2 ist eine Übersicht mit den Risiken gemäss Projektplan aus Anhang **??**.

Risiko	**Titel**
R1	Zu wenige Daten
R2	Schlechte Datenqualität (unvollständig)
R3	Fehlendes Know-how bezüglich Machine Learning
R4	Komplexität des Problems
R5	Machine Learning Model nicht optimal
R6	Ausfall der Infrastruktur (Laptop, Testserver, DGX Server)
R7	Credit assignment problem
R8	Jassregeln werden missachtet
R9	Jassserver (Zühlke) hat Bugs

Tabelle 2.5.2: Übersichtstabelle Risiken

2.5.1 Vorstudie 1: Frozen Lake

In dieser ersten Vorstudie wurde der Fokus vor allem auf die Machine Learning Bibliothek TensorFlow gelegt.

Ausgangslage

Zuvor wurde sehr viel theoretisches Wissen über Machine Learning angeeignet. Vor allem das Grundverständnis und die verwendeten Begrifflichkeiten im Bereich Machine Learning wurden nachgeschlagen und vertieft.

Ziel

Das Ziel dieser Vorstudie war, das zuvor erworbene theoretische Wissen nun anhand eines konkreten Beispiels anzuwenden. Diese Vorstudie bot die Möglichkeit, TensorFlow nun auch von der praktischen Seite kennen zu lernen und eventuelle Wissenslücken in diesem Bereich zu identifizieren.

Abzudeckendes Risiko

Der Fokus dieser Vorstudie lag auf dem Minimieren des **Risiko 3: Fehlendes Know-how bezüglich Machine Learning** und dem Sammeln erster Erkenntnisse im Bereich des **Risiko 7: Credit assignment problem**.

Problemstellung

Als zu lösendes Problem wurde die Frozen Lake[2] Umgebung des OpenAI Gym gewählt. Das Szenario ist ein gefrorener See mit den Dimensionen 4x4, den es zu überqueren gilt. Dabei ist der See aber nicht komplett gefroren, sondern hat einzelne Löcher, welche vermieden werden sollen.

Vorgehen

Überlegungen

Da viele andere Frameworks, wie zum Beispiel Keras, auf TensorFlow aufbauen, wurde in diesem ersten Schritt ein Verständnis über die Grundbibliothek TensorFlow verschafft, um dann in weiteren Schritten höhere Abstraktionsebenen wie Keras darauf aufbauend zu verstehen.

[2] https://gym.openai.com/envs/FrozenLake-v0/

Hilfsmittel

Da es in dieser Vorstudie vorerst nur um erste Einblicke in Machine Learning Code ging, wurde die Lösung des Frozen Lake Problems kopiert und analysiert. Die Lösung[3] stammt von Arthur Juliani, der mehrere Blogposts über Reinforcement Learning mit TensorFlow veröffentlicht hat.
Für die Analyse und das anschliessende Dokumentieren der Erkenntnisse wurde der Code in einem Jupyter Notebook (siehe Anhang Jupyter Notebook Vorstudie 1) geladen und ausgeführt.

Umsetzung

Im Jupiter Notebook konnte nun jeder einzelne Schritt genauer untersucht werden. Dabei lag der Fokus vor allem auf dem Erstellen des Netzwerkes, also wie ein solches Netzwerk in TensorFlow überhaupt aufgebaut und die diversen Parameter konfiguriert werden können. Ein weiterer wichtiger Punkt war anschliessend das Berechnen des Losswertes, welcher essentiell für die spätere Optimierung ist. Danach waren alle Bausteine definiert und die Analyse der tatsächlichen Ausführung und Optimierung des Netzwerkes konnte beginnen. In diesem Schritt war vor allem von Interesse, wie die zuvor definierten Funktionen und das Netzwerk in einer Session aufgerufen, mit Daten befüllt, evaluiert und zum Schluss trainiert werden konnten.
Bei der Auseinandersetzung mit TensorFlow kamen auch noch einige Fragen auf, welche recherchiert und dokumentiert werden mussten. So war zuerst nicht ganz ersichtlich, wie das Netzwerk schlussendlich trainiert wird, weil dabei vieles im Hintergrund passiert und im Code selber nur ein einziger Funktionsaufruf nötig ist. An diesem Beispiel wurde auch klar, was der Vorteil von TensorFlow ist und wie viel Arbeit diese Bibliothek übernimmt.

Ergebnis

Die Analyse dieses Problems mit einer existierenden Lösung war sehr aufschlussreich, da nun die Verbindung zwischen Theorie und Implementierung gemacht werden konnte. Zuvor war es schwierig, sich vorzustellen, wie die Theorie nun konkret angewendet werden kann. Im Rahmen dieser Vorstudie mussten auch neue Probleme recherchiert werden, welche sich erst durch die konkrete Anwendung ergaben. Sehr hilfreich war auch zu sehen, welche Aufgaben bereits von TensorFlow übernommen wurden und daraus, auf welche Punkte man sich bei der Umsetzung konzentrieren kann.
Da der Aufbau im Grunde immer sehr ähnlich ist, half diese Analyse auch um eine erste Übersicht über die verschiedenen Teile eines solchen Programms zu erhal-

[3] https://gist.github.com/awjuliani/4d69edad4d0ed9a5884f3cdcf0ea0874#file-q-net-learning-clean-ipynb

ten und sich eine konkretere Vorstellung davon zu machen, welche Elemente es braucht, um ein Netzwerk aufzubauen, zu trainieren und zu optimieren.

Die Hauptpunkte, welche aus dieser Vorstudie mitgenommen wurden, sind:

- wie Netzwerke definiert werden,
- wie Lossfunktionen implementiert werden,
- wie dem Netzwerk Werte übergeben werden,
- wie das Netzwerk ausgewertet wird und
- welche Parameter und Funktionen für die Optimierung eines Netzwerkes benötigt werden.

Es konnten **Risiko 3: Fehlendes Know-how bezüglich Machine Learning** und **Risiko 7: Credit assignment problem** minimiert werden.

2.5.2 Vorstudie 2: Taxi

Nachdem in der ersten Vorstudie die Grundlagen in TensorFlow erarbeitet worden sind, geht es nun um die ersten eigenen Schritte mit TensorFlow, um das Loggen von Parametern und um die Inbetriebnahme eines neuronalen Netzwerkes auf dem DGX Server[15].

Ausgangslage

Bei dieser Vorstudie wurde das Taxi Problem[4] aus der Sammlung des OpenAI Gym gewählt.

Ziel

In dieser Vorstudie soll die Lösung der ersten Vorstudie so angepasst werden, dass dieses neue Problem gelöst werden kann. Zusätzlich soll eine Auswertung bestimmter Parameter mit TensorBoard erfolgen, um den Lernfortschritt des Netzwerkes überwachen zu können. Sobald diese beiden Ziele erreicht sind, kann das Netzwerk auf dem DGX Server zum Laufen gebracht werden, um zu bestätigen, dass auch in dieser Umgebung alles ordnungsgemäss funktioniert und verstanden wurde.

Abzudeckendes Risiko

Da Machine Learning ein sehr grosses und komplexes Themengebiet ist, wird diese Vorstudie ebenfalls im Bereich des **Risiko 3: Fehlendes KnowHow bezüglich Machine Learning** angesiedelt.

Problemstellung

Das Taxi Problem ist ein 5x5 Spielfeld mit 4 markierten Positionen. Nun muss das vom Spieler gesteuerte Taxi einen Passagier von einer bestimmten Position abholen und zu einer anderen Position bringen und abladen. Es gibt jeweils Pluspunkte für das korrekte Auf- und Abladen eines Passagiers und Minuspunkte, falls an einem falschen Ort versucht wird ein Passagier auf- oder abzuladen.

[4] https://gym.openai.com/envs/Taxi-v1/

Vorgehen

Überlegungen

Da es sich bei diesem Problem bezüglich den Parametern am Input und Output um ein sehr ähnliches Problem wie das Frozen Lake handelt, sollte das Netzwerk des vorherigen Problems mit einigen Anpassungen auch diese Spielumgebung meistern können. Mit einigen Ergänzungen kann auch zum ersten Mal eine Visualisierung des Lernfortschrittes mit TensorBoard getestet werden. Durch das Trainieren des Netzwerkes auf dem DGX Server können im gleichen Schritt noch Erfahrungen mit der Umgebung auf dem DGX Server gesammelt werden.

Hilfsmittel

In dieser Vorstudie wurde die Lösung aus der Vorstudie 1 im Kapitel 2.5.1 als Ausgangslage verwendet. Des Weiteren wurde das Benutzerhandbuch des DGX Servers zur Hilfe genommen. Diese Vorstudie wurde ebenfalls in einem Jupyter Notebook (siehe Anhang Jupyter Notebook Vorstudie 2) erfasst und dokumentiert.

Umsetzung

Begonnen wurde mit dem Adaptieren der Architektur des Netzwerkes aus der Vorstudie 1 für die neue Problemstellung. Dabei konnten Teile wie das Berechnen des Losswertes und das Evaluieren und Trainieren des Netzwerkes übernommen werden. Der Input und Output Layer mussten, wegen des grösseren Spielfeldes angepasst werden. Für das Logging mit TensorBoard wurde der Code um ein paar Funktionen ergänzt. In dieser Version werden der Losswert und das Ergebnis der einzelnen Spiele aufgezeichnet und im TensorBoard visualisiert.
Um dies auf dem DGX Server laufen zu lassen, musste zuerst ein passender Docker Container gefunden werden, der sowohl Python 3 als auch TensorFlow installiert hat. Dieser konnte später auch für die Entwicklung des Prototyps verwendet werden. Nachdem alle relevanten Dateien in den Container kopiert wurden, konnte das Netzwerk trainiert und anschliessend die Aufzeichnungen im TensorBoard visualisiert und ausgewertet werden.

Ergebnis

Die Adaptierung der Architektur des Netzwerkes war erstaunlich einfach und auch die Anwendung des Netzwerkes war bei diesem Problem sehr erfolgreich. Die erste Auswertung der Parameter mit TensorBoard gelang auf Anhieb. Einzig beim Aufsetzen der Umgebung auf dem DGX Server traten zu Beginn Probleme auf, da Images der Container im Repository von NVIDIA für diesen Server nur TensorFlow in Kombination mit Python 2.7 unterstützten. Das nachträgliche Installieren aufgrund der

unterschiedlichen CUDA Treiber Versionen stellte sich als sehr zeitaufwendig heraus, weshalb nach anderen Lösungen gesucht wurde. Ein passendes Image mit den korrekt installierten Python und TensorFlow Versionen wurde auf Docker Hub von TensorFlow zur Verfügung gestellt. Nach dem Start dieses Containers konnte das Netzwerk ohne weitere Probleme trainiert und ausgewertet werden. Da TensorBoard mit TensorFlow mitausgeliefert wird, muss dies nicht separat installiert werden, sondern kann sofort verwendet werden.

Schlussfolgerung

Es existiert nun eine funktionierende Umgebung für das Trainieren von Netzwerken, die auch getestet wurde. Gleichzeitig wurden erste Erfahrungen mit dem Aufzeichnen von Parametern und der Auswertung dieser Aufzeichnungen gemacht.

2.5.3 Vorstudie 3: Blackjack

Mit dieser Vorstudie machte das Projektteam die ersten Erfahrungen für das Erstellen eines neuronalen Netzwerkes mittels Keras. Ausserdem wurden beim Blackjack ähnliche Problemelemente entdeckt, die auch das Jassspiel aufweist.

Ausgangslage

Bisher wurden nur Experimente durchgeführt, bei welchen das neuronale Netzwerk direkt mit TensorFlow aufgebaut wurde. Dabei handelt es sich um eine Programmierbibliothek mit welcher maschinelles Lernen realisiert werden kann. Jedoch ist das Erstellen eines neuronalen Netzwerkes immer noch relativ aufwändig und repetitiv. Keras bietet hier Funktionen an, die bereits gängige Standardwerte verwenden. So müssen nicht alle Parameter selbst optimiert werden.

Ziel

Dieses Experiment gibt einen Einblick in die Verwendung von Keras und ermittelt, ob Keras für diese Forschungsarbeit geeignet ist.
Es soll ermittelt werden, ob die vermutete Problemverwandtschaft zwischen Jass und Blackjack existiert und ob daraus Erkenntnisse abgeleitet werden können.

Abzudeckendes Risiko

Das **Risiko 3: Fehlendes Know-how bezüglich Machine Learning** wird durch das Analysieren der Lösung des Spieles Blackjack weiter verringert. Es wurde ein Spiel gewählt, bei dem Parallelen zum Jassspiel vermutet wurden, um das **Risiko 4: Komplexität des Problems** in einem ersten Schritt genauer unter die Lupe zu nehmen.

Problemstellung

Die Regeln und auch das Ziel des Spieles stammen vom allgemein bekannten Karten-Glücksspiel Blackjack [5]. Um zu gewinnen, muss der Spieler mit den erhaltenen Karten so genau wie möglich die Punktzahl 21 erreichen, darf jedoch nicht höher sein als dieser Wert, muss aber gleichzeitig eine höhere Summe besitzen als die Bank, sprich der Dealer.

[5] http://www.bicyclecards.com/how-to-play/blackjack/

Vorgehen

Überlegungen

Keras stellt für die Arbeit mit neuronalen Netzwerken eine API zur Verfügung, welche den Entwicklern Arbeit abnimmt, da sie eine weitere Abstraktionsschicht zu einer darunterliegenden Bibliothek für maschinelles Lernen verkörpert. Der Vorteil von Keras besteht darin, dass es auf unterschiedlichen Bibliotheken lauffähig ist, wie TensorFlow, CNTK und Theano.
Blackjack als Spiel besitzt von den Spieleigenschaften eine wichtige Parallele zum Jass. Die Spieler müssen bei beiden Spielen Entscheidungen treffen, ohne den Gesamtzustand des Spiels zu kennen. In beiden Kartenspielen besitzen die Spieler nur Wissen über ihre eigenen und die bereits gespielten Karten, jedoch beeinflussen auch die nicht gespielten Karten den Fortgang des Spieles und somit ihre Entscheidung. Dieses Problem lässt sich unter der Kategorie **Imperfect Information Games** einordnen. Dies ist ein wichtiger Unterschied zwischen der Problemstellung vom Jassspiel zu Schach oder Go, bei welchen alle Informationen über den aktuellen Zustand des Spiels auf dem Tisch bereit liegen.

Hilfsmittel

Das gelöste Problem ist das Spiel Blackjack [6] aus der Sammlung des OpenAI Gym. Auch dieses Experiment wurde in einem Jupyter Notebook (siehe Anhang Jupyter Notebook Vorstudie 3) realisiert, getestet und dokumentiert.

Umsetzung

Die Grundlage der dokumentierten Lösung des Spieles Blackjack stammt von Simen Eide [7]. Bei der Analyse wurde die Lösung allerdings für die aktuellsten Versionen von Python und Keras optimiert und aktualisiert.
Bei der Analyse und Verbesserung wurde der Code mehrfach modifiziert, um Veränderungen und Abhängigkeiten festzustellen, wie beispielsweise eine Verschlechterung des Endergebnisses durch Ändern der Optimierungsfunktion. Dafür wurden vermehrt Ausgabebefehle eingesetzt, damit nachvollzogen werden konnte, wo was geschieht. Dabei war wichtig, welche Daten die Spielumgebung mitteilt, wie die Daten des Spieles aufbereitet werden müssen, wo und wann die Daten dem neuronalen Netzwerk übergeben werden und was das Netzwerk daraus ableitet. Auch die Datenaufbereitung für das Training wurde genau analysiert, um zu lernen, welche Methoden es in Keras gibt. Neben dem Analysieren wurde auch die Keras Do-

[6] https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py
[7] https://gist.github.com/simeneide/f3d33868c8fc36ed3934da4aa71847f0

kumentation fleissig kontaktiert, jedoch war diese teilweise etwas spärlich, daher mussten einige Begrifflichkeiten anderweitig recherchiert werden.
Schlussendlich wurde versucht ein möglichst aufschlussreiches Resultat zu erreichen, welches auch eine Dokumentation der Lösung hervorbrachte.

Ergebnis

Ein Ergebnis dieser Vorstudie ist, dass es unterschiedliche Auswirkungen auf die Laufzeit, die benötigten Ressourcen oder das Endergebnis haben kann, wenn Veränderung an der Learningrate oder auch an der Netzgrösse gemacht werden. Solche Versuche konnten dank Keras relativ schnell und einfach durchgeführt werden. Dadurch ist Keras sehr geeignet für eine Forschungsarbeit, bei viele kleine Änderungen über einen grösseren Zeitraum fällig sind.
Eine weitere Erkenntnis aus diesen Versuchen ist, dass trotz gleichbleibenden Funktionen und Parametern die Ergebnisse unterschiedlich aussehen können. Dies ist nicht nur auf die unterschiedlichen Spielrunden und deren Ergebnisse zurückzuführen, sondern wird auch durch die zufällige Initialisierung des neuronalen Netzwerkes beeinflusst.
Blackjack weist in der Lösung einige Parallelen zum Jass auf. Es können beide Probleme der Kategorie Imperfect Information Games zugeordnet werden. Diese Vorstudie zeigt, wie ein Problem dieser Kategorie mit einem neuronalen Netzwerk gelöst werden kann. Eine weitere Erkenntnis ist, dass Blackjack wie auch das Jassspiel Klassifizierungsprobleme sind.

Codeausschnitt

Für Keras muss ein weiteres Package eingebunden werden, welches wiederum eine ziemlich genaue Definition der zu verwendenden Module und Komponenten zulässt.

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.regularizers import l2
from keras.optimizers import SGD
```

Listing 2.1: Packages und Module

Folgend wird die Definition des verwendeten neuronalen Netzwerkes für das Spiel Blackjack aufgeführt:
Es besteht aus einem Input Layer mit 4 Neuronen, einem Hidden Layer mit 4 Neuronen und einem Output Layer mit einem Neuron. Die 4 Inputneuronen entsprechen

den 4 Informationen, welche die Spielumgebung zurückgibt. Der Hidden Layer ist hier relativ einfach gestaltet ist aber fully connected zum Input und Output Layer. Am Output Layer existiert ein Neuron, da sich das Netzwerk am Ende bloss für eine Aktion entscheiden muss, ob es nochmals eine weitere Karte vom Dealer erhalten will oder nicht.

```
q_model = Sequential()
q_model.add(Dense(4, input_shape=(4,), kernel_initializer='
    uniform'))
q_model.add(keras.layers.normalization.BatchNormalization())
q_model.add(Activation("relu"))
q_model.add(Dense(1, kernel_regularizer=l2(0.01)))
sgd = SGD(lr=0.005)
q_model.compile(loss='mean_squared_error', optimizer=sgd,
    metrics=['mean_squared_error'])
```

Listing 2.2: Erstellen des neuronalen Netzwerkes

Folgend ist die Datenaufbereitung für das Training des Netzwerkes ersichtlich.
Um das neuronale Netzwerk optimal zu trainieren, müssen mehrere Episoden mit mehreren Spielrunden durchgeführt werden. Damit hat das Netzwerk genügend Runden und deren Auswertungen für das Training zur Verfügung. In den einzelnen Spielrunden wird das Verhalten des aktuellen Netzwerkes verwendet, um zu spielen und möglichst zu gewinnen. Am Ende jeder Episode wird das neuronale Netzwerk mit den aufgezeichneten Spielrunden (obs_list) inklusive Auswertung (action_list) trainiert(train_on_batch()). Dadurch lernt das Netzwerk von den alten Episoden und kann die Gewichte der Neuronenverbindungen der aktuellen Auswertung anpassen.

```
x = proc_input(np.array(obs_list), action_list)
y = np.array(r)
q_model.train_on_batch(x, y)
```

Listing 2.3: Neuronales Netzwerk trainieren

Schlussfolgerung

Es wird festgehalten, dass Keras für die Erstellung und fürs Experimentieren mit vielen Einstellungsmöglichkeiten sehr geeignet ist. Konfigurationen, für die Keras keine direkte Abstraktion bietet, können wenn nötig indirekt über Keras definiert und genutzt werden. Alternativ besteht immer noch die Möglichkeit direkt mit TensorFlow das Gewünschte umzusetzen.
Blackjack und der Jass gehören zur Kategorie Imperfect Information Games und

sind beides Klassifizierungsprobleme. Beim Jass ist es eine allgemeine Klassifizierung mit mehreren Klassen, im Gegensatz zur binären Klassifizierung beim Blackjack.

2.5.4 Vorstudie 4: Qualität und Quantität der Daten

Beschaffung und Kontrolle von aufgezeichneten Daten von Online-Jassportalen.

Ausgangslage

Ziel

Es soll versucht werden, Daten von unterschiedlichen Online-Jassportalen zu beschaffen.
Die Daten sollen für eine weitere Verwendung gesichtet, geprüft und beurteilt werden.
Mit solchen Daten sollen später Netzwerke trainiert werden können.

Abzudeckendes Risiko

Mit dieser Vorstudie werden die Risiken **R1: Zu wenige Daten** und **R2: Schlechte Datenqualität (unvollständig)** bearbeitet. Bei Risiko 1 wird der Fokus auf die Quantität gesetzt, damit genügend Daten für ein ideales Training vorhanden sind. Die Qualität der Daten wird beim Risiko 2 kritisch hinterfragt.

Problemstellung

Beim Trainieren von Netzwerken ist einer der bekanntesten Ansätze das Lernen aus vorhandenen Daten. Jedoch besitzt weder die HSR noch das Projektteam solche Datensätze von aufgezeichneten Jasspartien, weshalb Online-JaspPortale angefragt werden sollen.

Vorgehen

Überlegungen

Selber Daten bei existierenden Online-Jassportalen oder von existierenden Jassservern versuchen aufzuzeichnen, würde einen massiven zusätzlichen Mehraufwand bedeuten und dadurch die Entwicklung eines Prototyps gefährden.
Daher werden Jassportale angefragt, ob diese bereit wären, Daten von gespielten Partien zur Verfügung zu stellen. Dabei ist das Projektteam von externen Parteien abhängig und weiss im Vorfeld nicht, in welcher Qualität die Aufzeichnungen vorliegen. Daher muss die Quantität und Qualität der Daten im Vorfeld verifiziert und gegebenenfalls weitere Massnahmen ergriffen werden.

Hilfestellung

Das Projektteam hat die folgenden Entwickler und Betreiber von Jassportalen bzw. Jasssoftwares nach aufgezeichneten Daten von Jassspielen angefragt:

- **Bluewin Portal** (Betreiber von https://jass.bluewin.ch/)
- **jasse.ch** (Die Jass Plattform http://www.jasse.ch/ wurde von zwei Fachhochschulabsolventen entwickelt)
- **Swisscom (Schweiz) AG** (Dachorganisation von Bluewin Portal)
- **Optobyte AG** (Entwicklungsunternehmen von bekannter Jasssoftware und verschiedenen Online-Jassspielen)
- **Swisslos Interkantonale Landeslotterie** (Betreiber des Jassportal unter https://www.swisslos.ch/)

Umsetzung

Die Antworten der angefragten Parteien fielen unterschiedlich aus. Hierbei verwies Bluewin auf ihre Dachorganisation, welche sich trotz einer separaten Anfrage nicht zurückmeldete. Die Fachhochschulabsolventen, Obtobyte und Swisslos teilten alle mit, dass sie aktuell keine Partien aufzeichnen würden. Obtobyte und Swisslos boten an, ein Logging auf ihren Systemen für die Jassart Schieber zu aktivieren und die Logfiles für die Entwicklung eines JassBots zur Verfügung zu stellen. Nach einer erneuten Kontaktaufnahme seitens dem Projektteam, meldete sich nur Swisslos zurück, welche eine Vereinbarung und anonymisierte Datenaufzeichnungen von einem Zeitraum vom 8. bis 24. Oktober 2017 zustellten.
Diese Daten wurden stichprobenartig auf Vollständigkeit untersucht und wie sie zu interpretieren sind. Dafür wurden einzelne Spiele nachgestellt und analysiert.

Ergebnis

Die gelieferten Logfiles sind nachvollziehbar und sinnvoll gegliedert, was ein Weiterverarbeiten vereinfacht. Die Daten haben eine gute Qualität und sind mehrheitlich vollständig. Es gibt ein paar einzelne Datensätze, welche unvollständig sind. Bei diesen wurden die Spiele frühzeitig durch die Benutzer abgebrochen.
Die Quantität ist für den Zeitraum zufriedenstellend, da gemäss der Vereinbarung von Swisslos von **zirka 5 000 Jasspartien** die Rede ist. Bei einer genaueren Analyse der Daten wurden folgende Kennwerte ermittelt:

- **1 202 868 Stiche**
- **133 652 Runden** (9 Stiche pro Runde, zu Beginn jeder Runde wird ein Trumpf gewählt)
- durchschnittlich ca. 26.5 Runden pro Spiel bei 5 000 Jasspartien
- **4 811 472 Stiche**, die für das Training verwendet werden können, da jedes Spiel aufgrund der beteiligten Spieler vier Sichtweisen besitzt

Mit **ca. 4.8 Millionen Stichen** kann auch die Quantität für die Entwicklung eines ersten Prototyp als ausreichend betrachtet werden.
Die Daten in den Logfiles der Swisslos und die Kommunikation des Jassservers von Zühlke weisen in der Codierung der Daten Unterschiede auf, welche bei einer Weiterverarbeitung beachtet werden müssen. Zum Beispiel wird die Farbe Herz auf dem System von Swisslos mit einer **1** und vom Zühlke-Jassserver mit einer **0** codiert.

Schlussfolgerung

Damit die Daten der Logfiles für das Training verwendet werden können, müssen die Daten korrekt zusammengestellt und aufbereitet werden. Bei der Aufbereitung muss auch die Differenz der Logik zwischen dem Swisslos-Portal und dem Zühlke-Server beachtet werden. Für diesen Vorgang wird die Entwicklung eines Parsers notwendig sein.

2.5.5 Vorstudie 5: Jass Server und Jass Clients

Im Rahmen dieser Vorstudie werden der Jass Server und die beiden zu verwendenden Jass Clients getestet.

Ausgangslage

Der Jassserver, welcher beim Jasswettbewerb von der Firma Zühlke zum Einsatz kam, überwacht und verwaltet das Spiel. Mit den Clients kommuniziert er über standardisierte Nachrichten.

Der JassBot, welcher hier unter dem Namen JavaBot geführt wird, hat den in Kapitel 2.2.2 erwähnten Jasswettbewerb gewonnen. Das Projektteam erhielt den JavaBot von den Entwicklern zur Verfügung gestellt.

Auf GitHub ist zudem ein Gerüst eines JassBots in Python verfügbar.[8] Python ist die dominierende Sprache im Machine Learning Bereich und wurde deshalb als Implementierungssprache in dieser Arbeit verwendet. Das Python Gerüst kann bereits mit dem Server kommunizieren und hat auch einen integrierten Bot, der Karten nach dem Zufallsprinzip spielt.

Ziel

Das Ziel dieser Vorstudie ist eventuelle Bugs in der Server- oder Clientsoftware zu finden.

Abzudeckendes Risiko

Durch diese Vorstudie soll das **Risiko 9: Jassserver (Zühlke) hat Bugs** verringert werden.

Problemstellung

Es soll überprüft werden, dass sowohl der JavaBot sowie auch der PythonBot mit dem Server interagieren und über diesen Turniere ausgetragen werden können.

Vorgehen

Überlegungen

Durch das Spielen eines Turniers auf dem Server mit beiden Bots können offensichtliche Bugs behoben und Weitere mehrheitlich ausgeschlossen werden.

[8] https://github.com/jakeret/elbotto

Umsetzung

Da der Server von GitHub geklont und dann mit Docker ein Image erstellt werden kann, ist dieser schnell einsatzbereit.
Beim JavaBot muss vor dem Erstellen des Images noch die Adresse des Servers angepasst werden. Da zudem noch einige Tests fehlschlugen und deshalb kein Image erstellt werden konnte, musste die Ausführung der Tests im Dockerfile unterbunden werden. Danach konnte das Image erfolgreich erstellt und der Container gestartet werden.
Da im PythonBot ein zu importierendes Modul nicht richtig referenziert worden ist, musste dies vor dem Start noch korrigiert werden. Nachdem dieser Fehler behoben wurde, konnte auch der PythonBot erfolgreich gestartet werden.

Der Jassserver und zwei Instanzen des JavaBots werden jeweils in einem separaten Docker Containern gestartet, wohingegen zwei Instanzen des PythonBots auf einem gemeinsamen Docker Container betrieben werden können. Sobald alle Bots mit dem Server verbunden sind, wird das Turnier gestartet.

Ergebnis

Nachdem diese kleinen Anfangsschwierigkeiten überwunden wurden, funktionierte alles problemlos. Somit sind der Server und der JavaBot bereit, um die Entwicklung eines Machine Learning Bots zu unterstützen und Referenzwerte zu liefern. Der Machine Learning Bot wird auf dem getesteten Python Gerüst aufbauen. Durch die Nutzung der bereits implementierten Nachrichtenverarbeitung fokussiert sich die Entwicklung ganz auf die Logik des Bots.

2.5.6 Risikobeurteilung nach Vorstudien

Risiko 1: Zu wenige Daten

Am 25. Oktober stellte Swisslos, der Betreiber eines grossen Online-Jassportales, dem Projektteam die Daten von ca. 5 000 aufgezeichneten Jasspartien zur Verfügung. Diese sollten sicher für eine erste Evaluation reichen.

Risiko 2: Schlechte Datenqualität (unvollständig)

Die bisher erhaltenen Daten sind mehrheitlich vollständig und können mit einem selbst entwickelten Parser für das Neuronale Netzwerk aufbereitet werden. Bei den unvollständigen Daten handelt es sich um einzelne, abgebrochene Spiele. Informationen aus solchen Teilspielen können ebenfalls für das Trainieren des Netzwerkes extrahiert oder einfach herausgefiltert werden.

Risiko 3: Fehlendes Know-how bezüglich Machine Learning

In den ersten Projektwochen hat sich das Team viel Know-how über Machine Learning und dessen Einsatz angeeignet und dieses bereits in ersten Vorstudien praktisch angewendet.

Risiko 4: Komplexität des Problems

Das Projektteam hat die Problemstellung des Spieles Jass in einem ersten Schritt bereits analysiert und teilt die Problemstellung nun auf unterschiedlichen Ebenen auf. Zum einen wird unterschieden zwischen dem Bestimmen eines Trumpfes und dem Legen der richtigen Karte. Des Weiteren wurde das Jassproblem als ein Problem der Klassifizierung identifiziert. Für die Lösung der immer noch komplexen Teilprobleme werden die zwei Ansätze Supervised Learning und Reinforcement Learning verfolgt.

Risiko 5: Machine Learning Modell nicht optimal

Für die Entwicklung der Prototypen ist das Team mit zwei verschiedenen Ansätzen gestartet, mit der Option später den erfolgversprechenderen weiterzuverfolgen. Zum einen wird ein Netzwerk mit Hilfe der erhaltenen Daten durch Supervised Learning trainiert, um diese später eventuell durch Reinforcement Learning zu verbessern. Dieser Ansatz verspricht anfangs schnelle Ergebnisse, welche helfen werden eine geeignete Architektur des Netzwerkes zu evaluieren. Durch das Trainieren

mit Daten von menschlichen Spielern ist dieser Ansatz in seiner reinen Form ohne Reinforcement Learning durch die Qualität der Daten begrenzt. Daher wird parallel dazu der reine Reinforcement Learning Ansatz in einem weiteren Prototyp umgesetzt. Der Erfolg dieses Ansatzes ist von vielen Variablen abhängig und soll durch die Erfahrungen und Erkenntnisse des Supervised Learning bei der Definition der optimalen Variablen unterstützt werden.

Risiko 6: Ausfall der Infrastruktur (Laptop, Testserver, DGX Server)

Der gesamte Code sowie die Dokumentation befindet sich auf GitHub und ist somit vor Ausfällen der persönlichen Infrastruktur geschützt. Beim Testserver von HSR musste bereits in der frühen Phase des Projektes mehr Speicherplatz beantragt werden, da die benötigten Docker Images den begrenzten verfügbaren Platz schnell belegten. Grössere Installationen und Betriebsinformationen für die Testumgebung wurden dokumentiert, damit bei einem Ausfall eine schnelle Wiederherstellung gewährleistet ist.
Die ersten Testläufe auf dem DGX Server des ICOM waren erfolgreich. Für den schnellen DGX Server gibt es keine ebenbürtige Alternative an der HSR, allerdings können die Trainings der JassBots im Notfall auf dem Testserver der HSR oder auf den leistungsstarken Laptops der Studenten betrieben werden. Die Alternative eines Cloud Dienstes wurde nicht weiter verfolgt, da die Kosten dafür schnell ansteigen könnten und sich bei dieser Forschungsarbeit kein Kunde dahinter befindet, der für diese Unkosten aufkommen würde.

Risiko 7: Credit assignment problem

Bei der Auseinandersetzung mit der Fragestellung zum credit assignment problem wurde klar, dass diese Problematik im Forschungsgebiet der neuronalen Netzwerke in zwei unterschiedlichen Bereichen verwendet wird. Die Details zu diesen Erkenntnissen sind im Katipel 2.4.12 erläutert.

Risiko 8: Jassregeln werden missachtet

Da der Jassserver von Zühlke eine falsch gespielte Karte zurückweist, sollte das Reinforcement Learning Netzwerk durch den in einem solchen Fall erhaltenen negativen Reward versuchen eine derartige Entscheidung zu vermeiden. Eine programmierte Vorauswahl der spielbaren Karten, gemäss der Spielregeln, erscheint aufgrund des Ziels, einen vollständig auf einem neuronalen Netzwerk beruhenden JassBot zu entwickeln, nicht sinnvoll.

Risiko 9: Jassserver (Zühlke) hat Bugs

Beim Jassserver selbst wurden keine Bugs entdeckt.
Im zur Verfügung stehenden Python Gerüst für einen JassBot wurden jedoch kleinere Bugs entdeckt. Diese wurden gefixt und als entsprechende Pull Requests auf GitHub erfasst.

2.6 Prototyp

2.6.1 Konzept

Genauere Analyse des Jassproblems

Bei genauerer Betrachtung des Schieber Jasses fällt auf, dass ein Spieler jedes Mal eine Karte ablegt, wenn er an der Reihe ist. Dies geschiet unabhängig davon ob er eine passende/gute oder nur eine schlechte/ungeeignete Karte zu spielen hat. Dabei muss der Spieler stets aus allen aktuellen Handkarten eine auswählen.
Dies ist jedoch nicht die einzige Entscheidung, die in einem Spiel getroffen werden muss. Der Spieler muss mindestens jedes vierte Spiel auch einen Trumpf auswählen. Dabei sind ihm lediglich die Karten aus der eigenen Hand bekannt. Für die Trumpfentscheidung stehen dem Spieler grundsätzlich 7 Varianten zur Auswahl. Dies sind zum einen die 6 Trumpfvarianten, wie die 4 Farben, UNDEUFE und OBEABE, zum anderen noch die Option SCHIEBE, falls der Spieler Vorhand besitzt.
Mit dieser Analyse wurde der Entscheid gefällt, dass das Jassproblem der Spielvariante Schieber in zwei Teilproblemen betrachtet werden kann. Zum einen welche Karte abgelegt wird und zum anderen für welchen Trumpf sich ein Spieler entscheidet. Daher wird für jedes Teilproblem ein eigenes neuronales Netzwerk entworfen, parametrisiert, getestet und analysiert. Dabei werden Netzwerke, welche die zu legende Karte wählen nachfolgend als **Gamenetzwerk** bezeichnet. Die Netzwerke für die Trumpfentscheidung werden anschliessend als **Trumpfnetzwerk** betitelt.

Allgemeines Design

Die Ermittlung der optimalen Parameter aller möglichen Freiheitsgrade wurde auf mehrere Systeme verteilt. Die Definition der Funktion für die Rewardberechnung muss zwangsweise mittels Reinforcement Learning geschehen, da diese nur in dem Bereich Anwendung findet und für andere Systeme nicht relevant ist. Die Netzwerkgrösse, einzusetzenden Funktionen und deren Parameter sowie das Verhalten dieser Faktoren im Gesamtsystem mit Spieldaten kann auch auf einem anderen System ermittelt werden. Für dieses Teilproblem wird ein neuronales Netzwerk mit Supervised Learning eingesetzt, mit welchem schneller brauchbare Ergebnisse erzeugt werden können. Aufgrund dieser Ergebnisse werden anschliessend wieder Entscheidungen zur Änderung der Netzwerkgrösse, Funktionen und Parametern gefällt. Die daraus gewonnen Erkenntnisse können wiederum für das neuronale Netzwerk mit Reinforcement Learning adaptiert werden.
Aus den oben genannten Gründen wurde entschieden bei der Entwicklung des Prototyps zwei verschiedene Strategien parallel zu verfolgen. Zum einen wurde der Aufbau und die Konfiguration des neuronalen Netzwerkes mittels Supervised Learning ermittelt. Zum anderen wurde die Rewardberechnung für den Prototyp mit Hil-

fe eines Reinforcement Learning Systems erforscht. Aufgrund dieser Entscheidung wurde dieses Kapitel in die zwei Unterkapitel 2.6.2 Supervised Learning und 2.6.3 Reinforcement Learning gegliedert. In den einzelnen Unterkapiteln werden Experimente, deren Grund, Abläufe, Veränderungen, Ergebnisse und Entscheide aufgelistet. Die daraus resultierten Erkenntnisse beeinflussen wiederum andere Experimente sowie den finalen Prototyp, welcher unter dem Kapitel 2.8 Endergebnisse aufgeführt ist.

Für die Netzwerke wurde eine allgemeine Grundkonfiguration genommen, welche im Bereich der Klassifizierungsprobleme üblich ist. Diese Konfiguration wurde zuerst auf das Jassproblem angepasst, getestet und anschliessend wurden Parameter und Netzgrösse verändert. Diese Änderungen sind mit Supervised Learning getestet worden und gute Resultate wurden in das Reinforcement Learning Netzwerk übernommen.

Die entstehenden Netzwerke können im Grundgerüst von einem PythonBot getestet werden, welcher bereits aus der Vorstudie von Kapitel 2.5.5 bekannt ist. Mit diesem Softwaregerüst kann ein trainiertes Netzwerk auf dem Jassserver gegen andere Bots oder gegen Menschen spielen. Dieser Aufbau ist in der Abbildung 2.6.1 dargestellt und wird auch für das Training der Reinforcement Learning Netzwerke verwendet.

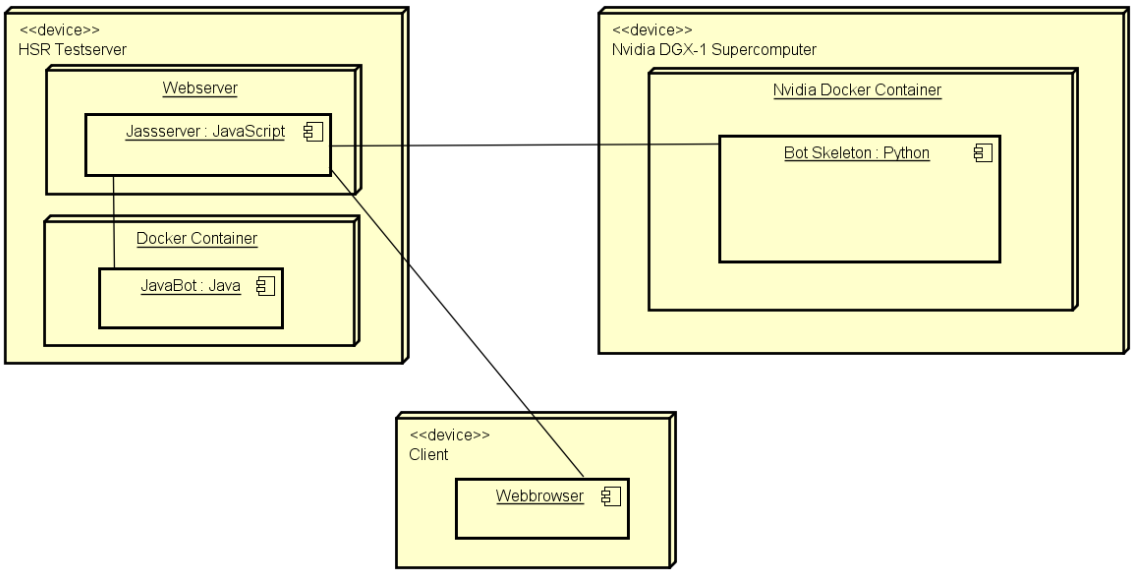


Abbildung 2.6.1: Deployment Diagramm der Umgebung des virtuellen Jassspieles

Design Gamenetzwerk

Das Design des Gamenetzwerk wurde über die Zeit der Forschungsarbeit stets weiterentwickelt. Hierbei hat sich neben der Anzahl der Hidden Layers und der Neuronenanzahl pro Layer auch der Input Layer verändert. Der Output Layer ist der einzige Layer im Gamenetzwerk, welcher während der gesamten Zeit konstant blieb.
Der Output Layer besitzt 36 Neuronen, da sich aus der Analyse des Jassproblems ergab, dass ein Spieler nur aus seinen Handkarten auswählen kann, welche in jeder Runde wieder ändern. Dabei besteht eine Auswahl von 36 möglichen Karten, weshalb dies 36 mögliche Entscheide bedeutet. Einem neuronalen Netzwerk wird eine konstante Grösse des Output Layer angegeben, daher besteht dieser aus 36 Neuronen.
Die Grösse des Input Layers hängt direkt von den Informationen ab, welche für ein neuronales Netzwerk benötigt werden. Grundsätzlich sollen dem Netzwerk mindestens Informationen über die aktuellen Hand- und Tischkarten sowie der aktuelle Trumpf übergeben werden. Hierbei können die Daten je nach Grösse und Organisation des Layers unterschiedlich aufbereitet sein. Folgend sind einige Grössen von Input Layer im Detail beschrieben und wie deren Organisation aussieht.

42 Neuronen am Input Layer

Die 42 Neuronen am Input Layer setzen sich aus den 36 Spielkarten und den 6 möglichen Varianten des zu wählenden Trumpfes zusammen. Die Neuronen am Output Layer widerspiegeln die 36 Jasskarten, von welchen das Netzwerk eine auswählen muss. Die Reihenfolge der symbolisierten Karten am Input und am Output ist dieselbe. Die 36 Spielkarten sind innerhalb der Farben jeweils **aufsteigend von 6 bis zum Ass** sortiert, wobei die Farben selbst die folgende Reihenfolge besitzen:

- **1.–9. Neuron:** Hearts
- **10.–18. Neuron:** Diamonds
- **19.–27. Neuron:** Clubs
- **28.–36. Neuron:** Spades

Gefolgt werden die Spielkarten von den 6 Trumpfvarianten, welche in der folgenden Reihenfolge dargestellt werden:

- **37. Neuron:** Hearts
- **38. Neuron:** Diamonds
- **39. Neuron:** Clubs
- **40. Neuron:** Spades
- **41. Neuron:** OBEABE
- **42. Neuron:** UNDEUFE

Dieses Design wurde mit der Überlegung gewählt, dass jede Karte einmal im ganzen Spiel existiert und über das ihm zugeteilte Neuron den Statuswert der jeweiligen Karte dem Netzwerk mitgeteilt werden kann. Dafür wurde mit folgender Statuskodierung gearbeitet:

- 0: Es ist keine Information über diese Karte bekannt, da diese im aktuellen Stich nicht vorkommt
- 1: Die Karte befindet sich in der eigenen Hand des JassBots
- 2: Diese Karte wurde als erste in diesem Stich gelegt
- 3: Diese Karte wurde als zweite in diesem Stich gelegt
- 4: Diese Karte wurde als dritte in diesem Stich gelegt

Mit diesem Design werden dem Netzwerk noch keine Informationen über die bereits gespielten Karten mitgeteilt. Diese Einschränkung wurde jedoch bewusst in Kauf genommen, weil zuerst erforscht werden sollte, wie viel das Netzwerk aus den einzelnen Stichen lernt und wie gut die getroffenen Entscheidungen für die aktuelle Situation und für das gesamte Spiel sind. Weiter wollte so die Frage geklärt werden, ob und wie schnell das Netzwerk so in der Lage ist zu lernen, welches Neuron welche Karte verkörpert, da jedes Neuron öfters verwendet wird.

150 Neuronen am Input Layer

Es wurde entschieden die Komplexität des Netzwerkes zu erhöhen und damit die Informationen dem Netzwerk in einer verständlicheren Form zu übergeben. Daher wird mit diesem Design eine binäre Kodierung eingeführt, welche für ein neuronales Netzwerk viel einfacher zu verstehen und zu erlernen ist, da somit jedes Neuron am Input genau zwei Zustände einnehmen kann. Bei einer 1 besitzt das Neuron eine Information für das Netzwerk und mit einer 0 trägt das Neuron keine relevante Information.
Der Input Layer mit seinen 150 Neuronen teilt sich in 4×36 Karten plus die 6 Trumpfvarianten auf. Die 36 Karten pro Set haben dabei genau dieselbe Anordnung wie das Kartenset beim Input Layer mit 42 Neuronen. Folgend ist die Zuordnung zu den Neuronen und die Bedeutung der vier Sets aufgelistet:

- **1.–36. Neuron:** Handkarten des JassBots erhalten eine 1
- **37.–72. Neuron:** Die Karte, welche im aktuellen Stich vom ersten Spieler gespielt wurde, trägt eine 1
- **73.–108. Neuron:** Die Karte, welche im aktuellen Stich vom zweiten Spieler gespielt wurde, trägt eine 1
- **109.–144. Neuron:** Die Karte, welche im aktuellen Stich vom dritten Spieler gespielt wurde, trägt eine 1

Gefolgt werden die 4 Spielkartensets wiederum von den 6 Trumpfvarianten in der folgenden Reihenfolge:

- **145. Neuron:** Hearts
- **146. Neuron:** Diamonds
- **147. Neuron:** Clubs
- **148. Neuron:** Spades
- **149. Neuron:** OBEABE
- **150. Neuron:** UNDEUFE

186 Neuronen am Input Layer

Bei diesem Designentwurf wird die Realität durch das Einführen des Kartenzählens noch mehr approximiert als bisher.
Mit 186 Neuronen am Input Layer wird ein für eine Maschine einfaches Kartenzählen realisiert, wodurch der Layer um ein Kartenset von 36 Karten ergänzt wird im Vergleich zum neuronalen Netzwerk mit 150 Neuronen am Input. Das Netzwerk muss erlernen, dass sobald eines dieser 36 Neuronen den Wert 1 besitzt, diese Karte bereits gespielt wurde und somit in diesem Stich nicht erneut auftauchen wird. Weiter soll mit dieser Idee untersucht werden, ob das Netzwerk in der Lage ist ein Muster zu erlernen. Da von Stich zu Stich immer mehr Karten gezählt werden und die Handkarten weniger werden, sollte das Netzwerk merken, dass es sich um ein Spiel mit zusammenhängenden Abläufen handelt. Der Input Layer baut sich wie folgt auf:

- **1.–36. Neuron:** Handkarten des JassBots erhalten eine 1
- **37.–72. Neuron:** Die Karte, welche im aktuellen Stich vom ersten Spieler gespielt wurde, trägt eine 1
- **73.–108. Neuron:** Die Karte, welche im aktuellen Stich vom zweiten Spieler gespielt wurde, trägt eine 1
- **109.–144. Neuron:** Die Karte, welche im aktuellen Stich vom dritten Spieler gespielt wurde, trägt eine 1
- **145.–180. Neuron:** Die Karten, welche im aktuellen Spiel bereits gespielt wurden, erhalten eine 1 (Kartenzähler)

Die letzten 6 Neuronen bilden wie beim Netzwerk mit 150 oder 42 Neuronen am Input die 6 Trumpfvarianten in der folgenden Reihenfolge:

- **181. Neuron:** Hearts
- **182. Neuron:** Diamonds
- **183. Neuron:** Clubs
- **184. Neuron:** Spades
- **185. Neuron:** OBEABE
- **186. Neuron:** UNDEUFE

Design Trumpfnetzwerk

Das Design des Trumpfnetzwerkes kann als simples neuronales Netzwerk bezeichnet werden, da für die Entscheidung eines Trumpfes bereits gute Resultate mit nur einem Hidden Layer erreicht wurden. Trotzdem entstanden auch von diesem Netzwerk unterschiedliche Varianten.

36-Trumpfnetzwerk

Die Grundlage für die Entscheidung eines Trumpfes sind die Handkarten. Daher müssen diese entsprechend am Input abgebildet werden. Somit entsteht für das erste Trumpfnetzwerk ein Input Layer mit 36 Neuronen für alle möglichen 36 Karten.

- **1.–9. Neuron:** Hearts
- **10.–18. Neuron:** Diamonds
- **19.–27. Neuron:** Clubs
- **28.–36. Neuron:** Spades

Der Hidden Layer wurde gleich gross gewählt wie der Input Layer. Die zwei Layers werden fully connected. Da das Problem, welches mit Hilfe des Trumpfnetzwerkes gelöst werden soll nicht sehr komplex ist, sollte diese Netzwerkgrösse reichen (Abbildung 2.6.2).
Der Output Layer besitzt 6 Neuronen, was die Anzahl der zur Auswahl stehenden Trumpfvarianten widerspiegelt.

- **1. Neuron:** Hearts
- **2. Neuron:** Diamonds
- **3. Neuron:** Clubs
- **4. Neuron:** Spades
- **5. Neuron:** OBEABE
- **6. Neuron:** UNDEUFE

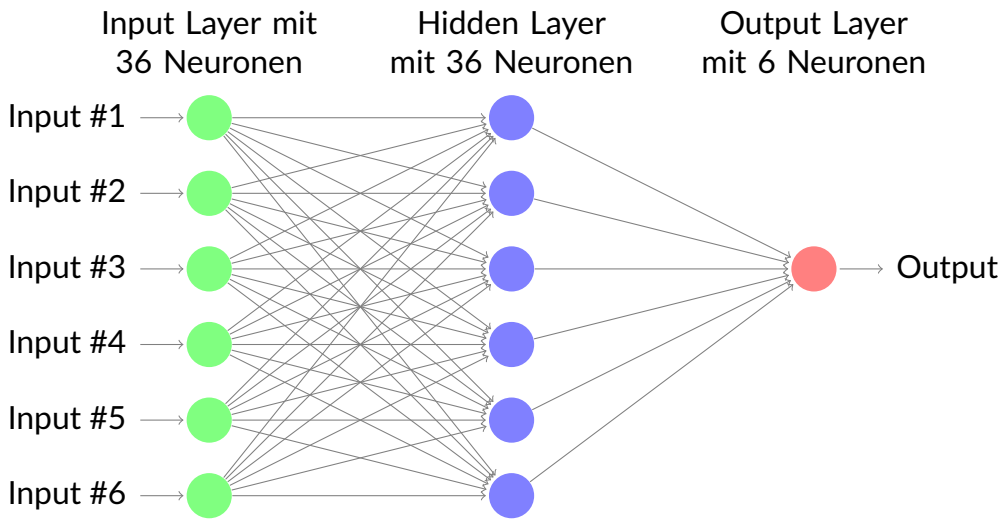


Abbildung 2.6.2: Darstellung des 36-Trumpfnetzwerkes im Massstab 1:6

37-Trumpfnetzwerk

Eine Weiterentwicklung des Trumpfnetzwerkes ist notwendig, damit dem neuronalen Netzwerk realitätsgetreue Informationen mitgeteilt werden können. Die Unterschiede im Design zum 36-Trumpfnetzwerk sind minimal und doch von hoher Relevanz zur Abbildung des Schieber Jass. Es handelt sich dabei um die Option SCHIEBE, von welcher sich auch der Spielname ableitet.

Das neue Design besitzt pro Layer ein Neuron mehr. Dies erlaubt am Output eine zusätzlich Entscheidung, ob anstelle einer Trumpfentscheidung geschoben werden soll.

- **1. Neuron:** Hearts
- **2. Neuron:** Diamonds
- **3. Neuron:** Clubs
- **4. Neuron:** Spades
- **5. Neuron:** OBEABE
- **6. Neuron:** UNDEUFE
- **7. Neuron:** SCHIEBE

Am Input ist diese Information ebenfalls wichtig, weil ein Spieler nicht erneut schieben darf, wenn sein Partner die Entscheidung des Trumpfes bereits zu ihm geschoben hat.

- **1.–9. Neuron:** Hearts
- **10.–18. Neuron:** Diamonds
- **19.–27. Neuron:** Clubs
- **28.–36. Neuron:** Spades
- **37. Neuron:** geschoben (1 geschoben; 0 nicht geschoben)

Der Hidden Layer wurde analog ebenfalls an die Grösse des Input Layers angepasst, da dieses Verhältnis in der Vorversion bereits gute Ergebnisse lieferte.

2.6.2 Supervised Learning

Das Projektteam erhielt von Swisslos aufgezeichnete Jasspartien, welche nach gewissen Kriterien gefiltert und für das Training vorbereitet werden mussten. Für die gezielte Datenaufbereitung wurde zuerst ein Parser entwickelt. Dieser musste je nach Netzwerk modifiziert werden, da sich die Netzwerke zum einen zwischen Game oder Trumpf und zum anderen in den zu verarbeitenden Daten unterscheiden.

Die Verteilung der vorhandenen Daten als Trainingsset und Testset, auch Validierungsdaten genannt, wurden für die zwei Netzwerktypen wie folgt festgelegt:

Netzwerk	Trainingsdaten	Testdaten
Game	90%	10%
Trumpf	70%	30%

Tabelle 2.6.1: Übersichtstabelle Trainings- und Testdaten Supervised Learning

Die Begründung dieser Verteilung beruht darauf, dass beim Game ein grösseres Netzwerk verwendet wird und somit mehr Trainingsdaten benötigt werden, um ein Underfitting zu vermeiden. Beim Trumpf hingegen ist das Netzwerk kleiner, wodurch ein grösserer Anteil der Daten als Testdaten benötigt werden, um ein Overfitting entdecken zu können.

Design

Da das Supervised Learning Netzwerk mit fremden Daten trainiert wurde, welche nicht vollständig mit der Codierung des Jassservers übereinstimmen, musste ein Parser entwickelt werden. Dieser liest die Daten aus den Logfiles der Swisslos, konvertiert diese für das Supervised Learning Netzwerk und übergibt dem Netzwerk die Daten fürs Training.
Da für die Konvertierung der Daten unter anderem Klassen benötigt wurden, welche im Python Grundgerüst bereits existierten, wurde das Package **training** in diesem Softwaregrundgerüst implementiert. Das Package enthält das gesamte Training für den Supervised Learning Bot.

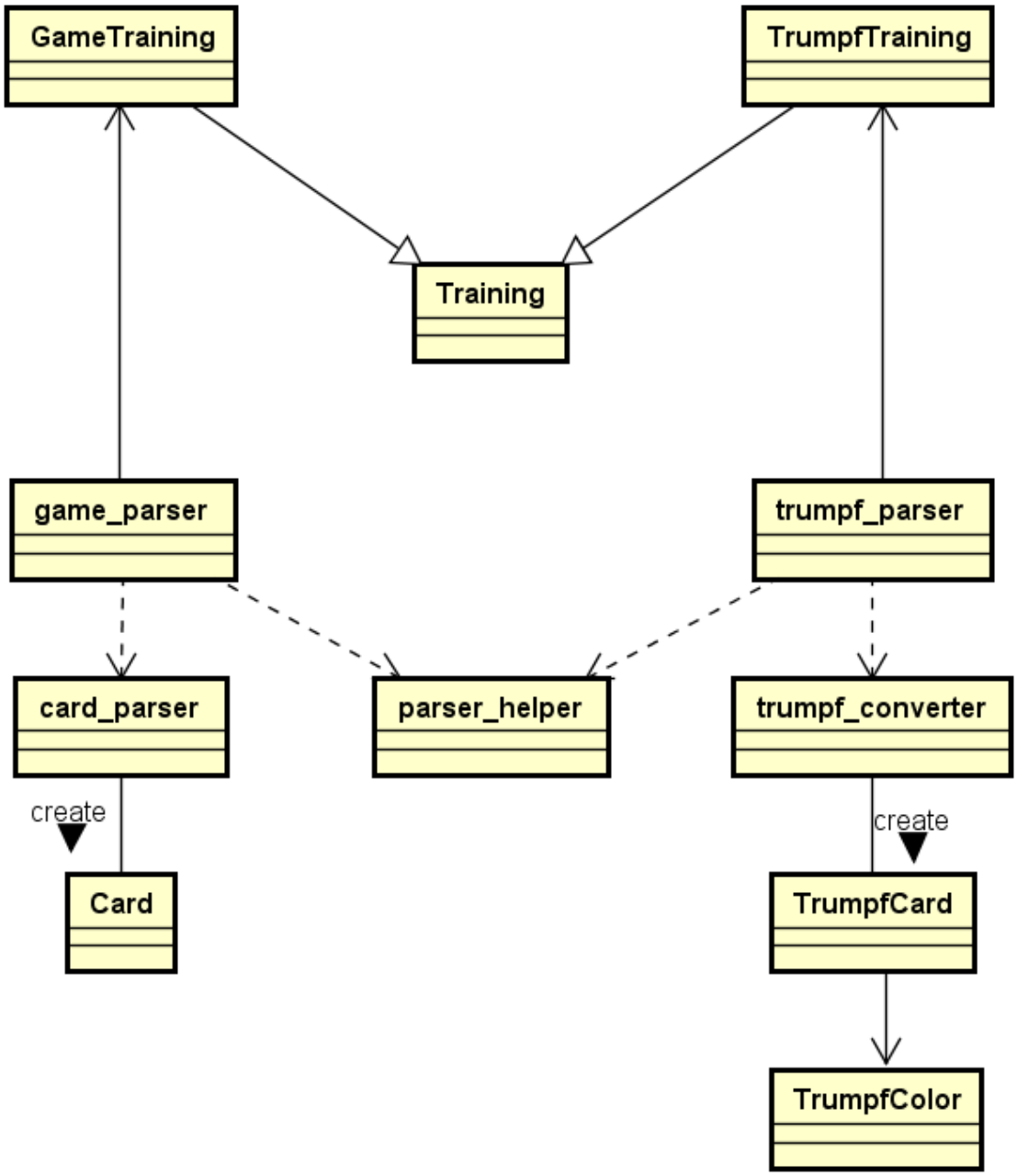


Abbildung 2.6.3: Klassendiagramm Supervised Learning

Im Klassendiagramm in Abbildung 2.6.3 ist je ein Training für das Trumpfnetzwerk und eines für das Gamenetzwerk dargestellt. Jedes Netzwerk benötigt einen eigenen Parser, was den Vorteil hat, dass auch jedes Netzwerk einzeln trainiert werden kann. Dies ist hilfreich, weil das Trumpfnetzwerk auch innerhalb von wenigen Minuten in der lokalen Entwicklungsumgebung trainiert werden kann, wohingegen das Gamenetzwerk selbst auf dem DGX Server noch einige Stunden benötigt.

Das Training inklusive Testing für das Gamenetzwerk wird über den game_parser gestartet und dasjenige für das Trumpfnetzwerk wird mittels trumpf_parser ausgelöst.

Tabelle der Experimente vom Supervised Learning

In den Supervised Learning Experimenten wurden diverse Netzwerke untersucht. Dabei hat jedes Netzwerk eine eigene Bezeichnung. Diese wurden in der Tabelle 2.6.2 aufgeführt.

Der Netzwerkname **SL_186_3_1Mio_leakyrelu** ist wie folgt aufgebaut:

SL: Einsatzbereich Supervised Learning
186: Anzahl Neuronen im Input Layer
3: Anzahl Hidden Layers
1Mio: Zusatzinformation zum Netzwerk, wie beispielsweise ein Netzwerk mit circa 1 Millionen Gewichten
leakyrelu: Bezeichnet den geänderten bzw. zu untersuchenden Konfigurationsparameter des Netzwerkes, wie beispielsweise Leaky ReLU

Experiment	Name	Domäne	Fokus
1	SL_42_1	Game	Statuscode
2	SL_150_1	Game	Einführung Realität
3	SL_150_2 SL_150_7 SL_150_9	Game	Netzwerkgrösse ermitteln
4	SL_186_2	Game	Kartenzähler
5	SL_186_5_Faktor1.5 SL_186_3_Faktor2 SL_186_3_Faktor3 SL_186_3_1Mio SL_186_5_2Mio SL_186_7_20Mio	Game	Netzwerkgrösse ermitteln
6	SL_186_3_1Mio_lr0.005 SL_186_3_1Mio_lr0.0001	Game	Learning Rate untersuchen
7	SL_186_3_1Mio_relu SL_186_3_1Mio_leakyrelu	Game	Aktivierungsfunktion untersuchen
8	SL_186_3_1Mio_sgd_cc SL_186_3_1Mio_adam_mse SL_186_3_1Mio_adam_cc	Game	Optimierungsfunktion und Lossfunktion ermitteln
9	SL_37_1	Trumpf	Datenaufbereitung
10	SL_37_1 SL_37_1_relu_adam_cc	Trumpf	Verwendung von Adam und Categorical crossentropy

Tabelle 2.6.2: Übersicht der Experimente von Supervised Learning

Experiment 1

Das erste Experiment ist ein einfaches neuronales Netzwerk und trägt den Namen **SL_42_1**. Die Architektur wird im folgenden kurz aufgelistet und beinhaltet gängige Startfunktionen:

- Input Layer: 42 Neuronen
- 1. Hidden Layer: 38 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Versuch 1

Das Ziel dieses Versuches ist es die ersten Messungen von einem neuronalen Netzwerk zu erstellen. Gleichzeitig soll auch die Funktionalität des Parsers erstmals getestet werden.
Das Netzwerk SL_42_1 wurde für den ersten Versuch in der originalen Konfiguration belassen und zwei Mal laufen gelassen. Dabei ergaben sich folgende Messungen.

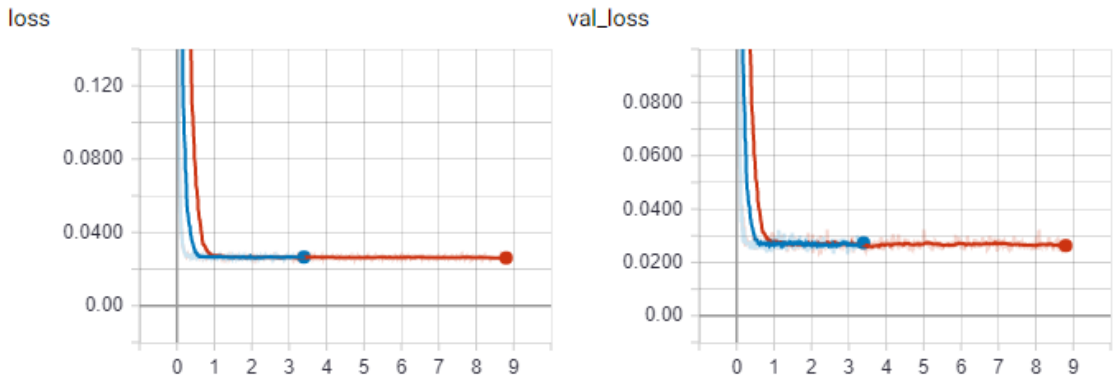


Abbildung 2.6.4: Experiment 1: Losswert im Training und bei der Validierung

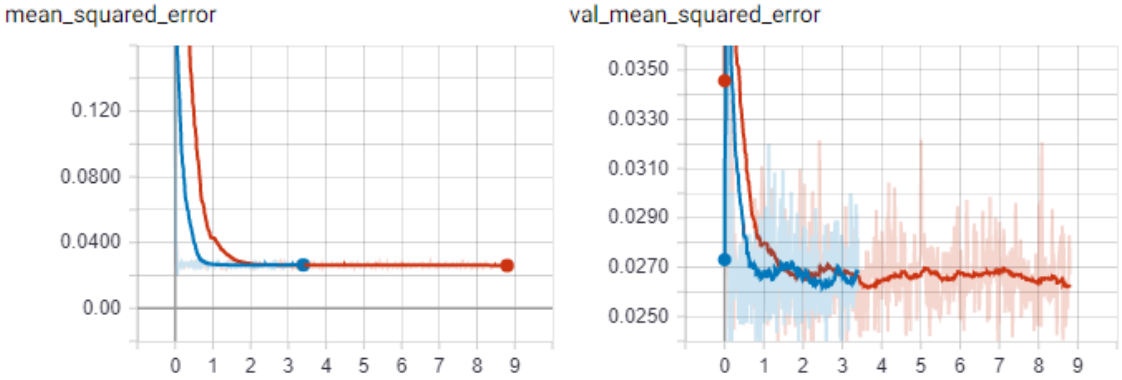


Abbildung 2.6.5: Experiment 1: Mean squared error im Training und bei der Validierung

Die Abbildungen 2.6.4 und 2.6.5 zeigen die zwei Kennzahlen Losswert und mean squared error sowohl für die Trainingsdaten auf der linken Seite als auch für die Testdaten rechts. Hierbei fällt auf, dass die Werte der beiden Testläufe ziemlich ähnlich sind, aber die Initialisierungswerte eines Netzwerkes selbst sind sehr unterschiedlich, was in der Abbildung 2.6.5 bei der Grafik des **val_mean_squared_error** anhand der Punkte am Startwert gut ersichtlich ist. Weitere Beobachtungen der Grafiken zeigen, dass die Netzwerke nach einer eher kurzen Zeit nicht mehr besser werden, da sich die Werte der Lossfunktion (Abb. 2.6.4) und des mean squared error (Abb. 2.6.5) nur noch minimal verändern. Dies lässt vermuten, dass das verwendete Netzwerk zu klein ist, da es sich um relativ hohe Werte handelt. Ein weiterer Unterschied fällt in der Dauer auf, welche für die Trainings benötigt wurden. Beim roten Graphen wurde während dem Training neben den Messungen auch noch Netzwerkgraphen für das TensorBoard erstellt, die den Aufbau des Netzwerkes grafisch darstellen. Dies nimmt viel Zeit in Anspruch, was hier im Vergleich zum blauen Graphen ersichtlich ist.

Erkenntnisse

Die angewendete Codierung der Spielinformationen ist der Grund, dass der Input Layer so klein ist. Jedoch ist es mit dieser Codierung für das Netzwerk schwieriger die verschiedenen Informationen im Input Layer auseinander zu halten, da diese sich nur durch ihre Werte unterscheiden. Dies könnte das Netzwerk verwirren, da es die Wertedifferenz ebenfalls als Information deuten könnte, obwohl dem nicht so ist.
Daher wird der Input Layer im nächsten Schritt ausgebaut. Die aktuellen Metriken scheinen für die Abschätzung der Güte des Netzwerks geeignet zu sein. In späteren Experimenten sollten sicher noch andere Metriken überwacht werden. Die Funktionen und Parameter des neuronalen Netzwerkes werden erst später verändert,

nachdem die Grösse als optimal eingestuft werden kann.

Experiment 2

Im zweiten Experiment wird ein neuronales Netzwerk mit dem Namen **SL_150_1** verwendet. Die Architektur wird im folgenden kurz aufgelistet, unterscheidet sich vom Netzwerk aus Experiment 1 jedoch lediglich in der Anzahl der Neuronen im Input Layer und denjenigen im Hidden Layer:

- Input Layer: 150 Neuronen
- 1. Hidden Layer: variierende Anzahl Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Das Ziel dieses Experimentes ist es, unterschiedliche Netzgrössen zu untersuchen, weshalb die Anzahl Neuronen für den Hidden Layer variieren. Daraus entstehen auch die in diesem Experiment durchgeführten Versuche.

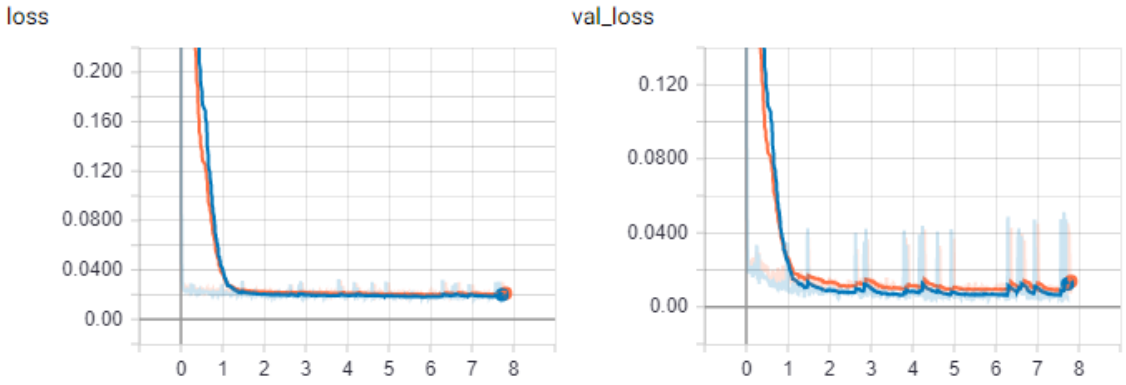


Abbildung 2.6.6: Experiment 2: Losswerte im Training und bei der Validierung

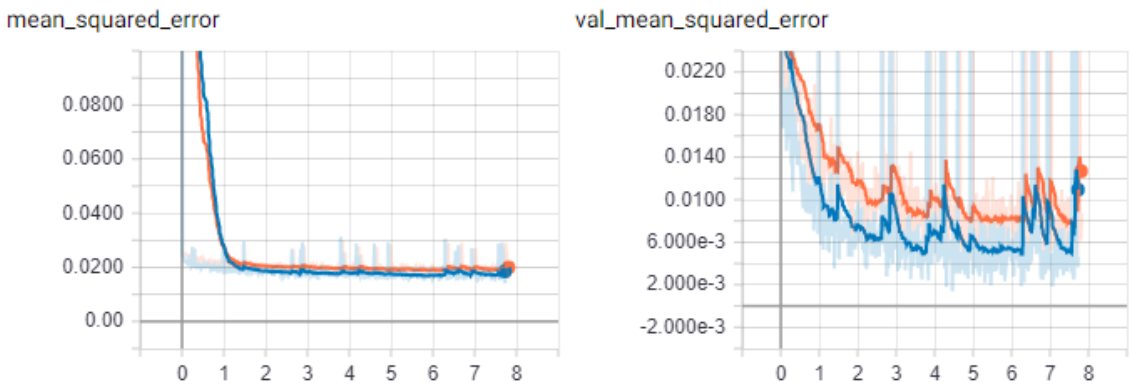


Abbildung 2.6.7: Experiment 2: Mean squared error im Training und bei der Validierung

Versuch 1

Mit dem ersten Versuch (oranger Graph in den Abbildungen 2.6.6 und 2.6.7) wird das Verhalten eines Netzwerkes untersucht, bei dem die Anzahl Neuronen stetig sinkt. Konkret bedeutet dies, dass sich nach dem Input Layer die Anzahl um zwei Drittel reduziert und anschliessend erneut um circa einen Drittel verringert wird.

Input Layer	1. Hidden Layer	Output Layer
150 Neuronen	50 Neuronen	36 Neuronen

Tabelle 2.6.3: Supervised Learning Netzwerk 150–50–36

Versuch 2

Beim zweiten Versuch (blauer Graph in den Abbildungen 2.6.6 und 2.6.7) wird das Verhalten eines Netzwerkes untersucht bei dem die Anzahl Neuronen zuerst ansteigt und anschliessend stark reduziert wird. Dementsprechend wird nach dem Input Layer die Anzahl der Neuronen im Hidden Layer um einen Drittel erhöht und danach am Output Layer auf circa einen Fünftel verkleinert.

Input Layer	1. Hidden Layer	Output Layer
150 Neuronen	200 Neuronen	36 Neuronen

Tabelle 2.6.4: Supervised Learning Netzwerk 150–200–36

Erkenntnisse

Die Abbildung 2.6.6 zeigt, dass die zwei unterschiedlichen Netzwerkgrössen beim Losswert im Training sowie bei der Validierung praktisch ein identisches Verhalten aufweisen.
Der Wert des mean squared error in der Abbildung 2.6.7 weist auch nur minimale Schwankungen auf, wobei bei der Validierung die Schwankung grösser ausfällt, als sie effektiv ist. Bei der genaueren Betrachtung der Skalierung fällt auf, dass sich beide Werte in einem sehr tiefen Bereich aufhalten. Eine solch minimale Differenz in diesem Bereich ist vernachlässigbar.
Damit Testergebnisse produziert werden können, welche sich stärker von denjenigen im aktuellen Experiment unterscheiden, sollten auch noch Netzwerke mit einer tieferen Struktur untersucht werden.

Experiment 3

Das dritte Experiment befasst sich mit neuronalen Netzwerken die mehrere Hidden Layers besitzen, daher tragen sie die Namen **SL_150_2**, **SL_150_7** und **SL_150_9**. Dabei wird die folgende Architektur verwendet:

- Input Layer: 150 Neuronen
- Anzahl Hidden Layer: variierende Anzahl Layers
- Anzahl Neuronen pro Hidden Layer: variierende Anzahl Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Das Ziel dieses Experimentes ist es die ersten neuronalen Netzwerke mit einer tieferen Struktur zu erforschen, weshalb auch die Anzahl Neuronen für die Hidden Layers variieren. Die folgenden drei Versuche befassen sich mit einer unterschiedlichen Anzahl von Hidden Layern, wie 2 Layern, 7 Layern und 9 Layern.

Versuch 1

Für den ersten Versuch wurde das Netzwerk aus Versuch 2 vom Experiment 2 als Grundlage verwendet und um einen zusätzlichen Hidden Layer ergänzt. Dadurch besitzt die neue Abstufung eine Halbierung der Neuronenanzahl des 1. Hidden Layers zum 2. Hidden Layer, welche vom letzten Hidden Layer wiederum auf ungefähr einen Drittel weiter abnimmt zum Output Layer.

Input Layer	1. Hidden Layer	2. Hidden Layer	Output Layer
150 Neuronen	200 Neuronen	100 Neuronen	36 Neuronen

Tabelle 2.6.5: Supervised Learning Netzwerk SL_150_2

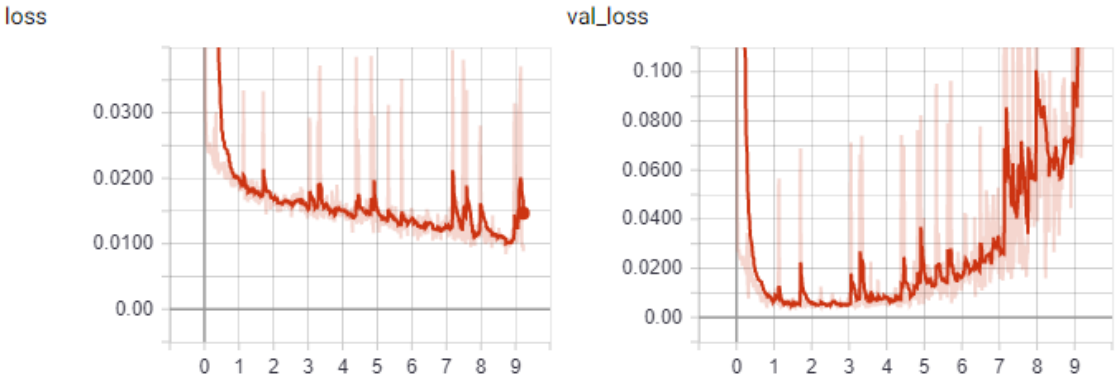


Abbildung 2.6.8: Overfitting beim Losswert im Training und bei der Validierung

Das Ergebnis von diesem Versuch ist insoweit interessant, da beim Losswert ein Overfitting beobachtet werden kann. Es wird oft versucht dieses Phänomen zu erreichen, da ein Netzwerk mit Overfitting eine gute Grundlage für die Verbesserung und Weiterentwicklung des Netzwerkes darstellt.

Versuch 2

Im zweiten Versuch wird ein Deep Learning Netzwerk untersucht. Dabei ist interessant zu sehen, wie sich ein Netzwerk verhält, dessen Hidden Layers gleichmässig vergrössert und anschliessend wieder im selben Verhältnis verkleinert werden.

Input Layer	**Hidden Layer**							**Output Layer**
	1	**2**	**3**	**4**	**5**	**6**	**7**	
150	220	340	500	340	220	150	80	36

Tabelle 2.6.6: Supervised Learning Netzwerk SL_150_7

In der Tabelle 2.6.6 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

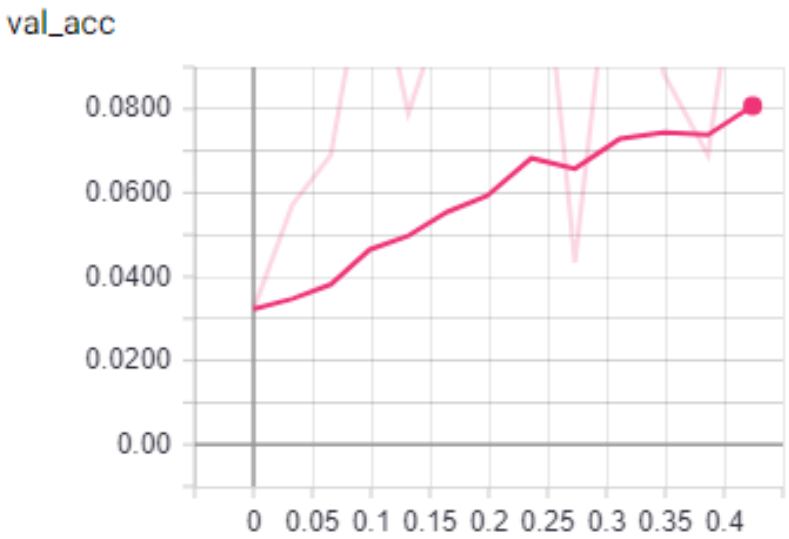


Abbildung 2.6.9: Accuracy Wert der Validierung vom SL_150_7

Mit dem Wert der accuracy kann ein absoluter Wert abgelesen werden, der bei diesem Netzwerk noch eher bescheiden ist.

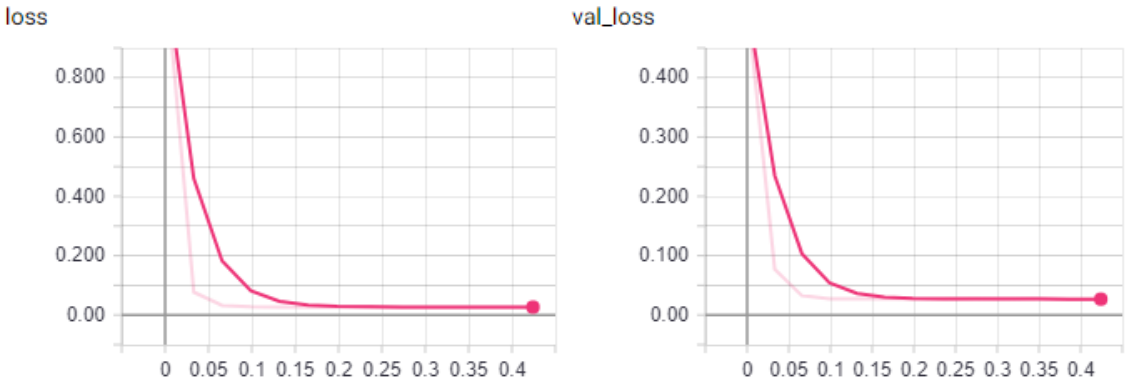


Abbildung 2.6.10: Losswert im Training und bei der Validierung vom Netzwerk SL_150_7

Der relative Losswert, welcher ein guter Indikator für die Güte eines Netzwerkes ist, scheint hier bereits relativ tief zu sein. Der Losswert pendelt sich jedoch ziemlich früh auf einem Wert ein und sinkt danach nicht mehr weiter ab.

Versuch 3

Im dritten Versuch wird das Netzwerk von Versuch 2 aus Experiment 3 modifiziert. Hierbei wurde der vierte Hidden Layer vergrössert, was die Einführung eines weiteren Hidden Layers verursachte, damit das Netzwerk immer noch gleichmässig vergrössert und verkleinert werden kann.

Input Layer	Hidden Layer									Output Layer
	1	2	3	4	5	6	7	8	9	
150	220	340	500	760	500	340	220	150	80	36

Tabelle 2.6.7: Supervised Learning Netzwerk SL_150_9

In der Tabelle 2.6.7 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

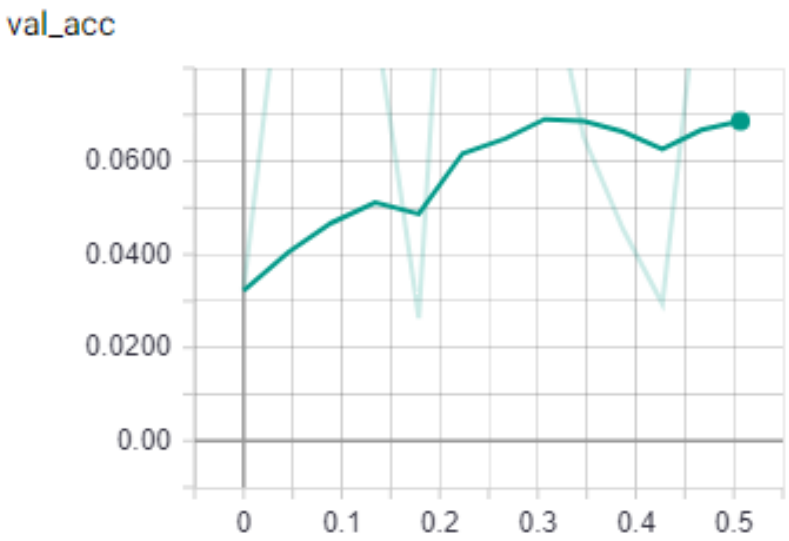


Abbildung 2.6.11: Accuracy Wert der Validierung vom Netzwerk SL_150_9

Der Wert der accuracy in Abbildung 2.6.11 besitzt einen noch schlechteren Wert als derjenige in Abbildung 2.6.9, wodurch ersichtlich wird, dass ein grösseres Netzwerk nicht zwangsweise eine Verbesserung mit sich bringt.

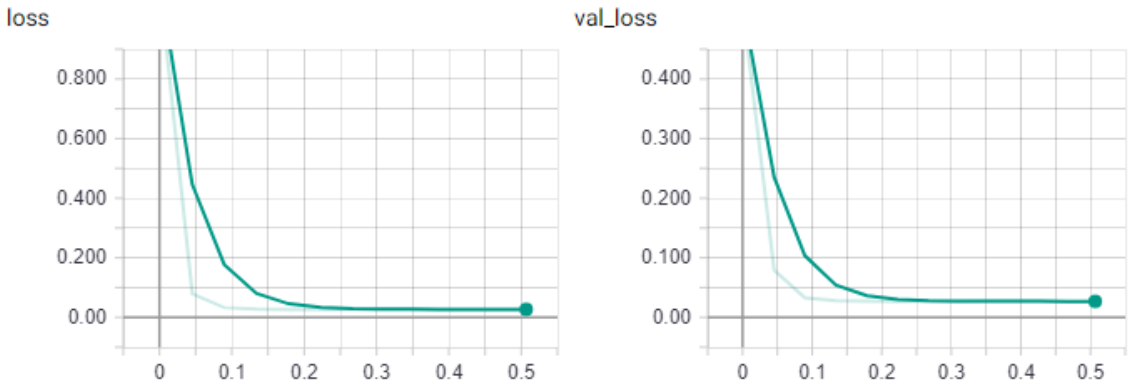


Abbildung 2.6.12: Losswert im Training und bei der Validierung vom Netzwerk SL_150_9

Das Verhalen des Losswertes in Abbildung 2.6.11 ist mit dem Wert aus der Abbildung 2.6.9 beinahe identisch. Dieser ist bereits ziemlich früh abgeflacht und das Netzwerk lernte nichts neues mehr.

Erkenntnisse

Einige Resultate der oben aufgeführten Versuche waren bereits sehr erfolgversprechend. Die verwendeten Netzwerke weisen jedoch noch grosse Unterschiede zur Realität auf.
Dass der Wert festgefahren ist, kann an der Grösse des Netzwerkes, aber auch an den verwendeten Paramter liegen. Mit den weiteren Experimenten soll versucht werden den Losswert weiter zu senken.
Anhand der früh abgeflachten Losswerte bei den Netzwerken mit tiefer Struktur ist erkennbar, dass für Netzwerke in dieser Grössenordnung zu wenige Daten vorhanden sind.

Experiment 4

Im vierten Experiment wird das neuronalen Netzwerken noch mehr der Realität angenähert, was einer Erweiterung des Input Layers entspricht, damit die Möglichkeit des Karten zählen entsteht. Dieses Netzwerk trägt somit die Bezeichnung **SL_186_2** und verwendet die folgende Architektur:

- Input Layer: 186 Neuronen
- 1. Hidden Layer: 420 Neuronen
- 2. Hidden Layer: 186 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Für die Einführung des Kartenzählers musste nicht nur das Netzwerk modifiziert sondern auch der Parser umgebaut werden.

Versuch 1

Dieser Versuch hat das Ziel, dass die Architektur mit 186 Neuronen im Input Layer eingeführt und das erste Mal getestet werden kann.

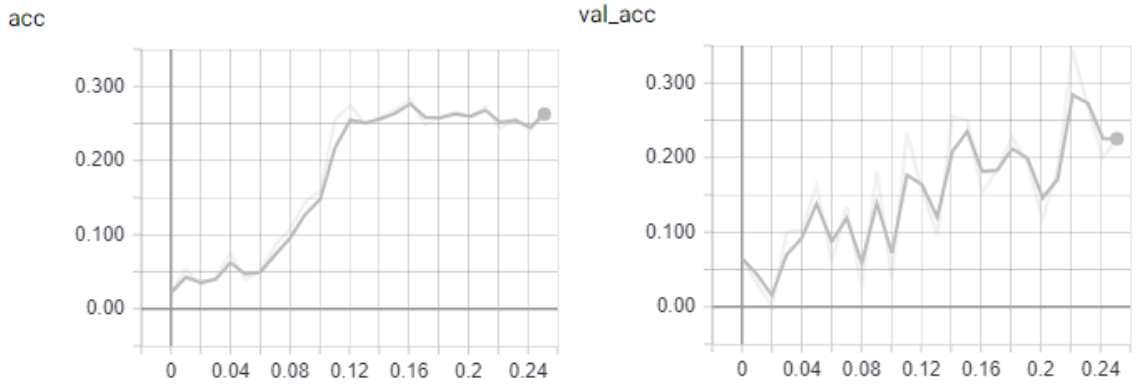


Abbildung 2.6.13: Experiment 4: Accuracy im Training und bei der Validierung

In der Abbildung 2.6.13 ist eine accuracy von ungefähr 0.25 ersichtlich. Im Vergleich zu den Messwerten im Versuch 2 und 3 aus Experiment 3 in den Abbildungen 2.6.9 und 2.6.11 sehen die absoluten Werte der accuracy bereits erfolgversprechender aus.

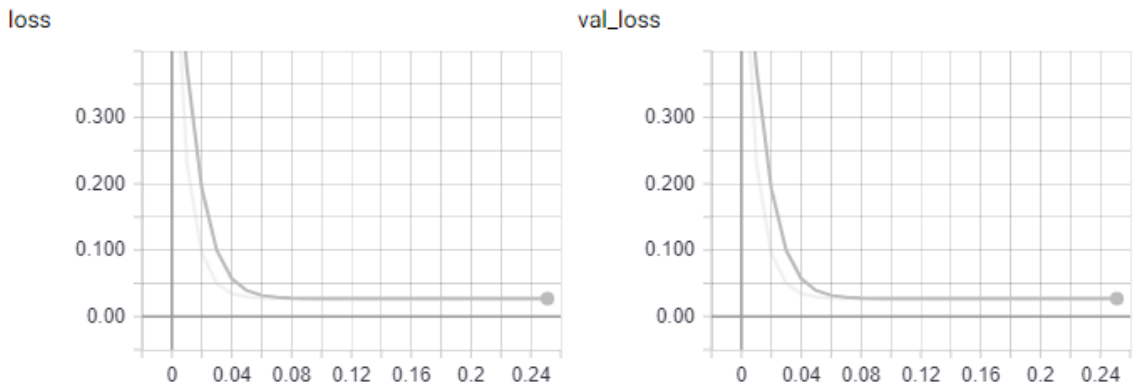


Abbildung 2.6.14: Experiment 4: Losswert im Training und bei der Validierung

Wird die Abbildung 2.6.14 mit den Werten in den Abbildungen 2.6.10 und 2.6.12 aus Experiment 3 verglichen, so fällt auf, dass diese Werte alle bereits nach kurzer Zeit auf dem Wert 0.027 stagnieren.

Erkenntnisse

Der Umbau des Parsers konnte mit wenig Aufwand bewerkstelligt werden, was das Einführen des Kartenzählens vereinfachte.
Mit Hilfe des grösseren Input Layers konnte eine verbesserte accuracy erreicht werden. Der Losswert stagnierte jedoch nach kurzer Zeit.

Experiment 5

Das fünfte Experiment befasst sich mit der Fragestellung, wie sich Deep Learning Netzwerke mit 186 Neuronen im Input Layer verhalten. Daher werden in den folgenden 6 Versuchen Netzwerke mit mehrere Hidden Layern, welche jeweils auch unterschiedliche Neuronenanzahl pro Layer besitzen, untersucht. Die Netzwerke tragen die Namen **SL_186_5_Faktor1.5**, **SL_186_3_Faktor2**, **SL_186_3_Faktor3**, **SL_186_3_1Mio**, **SL_186_5_2Mio** und **SL_186_7_20Mio**. Allen Netzwerken in diesem Experiment liegt dieselbe Grundkonfiguration zugrunde:

- Input Layer: 186 Neuronen
- Anzahl Hidden Layer: variierende Anzahl Layers
- Anzahl Neuronen pro Hidden Layer: variierende Anzahl Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Das Ziel dieses Experimentes ist, neuronale Netzwerke mit 186 Neuronen im Input Layer mit einer tieferen Struktur zu erforschen, weshalb auch die Anzahl Neuronen für die Hidden Layers variieren. Bei den ersten drei Versuchen wird dabei ein bestimmter Faktor für die Grösse der Netzarchitektur verwendet. Mit den anderen drei Versuchen wurden Netzwerkgrössen gewählt, die eine ungefähre Anzahl von Gewichten besitzen.

Versuch 1

Im ersten Versuch wird ein Deep Learning Netzwerk untersucht, dessen Architektur ungefähr mit einem Faktor von 1,5 vergrössert und wieder verkleinert wird.

Input Layer	**Hidden Layer**					**Output Layer**
	1	**2**	**3**	**4**	**5**	
186	280	180	120	80	53	36

Tabelle 2.6.8: Supervised Learning Netzwerk SL_186_5_Faktor1.5

In der Tabelle 2.6.8 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

Versuch 2

Beim zweiten Versuch wird die Netzwerkarchitektur etwa mit Faktor 2 vergrössert und verkleinert.

Input Layer	Hidden Layer			Output Layer
	1	2	3	
186	300	150	70	36

Tabelle 2.6.9: Supervised Learning Netzwerk SL_186_3_Faktor2

In der Tabelle 2.6.9 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

Versuch 3

Im dritten Versuch wird für die Vergrösserung und Verkleinerung der Netzarchitektur mit einem Faktor von etwa 3 gearbeitet.

Input Layer	Hidden Layer			Output Layer
	1	2	3	
186	560	180	60	36

Tabelle 2.6.10: Supervised Learning Netzwerk SL_186_3_Faktor3

In der Tabelle 2.6.10 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

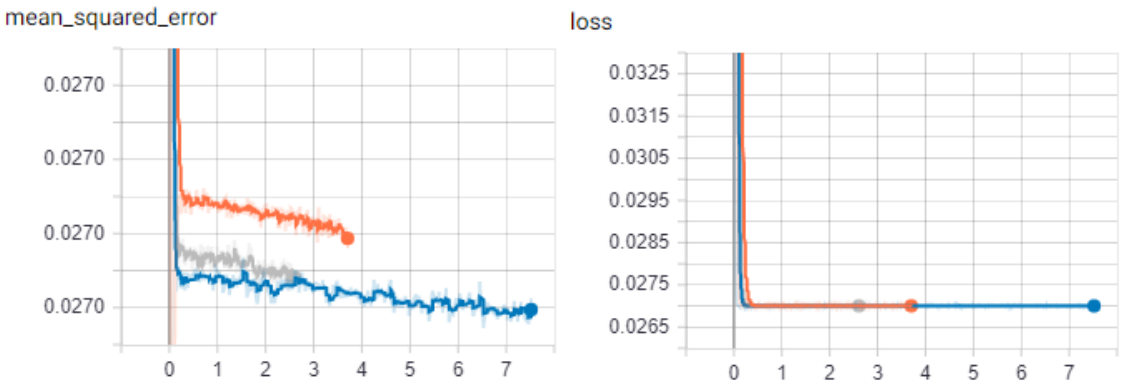


Abbildung 2.6.15: Losswerte und mean squared error der Netzwerke SL_186_5_Faktor1.5(orange), SL_186_3_Faktor2(blau) und SL_186_3_Faktor3(grau)

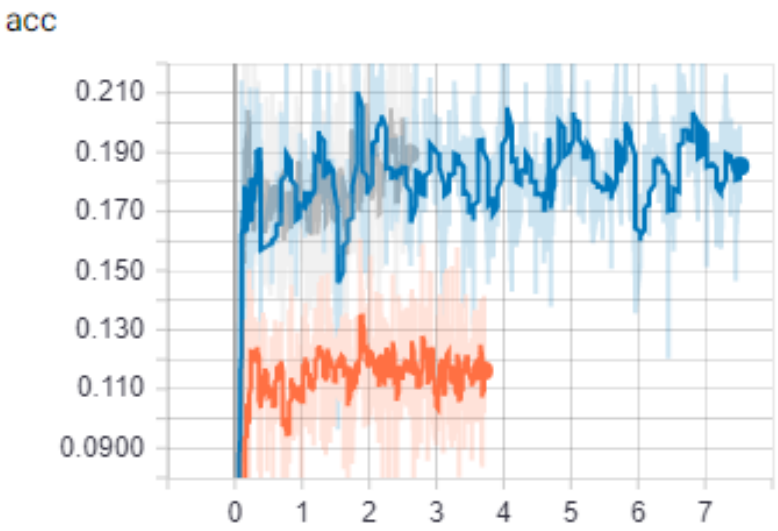


Abbildung 2.6.16: Accuracy der Netzwerke SL_186_5_Faktor1.5(orange), SL_186_3_Faktor2(blau) und SL_186_3_Faktor3(grau)

Versuch 4

Für den vierten Versuch wurde ein Netzwerk gewählt, welches ungefähr 1 Million Gewichte besitzt. Die dafür verwendete Berechnung ist unter 2.4 aufgeführt.

$$\begin{aligned} 186 \times 560 \ + \ 560 \times 1680 \ + \ 1680 \times 180 \ + \ 180 \times 36 = \\ 104\,160 + 940\,800 + 302\,400 + 6\,480 = 1\,353\,840 \end{aligned} \tag{2.4}$$

Input Layer	Hidden Layer			Output Layer
	1	2	3	
186	560	1680	180	36

Tabelle 2.6.11: Supervised Learning Netzwerk SL_186_3_1Mio

In der Tabelle 2.6.11 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

Versuch 5

Mit der gleichen Überlegung wie im vorherigen Versuch, wurde für den fünften Versuch ein Netzwerk mit etwa 2 Millionen Gewichten konstruiert.

Input Layer	Hidden Layer					Output Layer
	1	2	3	4	5	
186	560	1680	560	180	60	36

Tabelle 2.6.12: Supervised Learning Netzwerk SL_186_5_2Mio

In der Tabelle 2.6.12 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

Versuch 6

Im sechsten Versuch wird ein Deep Learning Netzwerk mit ungefähr 20 Millionen Gewichten eingesetzt.

Input Layer	Hidden Layer							Output Layer
	1	2	3	4	5	6	7	
186	560	1680	5040	1680	560	180	60	36

Tabelle 2.6.13: Supervised Learning Netzwerk SL_186_7_20Mio

In der Tabelle 2.6.13 steht die unterste Zeile für die Anzahl der verwendeten Neuronen pro Layer.

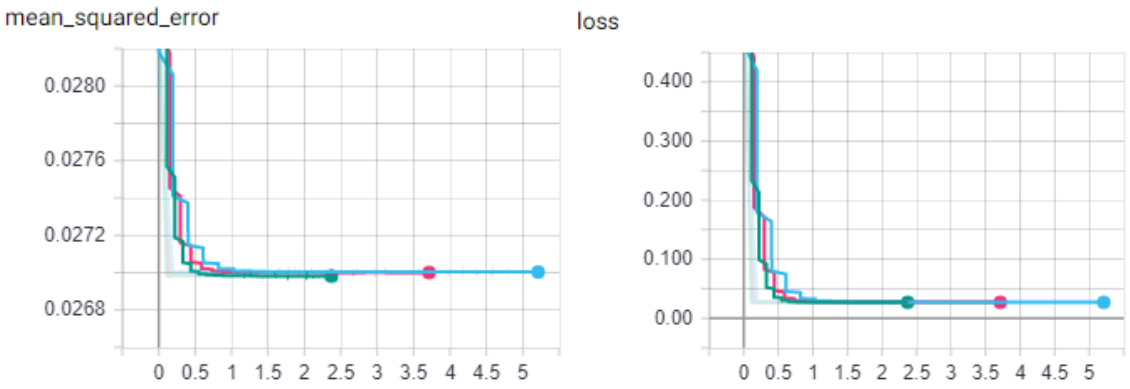


Abbildung 2.6.17: Losswerte und mean squared error der Netzwerke SL_186_3_1Mio(grün), SL_186_5_2Mio(pink) und SL_186_7_20Mio(hellblau)

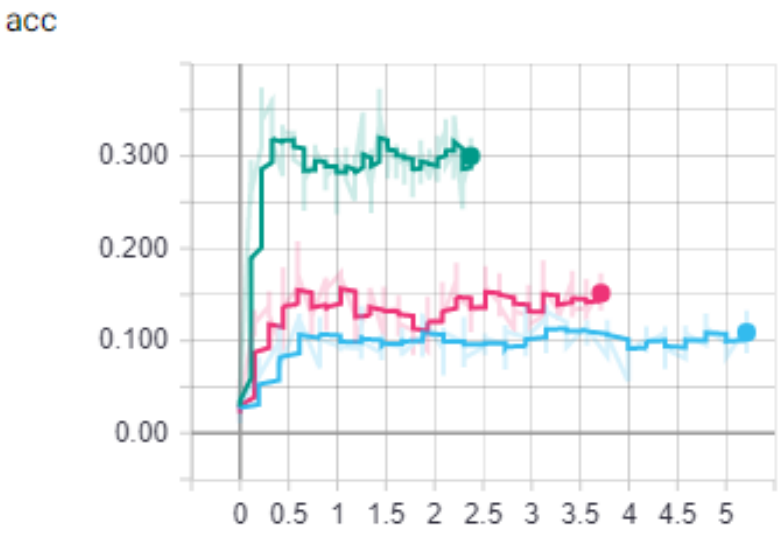


Abbildung 2.6.18: Accuracy der Netzwerke SL_186_3_1Mio(grün), SL_186_5_2Mio(pink) und SL_186_7_20Mio(hellblau)

Erkenntnisse

Die Versuche in diesem Experiment zeigen das Verhalten der unterschiedlichen Deep Learning Netzwerke. Dabei sind die ersten drei mit einer Anzahl von Gewichten unter 1 Million ausgestattet und die Netzwerke der Versuche 4, 5 und 6 mit einer Anzahl über 1 Million Gewichte.
In der Abbildung 2.6.15 fällt auf, dass alle drei Netzwerke die selben Werte 0.027 für den Loss und den mean squared error erreichen. Dieser Wert ist bereits von früheren Experimenten bekannt und taucht auch in der Abbildung 2.6.17 erneut auf. Auf den ersten Blick sieht dies nach einer Grenze aus. Allerdings unterschreitet

das Netzwerk SL_186_3_1Mio den Wert 0.027 ganz knapp. Dies weist darauf hin, dass es sich nicht um eine allgemeinen Grenze handelt, sondern eine Netzwerkgrösse mit ungefähr 1 Millionen Gewichten optimal zu sein scheint.
Bei der Betrachtung der Abbildungen 2.6.16 und 2.6.18 können unterschiedliche Werte ausgemacht werden. Hierbei schneiden die Netzwerke SL_186_5_Faktor1.5 und SL_186_7_20Mio mit einem Wert von etwa 0.11 am schlechtesten ab. Auch hier sticht das Netzwerk SL_186_3_1Mio mit einem Wert um 0.3 hervor. Dieser Wert ist noch besser als derjenige aus Experiment 4, was dafür spricht, dass mit der Netzwerkarchitektur vom SL_186_3_1Mio weitergearbeitet werden kann.

Experiment 6

Im sechsten Experiment wird mit dem besten Netzwerk SL_186_3_1Mio aus Experiment 5 die Learning Rate genauer unter die Lupe genommen. Es soll getestet werden, ob mit einer tieferen Learning Rate als der bisher verwendeten bessere Ergebnisse erzielt werden können. Dabei werden die Netzwerke **SL_186_3_1Mio_lr0.005** und **SL_186_3_1Mio_lr0.0001** mit der folgenden Konfiguration untersucht:

- Input Layer: 186 Neuronen
- 1. Hidden Layer: 560 Neuronen
- 2. Hidden Layer: 1680 Neuronen
- 3. Hidden Layer: 180 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: variiert

Vor diesem Experiment wurde die Datenaufbereitung durch den Parser nochmals überarbeitet. Dies war notwendig, da das Kartenzählen zwar eingeführt wurde, aber die Daten jedoch immer noch als einzelne Stiche aufbereitet wurden. Das Problem dabei war, dass dem Netzwerk ein einzelner Stich immer aus der Sicht aller Spieler mitgeteilt wurde. Der Stich wurde somit viermal aus unterschiedlichen Betrachtungswinkeln aufbereitet. Dadurch war es für das Netzwerk nicht möglich einen Zusammenhang zu den vorherigen oder nachkommenden Stichen herzustellen. Um diesen Nachteil zu beheben wurde der Parser so umgestaltet, dass nun alle Stiche eines Spieles aus der Sicht eines Spielers aufbereitet werden und so als Ganzes dem Netzwerk zum Trainieren und Testen übergeben wird.

Versuch 1

Der erste Versuch wird mit dem bisher verwendeten Wert für die Learning Rate von **0.005** durchgeführt.

Versuch 2

Im zweiten Versuch wurde die Learning Rate auf den Wert **0.0001** gesetzt.

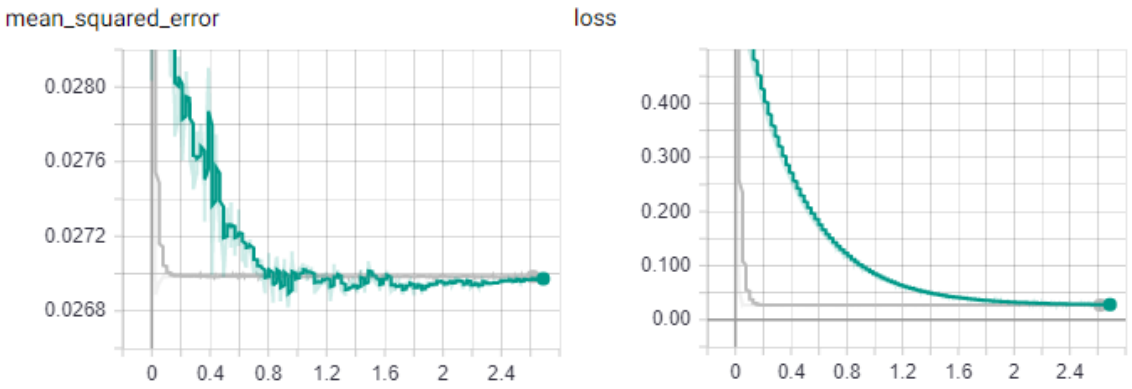


Abbildung 2.6.19: Losswert und mean squared error der Netzwerke SL_186_3_1Mio_lr0.005(grau) und SL_186_3_1Mio_lr0.0001(grün)

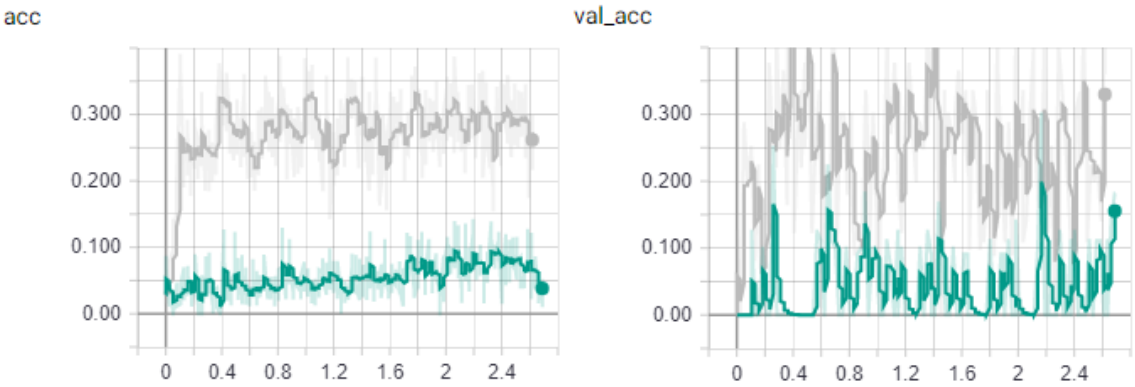


Abbildung 2.6.20: Accuracy der Netzwerke SL_186_3_1Mio_lr0.005(grau) und SL_186_3_1Mio_lr0.0001(grün)

Erkenntnisse

In der Abbildung 2.6.19 hat das Netzwerk mit der Learning Rate 0.0001 einen minimal besseren Losswert und mean squared error. Dies würde zwar für die kleinere Learning Rate sprechen, aber die höhere accuracy in der Abbildung 2.6.20 zeigt sehr deutlich auf, welche Learning Rate zu bevorzugen ist. Somit kommt auch für die weiteren Experimente der bisher verwendete Wert von 0.005 zum Einsatz.

Experiment 7

Mit dem siebten Experiment soll ermittelt werden, ob für das Netzwerk SL_186_3_1Mio noch eine bessere Aktivierungsfunktion gefunden werden kann. Die einzige sinnvolle Alternative zur aktuellen Aktivierungsfunktion ReLU ist leaky ReLU, weshalb diese zusätzlich untersucht wird. Folgende Grundkonfiguration kommt bei den Netzwerken **SL_186_3_1Mio_relu** und **SL_186_3_1Mio_leakyrelu** zum Einsatz:

- Input Layer: 186 Neuronen
- 1. Hidden Layer: 560 Neuronen
- 2. Hidden Layer: 1680 Neuronen
- 3. Hidden Layer: 180 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU oder Leaky ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Versuch 1

Mit dem ersten Versuch wird die Aktivierungsfunktion ReLU aufgezeichnet, damit die Messwerte in ein direktes Verhältnis mit den Messungen von Versuch 2 gestellt werden können.

Versuch 2

Im zweiten Versuch wird die Aktivierungsfunktion leaky ReLU getestet.

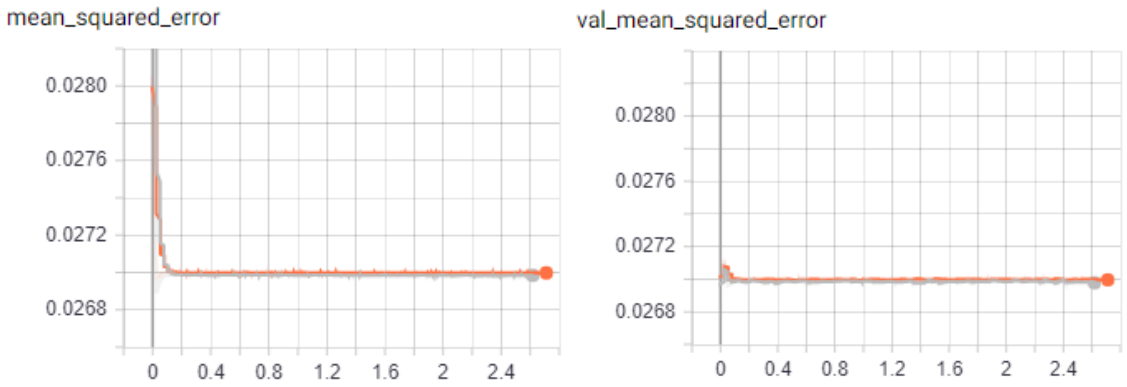


Abbildung 2.6.21: Mean squared error der Netzwerke SL_186_3_1Mio_relu(grau) und SL_186_3_1Mio_leakyrelu(orange)

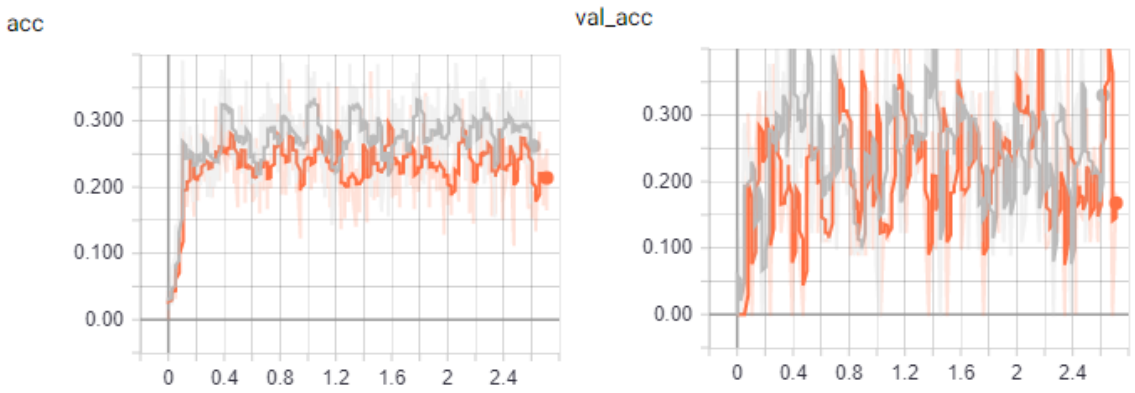


Abbildung 2.6.22: Accuracy der Netzwerke SL_186_3_1Mio_lr0.005(grau) und SL_186_3_1Mio_leakyrelu(orange)

Erkenntnisse

Mit Hilfe der Abbildung 2.6.21 wird ersichtlich, dass der mean squared error des Netzwerkes mit der Aktivierungsfunktion Leaky ReLU minimal schlechter ist als beim Netzwerk mit ReLU als Aktivierungsfunktion. Die Werte der beiden hier verglichenen Netzwerke im Bezug auf die accuracy sind im Training sowie in der Validierung von einem starken Rauschen geprägt. Jedoch kann erkannt werden, dass die Funktion ReLU auch hier leicht bessere Werte aufweist.
Des Weiteren wurde bei der erneuten Auseinandersetzung mit den Aktivierungsfunktionen klar, dass die Aktivierungsfunktion Softmax für den Output Layer sinnvoll wäre, da mit dieser Funktion eine Normalisierung der Werte am Output des Netzwerkes sichergestellt werden kann.

Experiment 8

Mit dem achten Experiment soll die Lossfunktion auf categorical crossentropy geändert werden, da diese Lossfunktion für Klassifizierungsprobleme optimiert ist. Da sich mit dieser Änderung auch Metriken ändern, wird zusätzlich eine Veränderung der Optimierungsfunktion untersucht, weil so ein besserer Vergleich statt finden kann.
Es handelt sich daher um drei Versuche mit den folgenden Netzwerke **SL_186_3_1Mio_sgd_cc**, **SL_186_3_1Mio_adam_mse**, **SL_186_3_1Mio_adam_cc** verglichen werden. Wobei cc für categorical crossentropy steht.

- Input Layer: 186 Neuronen
- 1. Hidden Layer: 560 Neuronen
- 2. Hidden Layer: 1680 Neuronen
- 3. Hidden Layer: 180 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Aktivierungsfunktion des Output Layer: Softmax
- Optimierungsfunktion: SGD oder Adam
- Lossfunktion: Mean squared error oder Categorical crossentropy
- Learning Rate: 0.005

Versuch 1

Im ersten Versuch wird die Kombination der Optimierungsfunktion SGD und der Lossfunktion categorical crossentropy im Netzwerk SL_186_3_1Mio_sgd_cc untersucht.

Versuch 2

Mit dem zweiten Versuch wird das Verhalten des Netzwerkes SL_186_3_1Mio_adam_mse, bei dem die bisherige Lossfunktion mean squared error mit der Optimierungsfunktion Adam kombiniert wird, untersucht.

Versuch 3

Beim dritten Versuch wird die Optimierungsfunktion und die Lossfunktion verändert und im Netzwerk SL_186_3_1Mio_adam_cc kombiniert getestet.

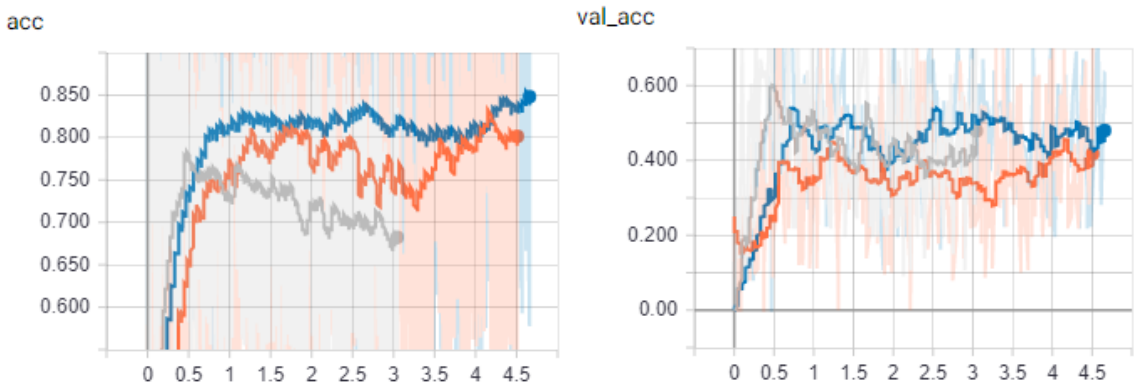


Abbildung 2.6.23: Accuracy der Netzwerke SL_186_3_1Mio_sgd_cc(grau), SL_186_3_1Mio_adam_mse(orange) und SL_186_3_1Mio_adam_cc(blau)

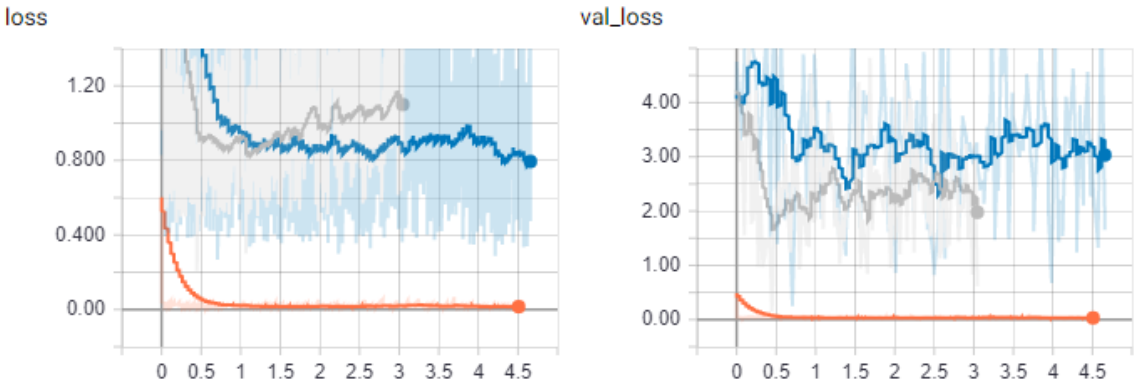


Abbildung 2.6.24: Accuracy der Netzwerke SL_186_3_1Mio_sgd_cc(grau), SL_186_3_1Mio_adam_mse(orange) und SL_186_3_1Mio_adam_cc(blau)

Erkenntnisse

In den Abbildungen 2.6.23 und 2.6.24 ist an den orangen und blauen Graphen erkennbar, dass die Optimierungsfunktion Adam mehr Zeit für die Optimierung hat als SGD, jedoch sind die Werte mit der Funktion Adam besser. Die accuracy des Netzwerkes SL_186_3_1Mio_adam_cc(blau) ist höher und konstanter als diejenigen der anderen beiden. Die Lossfunktion mean squared error und categorical crossentropy sind anhand des Losswertes eher schwer miteinander vergleichbar, wie dies aus der Abbildung 2.6.24 ersichtlich wird.
Die Kombination von Adam und categorical crossentropy scheint die Stabilste und Erfolgversprechendste zu sein. Somit ist die Netzwerkarchitektur und die Konfiguration des Versuches 3 aus dem Experiment 8 das Endergebnis von allen Supervised Learning Experimente vom Netzwerk Game.

Experiment 9

Im neunten Experiment wird das Netzwerk für die Auswahl des Trumpfes untersucht. Damit die Realität bereits von Beginn an korrekt abgebildet ist, wird gleich mit dem 37-Trumpfnetzwerk **SL_37_1** und der folgenden Konfiguration gestartet.

- Input Layer: 37 Neuronen
- 1. Hidden Layer: 37 Neuronen
- Output Layer: 7 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Aktivierungsfunktion des Output Layer: Softmax
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Das Ziel diese Experimentes war nicht die Architektur oder die Konfiguration zu überprüfen, sondern die Datenaufbereitung zu optimieren. Dem Projektteam stehen **139 633 Trumpfentscheidungen** in den erhaltenen Datensätzen zur Verfügung. Dabei wurde in **84 344** Fällen die Trumpfentscheidung dem Partner überlassen, was etwa **60%** entspricht. Auch gute Jassspieler schieben oft, jedoch ist bei diesen Spielen ein Anteil von 60% sehr hoch, was das Lernen verfälschen kann. Die erhaltenen Trumpfentscheidungen weisen ansonsten eine Gleichverteilung auf.

Um die Verfälschung der geschobenen Entscheidungen zu eliminieren, wurden für das Training einige Datensätze herausgefiltert. Das neuronale Netzwerk wird anschliessend mit einem vorbestimmten Testsample überprüft. Für die manuelle Überprüfung werden folgende Handkarten dem trainierten Netzwerk vorgelegt:

- Herz 6
- Herz 8
- Herz 9
- Herz Junge
- Herz Ass
- Ecke 9
- Ecke 10
- Schaufel 8
- Schaufel 10

Bei diesen Handkarten würde ein Mensch als erstes dazu tendieren, Herz als Trumpf zu wählen.

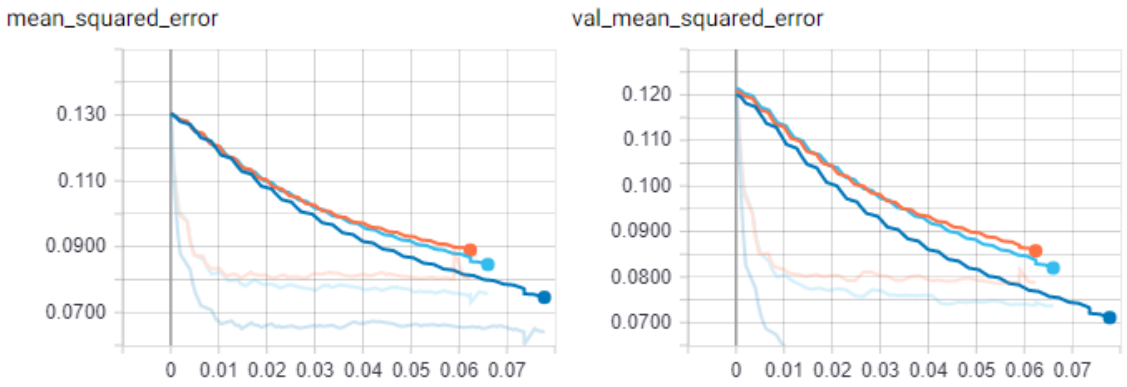


Abbildung 2.6.25: Vergleich der Trumpfnetzwerke im Bezug auf den mean squared error

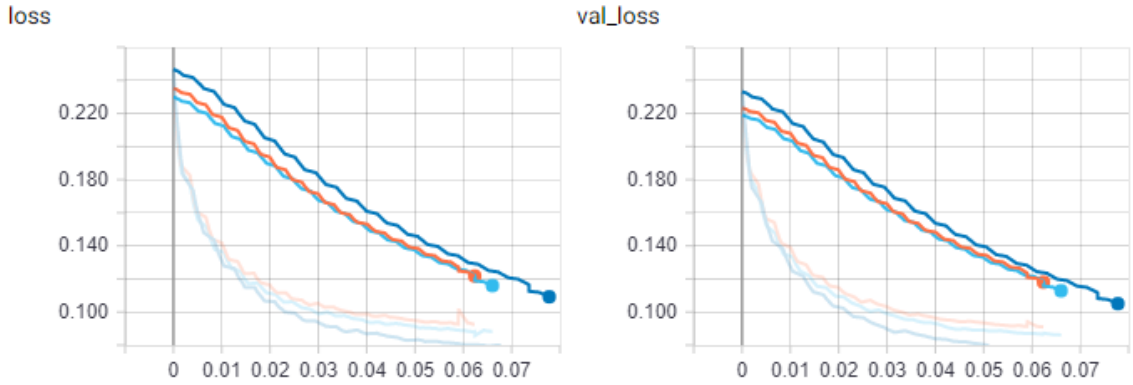


Abbildung 2.6.26: Losswerte der Trumpfnetzwerke im Vergleich

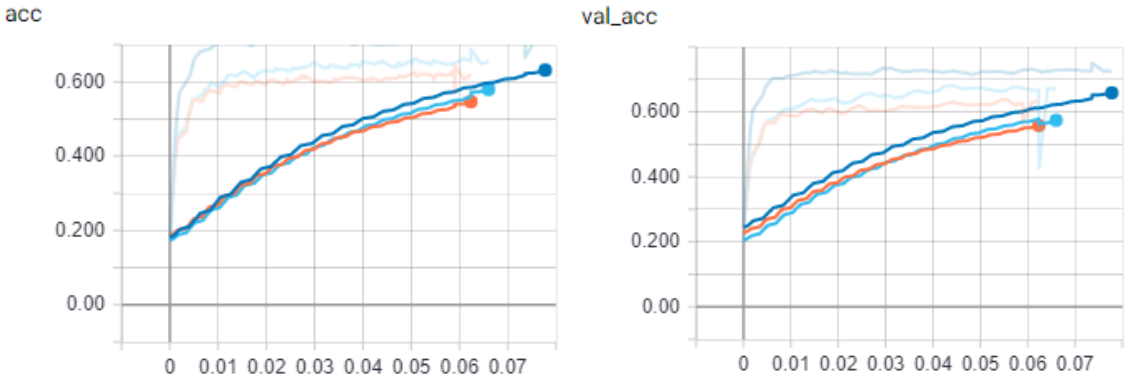


Abbildung 2.6.27: Accuracy der Trumpfnetzwerke im Vergleich

Versuch 1

Mit dem ersten Versuch wird aufgezeigt, wie sich das Netzwerk verhält, wenn alle Entscheidungen mit der Option SCHIEBE gelernt werden. In den Abbildungen 2.6.25, 2.6.26 und 2.6.27 ist bei der ungefilterten Dateneingabe (dunkelblau) jeweils der beste Wert erkennbar.
Der manuelle Test mit der Auswertung von Tabelle 2.6.14 zeigt hingegen, dass das Netzwerk eher schieben würde als selbst Herz als Trumpf zu wählen.

TRUMPF	**Voraussage**
Herz	0.3394807279109955
Ecke	0.05930786207318306
Kreuz	0.014802858233451843
Schaufel	0.014802858233451843
OBEABE	0.049792516976594925
UNDEUFE	0.10145732760429382
SCHIEBE	0.40142080187797546

Tabelle 2.6.14: Supervised Learning Trumpfnetzwerk ungefilterte Trumpfentscheidungen

Versuch 2

Mit dem zweiten Versuch wird nur noch die Hälfte aller Entscheidungen zu der Option SCHIEBE für das Netzwerk aufbereitet. Die Ergebnisse der Hälfte der SCHIEBE-Entscheidungen (orange) sind in den Abbildungen 2.6.25, 2.6.26 und 2.6.27 zu sehen. Dabei fällt auf, dass sie zwar in der Nähe der anderen Messungen liegen, jedoch die schlechtesten Ergebnisse abliefern, was nicht für diese Art der Filterung spricht. Die Voraussagen des Netzwerkes in der Tabelle 2.6.15 zeigen, dass Herz als Trumpf gewählt würde, was auch der menschlichen Entscheidung entspricht.

TRUMPF	**Voraussage**
Herz	0.49925366044044495
Ecke	0.07187937200069427
Kreuz	0.016980063170194626
Schaufel	0.051107872277498245
OBEABE	0.044772207736968994
UNDEUFE	0.0851583182811737
SCHIEBE	0.2308485060930252

Tabelle 2.6.15: Supervised Learning Trumpfnetzwerk mit der Hälfte der Trumpfentscheidungen

Versuch 3

Aus den Ergebnissen der Versuche 1 und 2 des aktuellen Experimentes lag es nahe, eine Art der Filterung zu wählen, die dazwischen liegt. Daher wird mit dem dritten Versuch ⅔ der Entscheidungen zum SCHIEBE für das neuronale Netzwerk aufbereitet. Die Ergebnisse der Aufzeichnungen von ⅔ SCHIEBE (hellblau) weist in den Abbildungen 2.6.25, 2.6.26 und 2.6.27 wie erwartet eine mittlere Güte auf.
Auch die Voraussagen des Netzwerkes für die Trumpfentscheidung gemäss der Tabelle 2.6.16 sehen zufriedenstellend aus.

TRUMPF	**Voraussage**
Herz	0.4332450330257416
Ecke	0.08428604900836945
Kreuz	0.025662725791335106
Schaufel	0.06633790582418442
OBEABE	0.0533493235707283
UNDEUFE	0.06523536890745163
SCHIEBE	0.27188366651535034

Tabelle 2.6.16: Supervised Learning Trumpfnetzwerk mit ⅔ der Trumpfentscheidungen

Erkenntnisse

Für die Datenaufbereitung fällt der Entscheid auf die Verwendung von ⅔ aller Entscheidungen für die Option SCHIEBE, was ein guter Mittelweg darstellt, die Spielpraxis relativ gut abbildet und das klassisch defensive Spiel unterstützt. Der Parser zur Datenaufbereitung wird dementsprechend angepasst und im weiteren mit dieser Filterung verwendet.

Experiment 10

Mit dem zehnte Experiment werden Erkenntnisse aus dem Experiment 8 des Gamenetzwerkes mit solchen aus Experiment 9 in Verbindung gebracht. Der Vergleich zwischen den Netzwerken **SL_37_1** und **SL_37_1_relu_adam_cc** soll dabei aufzeigen, wie sich die neue Konfiguration auswirkt.

Versuch 1

Aufbau und Konfiguration des Netzwerk **SL_37_1**:

- Input Layer: 37 Neuronen
- 1. Hidden Layer: 37 Neuronen
- Output Layer: 6 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Aktivierungsfunktion des Output Layer: Softmax
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

Versuch 2

Aufbau und Konfiguration des Netzwerk **SL_37_1_relu_adam_cc**:

- Input Layer: 37 Neuronen
- 1. Hidden Layer: 37 Neuronen
- Output Layer: 6 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Aktivierungsfunktion des Output Layer: Softmax
- Optimierungsfunktion: Adam
- Lossfunktion: Categorical crossentropy
- Learning Rate: 0.005

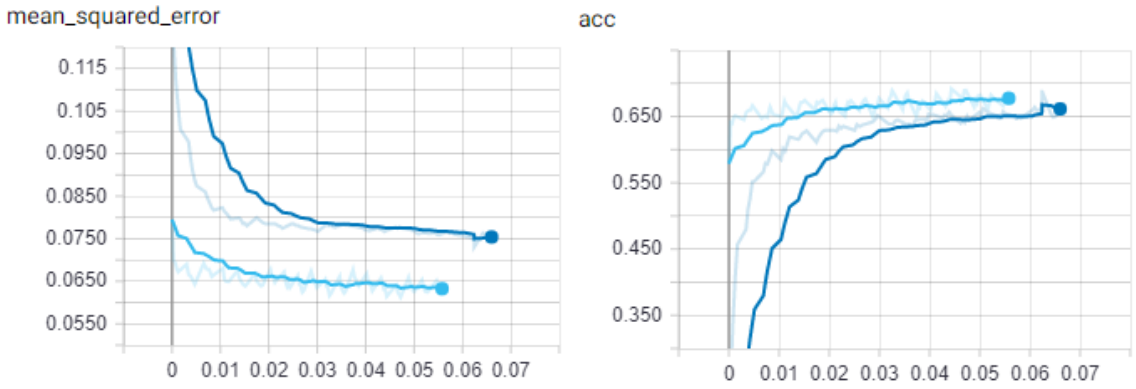


Abbildung 2.6.28: Accuracy und mean squared error der Netzwerke SL_37_1(dunkelblau) und SL_37_1_relu_adam_cc(hellblau) im Vergleich

Erkenntnisse

In der Abbildung 2.6.28 werden die Netzwerke SL_37_1 (dunkelblau) und SL_37_1_relu_adam_cc (hellblau) einander gegenübergestellt. Dabei kann ein markanter Unterschied zwischen den beiden Netzwerken ausgemacht werden. Die erkennbare Differenz beweist, dass die neue Konfiguration bessere Ergebnisse hervorbringt, weshalb das Netzwerk SL_37_1_relu_adam_cc die finale Version des Trumpfnetzwerkes darstellt.

2.6.3 Reinforcement Learning

Da es sich beim JassBot um ein Problem der Spieltheorie handelt, bei welchem unbekannte Faktoren existieren, sollte das System in der Lage sein, dynamisch auf neue Situationen reagieren zu können und daraus auch Neues zu lernen. Aufgrund dieser Tatsache fiel die Entscheidung auf die Lernmethode des Reinforcement Learning. Der dafür verwendeten Beurteilung liegt ein Bewertungsmuster zugrunde, welches stark vom abzubildenden Spiel abhängt.

Für die Experimente wurde der Jassserver so modifiziert, dass ein Turnier aus mehr als den standardmässigen fünf Spielen auf 2 500 Punkte bestand. Dies war nötig, um die Bots über längere Zeit trainieren zu können. Für den Reinforcement Learning Prototyp wurde auf dem Server die maximale Antwortzeit eines Clients von 500ms auf 15 Sekunden erhöht, da das Trainieren des Netzwerkes während des Spiels meistens länger dauerte als dieses initiale Timeout.

Im Gegensatz zum Supervised Learning sind beim Reinforcement Learning die Ergebnisse keine Wahrscheinlichkeitswerte für die einzelnen Optionen, sondern der erwartete Reward. Dies basiert auf der Trainingsart, bei welcher das Netzwerk mit den Rewardwerten für einzelne Aktionen trainiert wird.

Design

Das Reinforcement Learning Netzwerk wurde mit Hilfe eines in Python entwickelten Grundgerüsts erstellt. In diesem Softwaregrundgerüst wurden im Laufe der Arbeit einige Fehler entdeckt, welche dem Entwickler über pull requests mitgeteilt wurden.

Der Aufbau des Reinforcement Learning Bots ist im folgenden Klassendiagramm dargestellt.

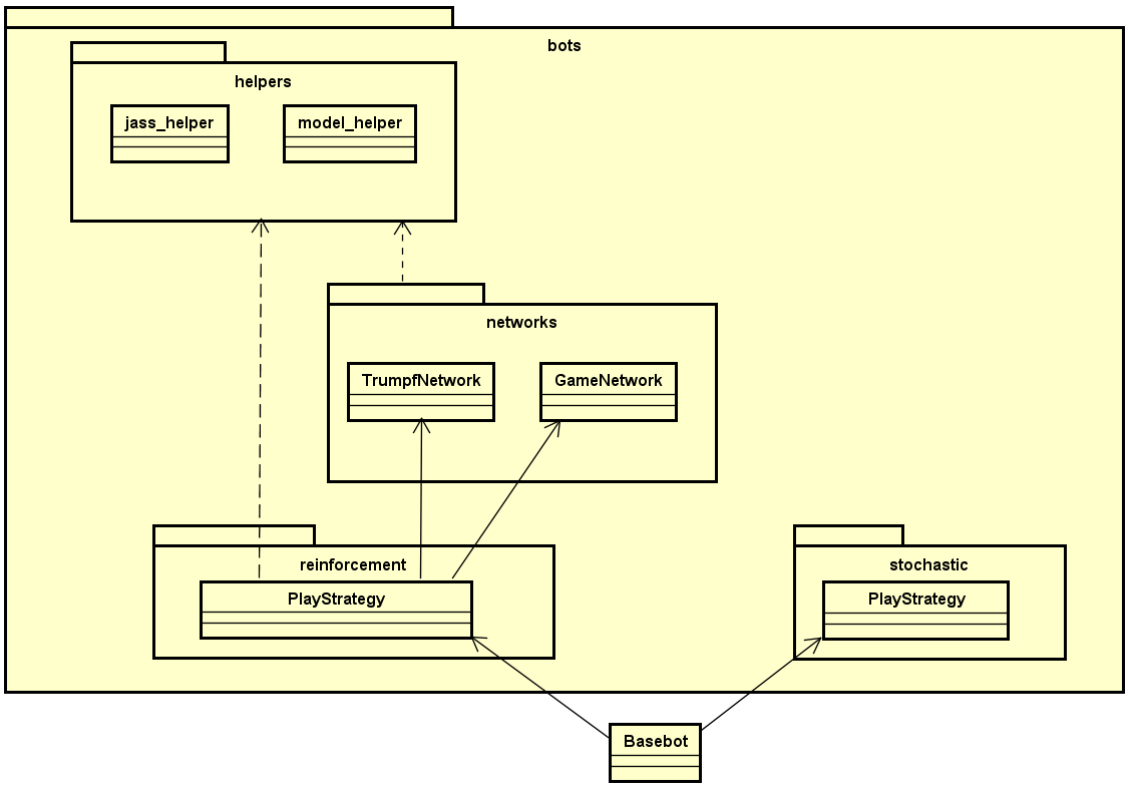


Abbildung 2.6.29: Klassendiagramm Reinforcement Learning

In der Abbildung 2.6.29 ist ersichtlich, dass der Bot aus der Klasse Basebot besteht, welche die gesamte Kommunikation mit dem Server verwaltet. Diese Klasse ruft dann weitere Funktionen zum Teil in der PlayStrategy auf. Die zu verwendende Strategie wird beim Start des Bots definiert. Der gesamte Reinforcement Learning Bot wurde in der PlayStrategy des Package reinforcement sowie in den Packages networks und helpers entwickelt.

Experimentübersicht

In den folgenden Tabellen ist eine kurze Übersicht über die durchgeführten Experimente aufgeführt. Die unterschiedlichen Netzwerke in den Experimenten sind nach ihrer Architektur benannt.
Mit der Spalte Domäne wird ausserdem unterschieden um welche Art von Netzwerk es sich handelt. Experimente zu neuronalen Netzwerken, die das Spielen von Karten lernen, werden nachstehend als Game bezeichnet. Die andere Kategorie sind Netzwerke, welche die Aufgabe haben den Trumpf zu bestimmen. Diese werden nachstehend als Trumpf bezeichnet.

Experiment	Name	Domäne	Fokus
1	RL_42_1	Game	neuronales Netzwerk für Spielentscheidungen
2	RL_150_2	Game und Trumpf	Netzwerkausbau
3	RL_186_5	Game und Trumpf	Rewardberechnung verbessern

Tabelle 2.6.17: Übersicht der Experimente von Reinforcement Learning

Die Ergebnisse der einzelnen Experimente sind in den jeweiligen Unterkapiteln festgehalten.

Experiment 1

Mit dem Wissen aus den Vorstudien wurde nun der erste Reinforcement Learning Protyp mit der folgenden Architektur erstellt:

- Input Layer: 42 Neuronen
- 1. Hidden Layer: 38 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.005

In diesem Experiment wurde nur ein Netzwerk für das Kartenspielen implementiert. Ein Trumpfnetzwerk wurde erst später entwickelt, nachdem erste Erfahrungen gesammelt worden sind. Das Ziel war, ein erstes neuronales Netzwerk umzusetzen, das einen Input vom Jasserver verarbeiten und eine Aktion ausführen kann. Auch das Training wurde erst in einem zweiten Schritt implementiert.

Für dieses Netzwerk mussten die Informationen, wie die Hand- und Tischkarten, in einen Tensor konvertiert werden, der dann dem Netzwerk übergeben werden konnte.

Netzwerk definieren

Keras bietet zwei Arten, wie Netzwerke definiert werden können. Die in diesem Experiment verwendete Methode wird sequential model[12] genannt. Dabei wird das Netzwerk vom Input Layer bis zum Output Layer aufgebaut. Die verschiedenen Layer werden nacheinander, mit Angaben zur Verbindungsart zwischen den Layern (in dieser Arbeit immer **Dense**, also fully connected), dem Netzwerk hinzugefügt und Keras verbindet diese im Hintergrund korrekt miteinander.

```
q_model = Sequential()
q_model.add(Dense(38, input_shape=(42,), kernel_initializer=
 'uniform'))
q_model.add(keras.layers.normalization.BatchNormalization())
q_model.add(Activation("relu"))
q_model.add(Dense(36, kernel_regularizer=l2(0.01)))
sgd = SGD(lr=0.005)
q_model.compile(loss='mean_squared_error', optimizer=sgd,
 metrics=['mean_squared_error'])
```

Listing 2.4: Netzwerk definieren

Vorhersage

Um das Netzwerk zu nutzen, stellt Keras die **predict()** Funktion zur Verfügung. Dieser Funktion werden die Eingabewerte, bestehend aus den Informationen über den Trumpf, die Hand- und Tischkarten, übergeben. Das Netzwerk gibt dann für jede Karte einen erwartete Reward zurück. Daraus wird die Handkarte mit dem höchsten erwarteten Reward gesucht und anschliessend gespielt.

```
q = self.q_model.predict(i)

card_to_play = hand_cards[0]

card_q = None
for c in hand_cards:
  if card_q is None or card_q < q[0, c.id]:
    card_to_play = c
    card_q = q[0, c.id]
```

Listing 2.5: Netzwerk auswerten

Nach jedem Stich werden die Eingabewerte und die gespielte Karte zusammen mit den erzielten Punkten gespeichert und später für das Training des Netzwerks verwendet.

```
minibatch = random.sample(self.memory, self.batch_size)
for state, action, reward, next_state, done in minibatch:
  target = reward
  if not done:
    target = (reward + self.gamma *
          np.amax(self.q_model.predict(next_state)[0]))
  target_f = self.q_model.predict(state)
  target_f[0][action] = target
  history = self.q_model.fit(state, target_f, epochs=1,
 verbose=0)
```

Listing 2.6: Netzwerk trainieren

Beim Training werden zufällig eine bestimmte Anzahl gespeicherter Stiche ausgewählt. Für jeden Stich wird anschliessend eine Reward berechnet. Dieser Reward setzt sich aus den in diesem Stich gemachten Punkten und dem Temporal Difference Wert (TD-Wert) zusammen. Der TD-Wert ist der im nächsten Zug maximal zu erwartende Reward mit einem Discountfaktor (Gamma) multipliziert, wodurch eine gewisse Voraussicht trainiert werden soll.

Ergebnis

Nach ersten Erfahrungen mit dieser Netzwerkarchitektur im Supervised Learning wurde relativ früh entschieden, dass es erfolgversprechender ist, die Eingabewerte binär zu codieren. Deshalb wurden mit dieser Architektur keine Ergebnisse aufgezeichnet.

Experiment 2

Nachdem im vorherigen Experiment die erste Version des Gamenetzwerks implementiert wurde, ist das Ziel dieses Experiments einerseits den Input Layer auf 150 Neuronen zu vergrössern und andererseits einige neue Features wie zum Beispiel ein Trumpfnetzwerk und das Logging mit TensorBoard hinzuzufügen. Zum Schluss sollen dann noch verschiedene Testläufe mit geänderten Parametern stattfinden.

Architektur **Gamenetzwerk**:

- Input Layer: 150 Neuronen
- 1. Hidden Layer: 100 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.001

Architektur **Trumpfnetzwerk**:

- Input Layer: 36 Neuronen
- 1. Hidden Layer: 36 Neuronen
- Output Layer: 6 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: SGD
- Lossfunktion: Mean squared error
- Learning Rate: 0.001

GPU RAM

Um mehrere Netzwerke parallel auf einer GPU auszuführen, musste ein Parameter in TensorFlow gesetzt werden. Standardmässig reserviert TensorFlow beim Start so viel GPU RAM wie möglich. Dies verhindert, dass mehrere Instanzen gestartet werden können. Mit dem Parameter **per_process_gpu_memory_fraction** wird der RAM Verbrauch pro Prozess begrenzt. So können problemlos mehrere Netzwerke gleichzeitig laufen.

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
sess = tf.Session(config=config)
keras.set_session(sess)
```

Listing 2.7: Alloziertes Memory reduzieren

In späteren Versionen wurde dieser Parameter durch den Folgenden ersetzt.

```
config.gpu_options.allow_growth = True
```

Damit alloziert TensorFlow nur so viel RAM wie nötig und es muss nicht bereits im Voraus ein fester Wert angegeben werden.

Implementation Trumpfnetzwerk

Bei der Implementation des Trumpfnetzwerkes tauchte ein Problem auf. Sobald beide Netzwerke definiert waren und zusammen trainiert werden sollten, gab TensorFlow Fehler aus. Untersuchungen ergaben, dass es problemlos möglich ist, zwei Netzwerke parallel laufen zu lassen, jedoch nicht gleichzeitig zu trainieren.

Pro Prozess kann nur ein Netzwerk trainiert werden, weshalb versucht wurde die beiden Netzwerke, Gamenetzwerk und Trumpfnetzwerk, in einem Netzwerk zu vereinen. Es sollte aber weiterhin möglich sein beide einzeln verwenden zu können. Dies ist mit der Art, wie die Netzwerke bis anhin definiert wurden nicht möglich. Deshalb konnten die Netzwerke nicht wie bisher über das sequential model definiert werden, sondern mussten über die functional API[11] erstellt werden. Bei der functional API müssen die einzelnen Komponenten der Layer selbst definiert und anschliessend miteinander verknüpft werden.

Im untenstehenden Auszug wird zuerst das Trumpfnetzwerk und im zweiten Abschnitt das Gamenetzwerk definiert. Im letzten Abschnitt wird aus den beiden Netzwerken ein Drittes definiert. Im Anschluss werden alle drei Netzwerke kompiliert und können verwendet werden.

Nun werden das Trumpf- und Gamenetzwerk nur noch während dem Spiel einzeln aufgerufen, also für das Bestimmen eines Trumpfes und das Wählen einer zu spielenden Karte. Trainiert werden beide Netzwerke über das Aufrufen der **fit()** Funktion auf dem **combined_model**.

```
trumpf_input = Input(shape=(36,), name='trumpf_input')
trumpf_dense_1 = Dense(36,
  activation='relu',
  kernel_initializer='truncated_normal')(trumpf_input)
trumpf_batch_norm_1 = BatchNormalization()(trumpf_dense_1)
trumpf_dense_out = Dense(6,
  kernel_regularizer=l2(0.01),
  name='trumpf_output')(trumpf_batch_norm_1)

self.trumpf_model = Model(
  inputs=trumpf_input, outputs=trumpf_dense_out)
```

```
game_imput = Input(shape=(150,), name='game_input')
game_dense_1 = Dense(100,
  activation='relu',
  kernel_initializer='truncated_normal')(game_imput)
game_batch_norm_1 = BatchNormalization()(game_dense_1)
game_dense_out = Dense(36,
  kernel_regularizer=l2(0.01),
  name='game_output')(game_batch_norm_1)

self.game_model = Model(
  inputs=game_imput, outputs=game_dense_out)


self.combined_model = Model(
  inputs=[trumpf_input, game_imput],
  outputs=[trumpf_dense_out, game_dense_out])

sgd = SGD(lr=0.001)
self.trumpf_model.compile(optimizer=sgd,
      loss='mean_squared_error',
      metrics=['mean_squared_error', 'accuracy'])

self.game_model.compile(optimizer=sgd,
      loss='mean_squared_error',
      metrics=['mean_squared_error', 'accuracy'])

self.combined_model.compile(optimizer=sgd,
      loss='mean_squared_error',
      metrics=['mean_squared_error', 'accuracy'])
```

Listing 2.8: Kombiniertes Netzwerk

Probleme

Für das Gamenetzwerk mit Reinforcement Learning wurde in einigen Fällen der Wert **NaN** (Not a Number) in die Logdateien geschrieben, worauf das Netzwerk dann abstürzte. Eine genauere Analyse bestätigte, dass die Zeit bis zum Absturz mit der Learning Rate zusammenhing und Netzwerke mit einer kleineren Learning Rate später abstürzten als solche mit einer höheren.

Der Wert **NaN** kann die folgenden zwei Ursachen haben:

- Das Netzwerk gibt eine unendlich grosse Zahl aus, welche nicht mehr als eine Nummer verstanden wird.
- Es gibt innerhalb des Netzwerkes eine Division durch 0, was mathematisch verboten ist und somit einen Fehler erzeugt.

Da jedoch alle Netzwerke, unabhängig von der Learning Rate schliesslich abstürzten, wurde die Ursache auch an anderen Orten gesucht.

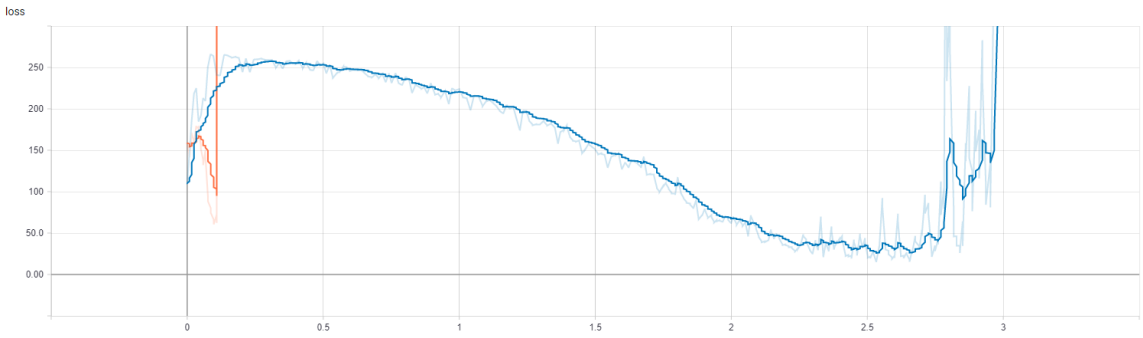


Abbildung 2.6.30: Losswerte mit zwei verschiedenen Learning Rates

Die Auswertung in Abbildung 2.6.30 der Losswerte zeigt, wie unterschiedlich die Trainingszeit sein kann. In diesem Beispiel wurden eine Learning Rate von 0.001 und 0.0001 verwendet.
Da der Losswert zum Schluss hin sehr stark ansteigt, wurde vermutet, dass dies mit dem relativ grossen Reward zusammenhängt und deshalb überkorrigiert wird. Zurzeit ist der Reward die im Stich erreichte Punktzahl, welche also relativ hoch ausfallen kann. Normalerweise wird mit Rewards im Bereich von -1 bis 1 gearbeitet. Aus diesem Grund wurde ab diesem Zeitpunk der Reward um den Faktor 100 verkleinert. Ein Blick auf den Losswert in der Abbildung 2.6.31 nach dieser Änderung zeigt, dass damit das Problem gelöst werden konnte. Allgemein sieht die Kurve des Losswertes nun viel besser aus.

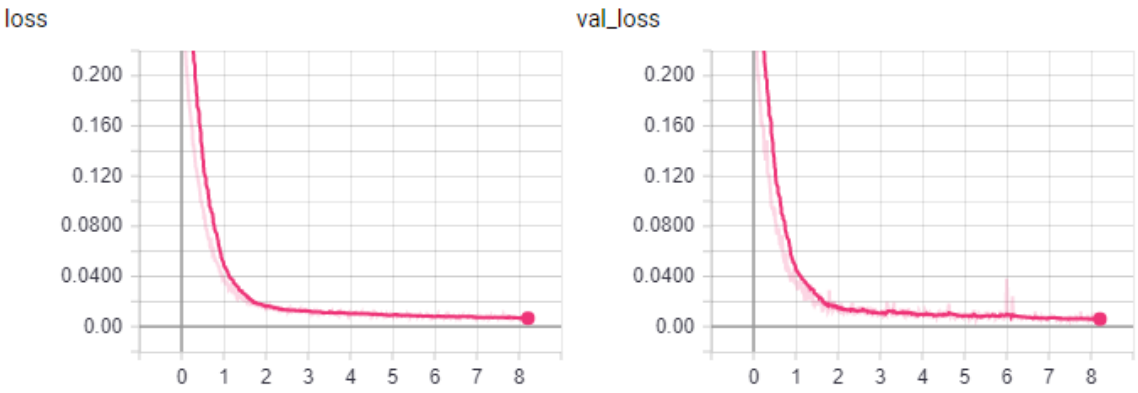


Abbildung 2.6.31: Losswert mit skaliertem Reward

Ergebnis

Somit ist nun ein erstes Reinforcement Learning Netzwerk bereit und kann trainiert werden. In einem nächsten Schritt soll nun die Realität noch genauer abgebildet werden, indem dem Netzwerk beigebracht wird, Karten zu zählen.

Experiment 3

Mit dem 186 Neuronen grossen Input Layer werden nun auch Informationen über die bereits gespielten Karten dem Netzwerk übergeben.

Architektur **Gamenetzwerk**:

- Input Layer: 186 Neuronen
- 1. Hidden Layer: 420 Neuronen
- 2. Hidden Layer: 950 Neuronen
- 3. Hidden Layer: 420 Neuronen
- 4. Hidden Layer: 186 Neuronen
- 5. Hidden Layer: 80 Neuronen
- Output Layer: 36 Neuronen
- Verbindung zwischen den Layern: fully connected
- Aktivierungsfunktion: ReLU
- Optimierungsfunktion: Adam
- Lossfunktion: Categorical crossentropy
- Learning Rate: 0.001

Neuer Trumpfreward

Bis anhin wurde der Reward für das Trumpfnetzwerk aus den Stichen berechnet, die in diesem Spiel gemacht wurden. Bei dieser Bewertungsmethode hängt der Reward jedoch sehr stark von der Spielqualität des Gamenetzwerkes ab. Somit ist vor allem zu Beginn des Trainings der Reward schlechter, da auch das Gamenetzwerk noch untrainiert ist und deshalb schlechter spielt. Dieser Zusammenhang verfälscht den Reward des Trumpfnetzwerkes, welches dadurch ebenfalls schlechtere Lernergebnisse erzielt.
Aus diesem Grund wurde nach einer anderen Bewertungsmethode gesucht. Nach diversen Testläufen mit verschiedenen Bewertungsmetriken wurde ein vom Java-Bot und den Tipps eines guten Jassspielers inspirierte Bewertungsmethode gewählt. Diese weist den verschiedenen Handkarten Werte zu und berechnet auf diese Weise eine Wertung für jede Trumpfvariante. Falls das Netzwerk den richtigen Trumpf gewählt hat, ist der Reward 0. Ansonsten ist der Reward die Differenz zwischen der Wertung des gewählten Trumpfs und der des richtigen Trumpfs.
Mit dieser Methode ist gewährleistet, dass die Bewertung des Trumpfentscheids völlig unabhängig von der Spielqualität des Gamenetzwerkes ermittelt werden kann und somit nicht von diesem negativ beeinflusst wird.

Versuch mit Gradientenclipping

Bei Versuchen mit früheren Netzwerken, auch im Zusammenhang mit den **NaN** Werten aus dem Experiment 2, wurde immer wieder festgestellt, dass die Werte der Gradienten sehr gross werden. Gradienten werden von der Optimierungsfunktion verwendet und geben an, wie stark die Gewichte verändert werden sollen. Grosse Gradienten bedeuten daher auch grosse Veränderungen im Netzwerk, welche schlussendlich zu NaN Werten führen können. Daher wurde mit Hilfe des Gradientenclippings versucht dies zu unterbinden.

Beim Gradientenclipping werden alle Werte, deren Betrag grösser als die spezifizierte Abweichung ist, auf diese maximale Abweichung zurückgesetzt. Dies verhindert, dass grosse Gradienten das Netzwerk pro Optimierungsschritt zu sehr verändern.
Die folgende Grafik vergleicht zwei Netzwerke mit und ohne Gradientenclipping.

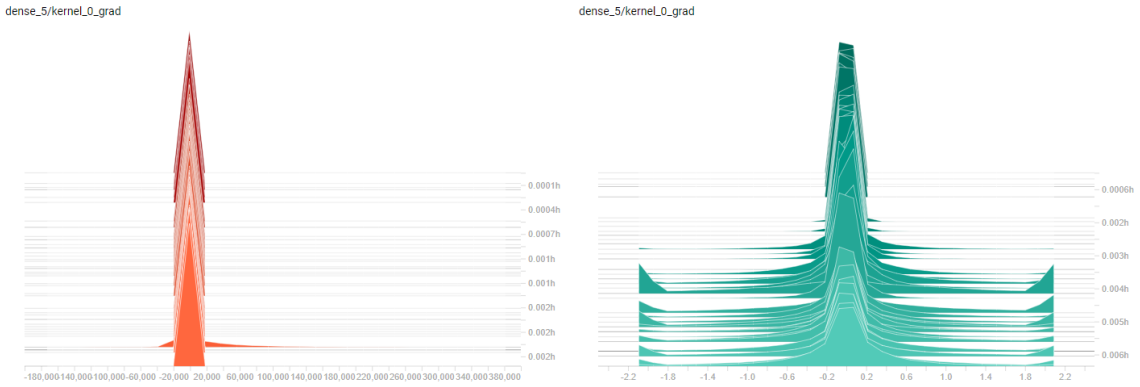


Abbildung 2.6.32: Vergleich Gradientenclipping

Die Häufigkeitsverteilung über die Zeit in der linken Grafik der Abbildung 2.6.32 zeigt, dass einige Gradientenwerte im fünfstelligen Bereich liegen und das Netzwerk damit stark verändern. Da beim Gradientenclipping alle Werte über dem definierten Schwellwert auf den Schwellwert gesetzt werden, ist im rechten Histogramm bei einem Schwellwert von 2 eine grössere Ansammlung zu sehen.
Schlussendlich hat sich die Hoffnung, durch das Gradientenclipping NaN Werte zu verhindern, nicht erfüllt und die Idee wurde nach den Tests wieder verworfen.

Erweiterung Temporal Difference Learning

Im Jass ist nicht nur der aktuelle Stich von Bedeutung. Es sollte nämlich je nach Situation auch mal eine niedrige Karte gespielt werden, um in späteren Stichen mit den hohen Karten zu stechen. Eine solche Abwägung zwischen sofortigem Gewinn und kurzfristig kleinem, aber längerfristig grösserem Gewinn, sollte auch der Jass-Bot erlernen können. Dazu wurde die Berechnung des Rewards erweitert. Zuvor wurde nur der erwartete Reward des direkt nachfolgenden Stiches noch miteinbezogen. Dies soll nun auf alle nachfolgenden Stiche in der aktuellen Runde ausgeweitet werden. Konkret bedeutet dies, dass sich der Reward des ersten Stiches R_1 aus dem aktuellen Reward r_1 (den gewonnen Punkten in diesem Stich) und dem zukünftig erwarteten Reward zusammensetzt. Der zukünftig erwartete Reward ist die Summe der erwarteten Rewards der einzelnen Stiche $r_{2:9}$ multipliziert mit einem grösser werdenden Discountfaktor $\gamma^{1:8}$ je weiter diese in der Zukunft liegen. Der Reward der ersten drei Stiche würde demnach folgendermassen berechnet werden:

$$R_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \gamma^4 r_5 + \gamma^5 r_6 + \gamma^6 r_7 + \gamma^7 r_8 + \gamma^8 r_9$$

$$R_2 = r_2 + \gamma r_3 + \gamma^2 r_4 + \gamma^3 r_5 + \gamma^4 r_6 + \gamma^5 r_7 + \gamma^6 r_8 + \gamma^7 r_9$$

$$R_3 = r_3 + \gamma r_4 + \gamma^2 r_5 + \gamma^3 r_6 + \gamma^4 r_7 + \gamma^5 r_8 + \gamma^6 r_9$$

Gamenetzwerk Reward

Der Reward für einen verlorenen Stich war bisher 0. Daher konnte das Netzwerk nicht lernen, ob mit diesem Stich viele oder womöglich gar keine Punkte verloren gingen. Um dies zu korrigieren, wurden nun die gewonnenen Stichpunkte als positive Rewards und die verlorenen als negative Rewards dem Netzwerk übergeben. Zusätzlich wurde bei einem Match, das heisst alle Stiche in dieser Runde wurden gewonnen, die zusätzlichen 100 Punkte mit in den Reward einberechnet. So soll auch hier ein Anreiz bestehen, möglichst alle Stiche zu gewinnen.

Ziel

Mit diesem Experiment wurde einerseits versucht, die Realität nochmals ein Stück genauer abzubilden. Andererseits wurden beide Rewardberechnungen überarbeitet, um auch dort noch ein besseres Feedback an das Netzwerk geben zu können. Zusammen mit der Erweiterung des Temporal Difference Learnings sollte der Bot nun genug gute Daten als Input haben, um auf einem akzeptablen Niveau zu spielen. Auch wenn mit dem Gradientenclipping keine Verbesserung erreicht wurde, konnte damit eine Möglichkeit für die hohen Werte, die das Netzwerk zum Absturz brachten, ausgeschlossen werden.

2.6.4 Hybridnetzwerk

Bereits sehr früh im Projekt kam die Idee auf, eine Kombination aus beiden Trainingsansätzen auszuprobieren. Dadurch sollte die natürliche Grenze des Supervised Learnings, welches durch die Qualität der Daten limitiert ist, überwunden werden und durch das Reinforcement Learning nochmals eine Leistungssteigerung erreicht werden.
Da aber bis zuletzt auch beim Supervised Learning und Reinforcement Learning noch Verbesserungen möglich waren und deshalb die Prioritäten auf diese beiden Ansätze gelegt wurden, konnte das Experiment mit dem Hybridnetzwerk erst in der letzten Implementationswoche durchgeführt werden. Deshalb konnte für diesen Netzwerkaufbau keine zusätzliche Entwicklungszeit aufgewendet werden. Daher wurden nur die bereits entwickelten und vorhanden Komponenten verwendet und in einem Netzwerk kombiniert. Dies bedeutet, dass das vortrainierte Gamenetzwerk aus dem Supervised Learning Experiment 8, Versuch 3 und das Trumpfnetzwerk aus dem Supervised Learning Experiment 10, Versuch 2 mit dem Reinforcement Learning Netzwerk aus Experiment 3 trainiert wurde.
Eine Auswertung im TensorBoard zeigt folgende Kennzahlen:

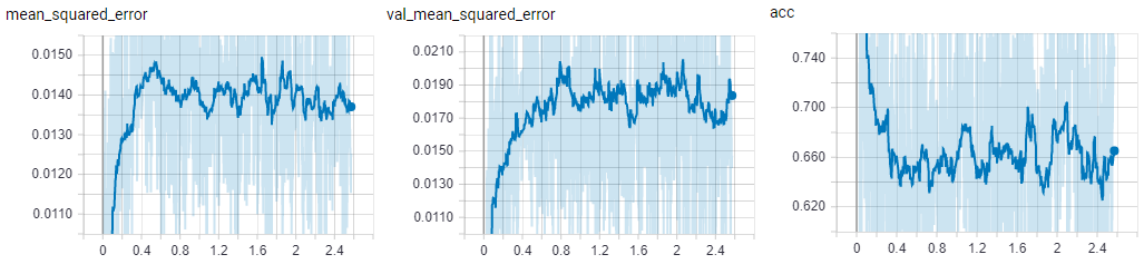


Abbildung 2.6.33: Auswertung Hybridnetzwerk

Aus den Diagrammen in Abbildung 2.6.33 ist eine anfängliche Verschlechterung und anschliessende Stagnation der Netzwerkperformance ersichtlich. Da zu diesem Netzwerk noch nicht sehr viele Auswertungsdaten zur Verfügung stehen, ist eine Analyse der Ursachen eher schwierig. Es wurden bisher zwei Vermutungen für diese Verschlechterung gefunden.
Die erste Vermutung ist, dass die Rewardberechnung des Reinforcement Learning die Realität noch nicht optimal wiedergibt und dadurch das vorher bessere Netzwerk mit dem Training wieder verschlechtert wird.
Ein anderer Grund könnte die zu kurze Trainingszeit sein. Es wird vermutet, dass sich das Netzwerk zuerst an die neue Trainingsart anpassen muss, um dann nach einer gewissen Zeit wieder besser werden zu können.

2.7 Auswertungen

In diesem Kapitel werden die Ergebnisse von Turnieren und das Feedback von menschlichen Spielern zusammengefasst.

2.7.1 Leistungsvergleich Prototypen

Zum Schluss der Implementierungsphase spielten die Prototypen und wichtige Zwischenversionen in einem Turnier gegeneinander. Das Ziel dieser Turniere war, einen Vergleich der Leistungen der verschiedenen Bots zu bekommen und vielversprechende Lösungsansätze zu identifizieren.

In der folgenden Tabelle sind alle im Turnier vertretenen Prototypen mit der jeweiligen Architektur des Gamenetzwerk sowie einem Verweis zum entsprechenden Versuch für detailliertere Informationen über das jeweilige Netzwerk aufgelistet.

Name	**Netzwerkstruktur**	**Experiment / Versuch**
Supervised186latest	186-560-1680-180-36	Experiment 8, Versuch 3
Supervised150	150-200-100-36	Experiment 3, Versuch 1
HybridBot	186-560-1680-180-36	Hybridnetzwerk
Reinforcement150	150-100-36	Experiment 2
Reinforcement186	186-420-950-420-186-80-36	Experiment 3
Supervised186	186-560-1680-180-36	Experiment 6, Versuch 2

Tabelle 2.7.1: Übersicht der im Turnier angetretenen Bots

Bei den in der Tabelle 2.7.1 aufgeführten Bots wurde immer das Trumpfnetzwerk aus dem Supervised Learning Experiment 10, Versuch 2 verwendet.

In einem Turnier spielte jeder Bot 150 Jassspiele gegen jeden anderen Bot. Beim ersten Turnier traten verschiedene Prototypen und ein RandomBot, der Karten zufällig spielt, gegeneinander an. So wurden in diesem Turnier insgesamt 3 150 Partien gespielt.

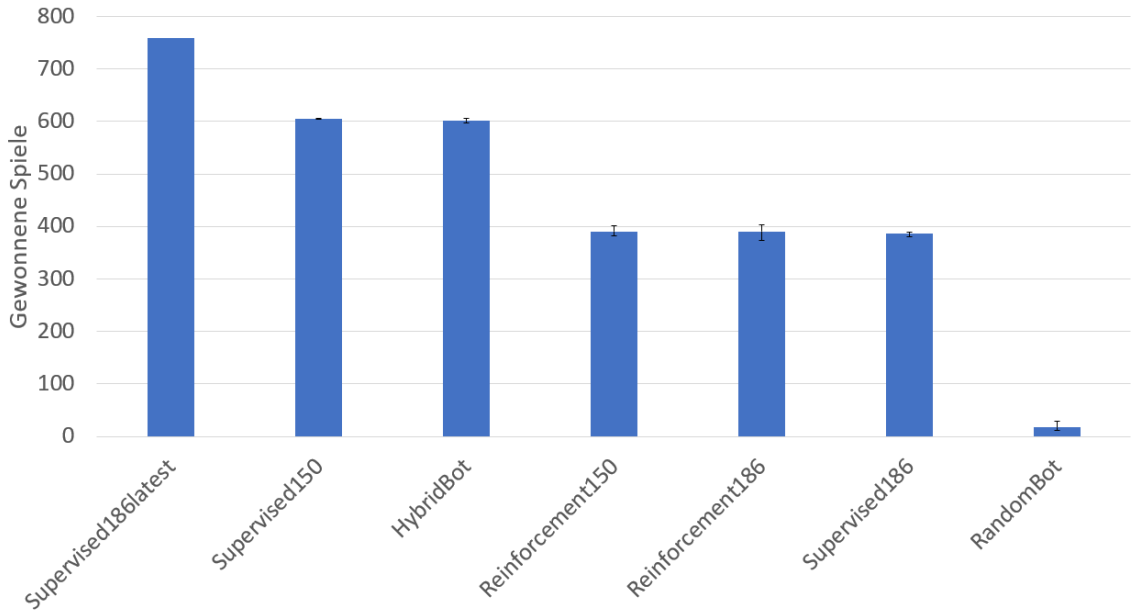


Abbildung 2.7.1: Turnier zwischen verschiedenen Prototypen

In Abbildung 2.7.1 ist zu sehen, dass alle Prototypen nur durch ihre verschiedenen Trainings deutlich besser wurden als der Bot, welcher nur zufällig Karten spielt. Die Verschlechterung des **HybridBots** ist in dieser Auswertung auch erkennbar. Der **HybridBot** stammt vom **Supervised186latest** Netzwerk ab und wurde danach noch mit Reinforcement Learning trainiert, wodurch er auf das Niveau des **Supervised150** Netzwerkes abstieg. Es ist ausserdem interessant, dass das **Supervised150** Netzwerk ohne Karten zu zählen ebenfalls ein relativ gutes Ergebnis erreichte.

Dadurch, dass der JavaBot stets nach etwa einer Stunde nicht mehr weiterspielte, wurde das erste Turnier nur unter den Prototypen und dem RandomBot ausgetragen. Um dennoch einen Vergleich zum angestrebten Ziel, dem JavaBot, zu erhalten, trat am Schluss der beste Prototyp in 450 Spielen gegen den JavaBot an.
Beide Turniere wurden je dreimal durchgeführt, um so auch eventuelle Schwankungen in den Resultaten ausweisen zu können.

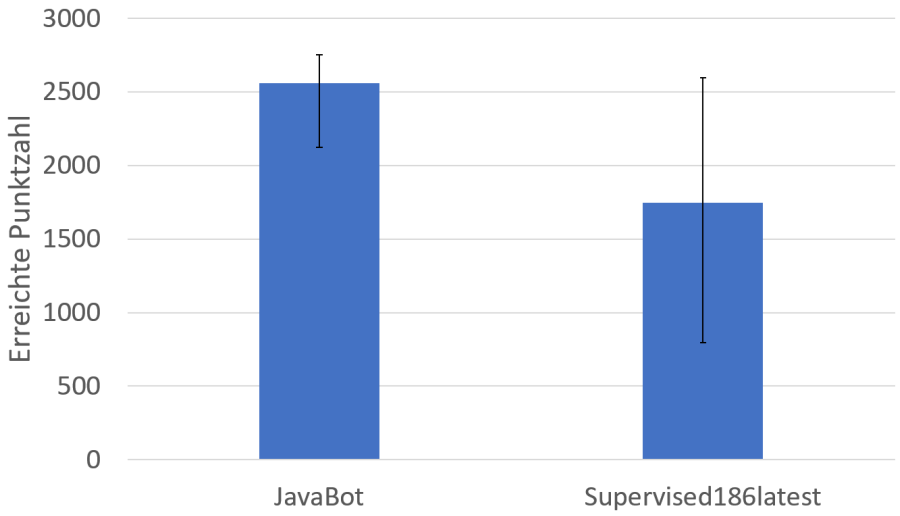


Abbildung 2.7.2: JavaBot gegen den besten Prototyp

Aus der Auswertung in Abbildung 2.7.2 ist ersichtlich, dass der JavaBot im Moment noch besser ist als der beste Prototyp. Immerhin hat der Prototyp in den 450 durchgeführten Spielen den JavaBot in 7 besiegt. Natürlich kann dies auch zum Teil auf das Kartenglück zurückzuführen sein.

Erkenntnisse

Aus diesen Turnieren können die folgenden Erkenntnisse zu den verschiedenen Netzwerken gewonnen werden:
Das **Supervised150** Netzwerk ist auch ohne Kartenzählen bereits sehr gut und sogar besser als eine frühe Version des Supervised Learning Netzwerkes und die aktuelle Reinforcement Learning Version.
Bei den Netzwerken mit implementiertem Kartenzählen (**Supervised186** und **Supervised186latest**) konnte eine deutliche Verbesserung im Laufe des Projekts festgestellt werden. Diese Art von Netzwerk war zu Beginn noch deutlich schlechter als ein Netzwerk ohne Kartenzählen. Durch diverse Verbesserungen schaffte es dieses Netzwerk zum Schluss mit deutlichem Abstand an die Spitze der Rangliste.
Die mit Reinforcement Learning trainierten Netzwerke (**Reinforcement150** und **Reinforcement186**) sind zur Zeit noch auf der Stufe eines frühen Supervised Learning Netzwerkes mit integrierem Kartenzählen. Dies kann damit erklärt werden, dass beim Reinforcement Learning der Fokus hauptsächlich auf der Implementierung lag und erst wenig Zeit in die Optimierung der verschiedenen Parameter investiert werden konnte.

Beim **HybridBot** ist dieses Defizit des Reinforcement Learnings am deutlichsten zu sehen. Dieser war auf dem selben Stand wie das Supervised186latest bevor er mit Reinforcement Learning trainiert wurde. Durch das Training verschlechterte sich das Netzwerk auf das Niveau des Supervised150 Netzwerks.
Erfreulich ist, dass alle Netzwerke den RandomBot deutlich besiegten.

Beim zweiten Turnier gegen den **JavaBot** konnte das **Supervised186** Netzwerk diesen in ein paar wenigen Spielen besiegen. Jedoch verlor es zum Teil auch sehr deutlich. Diese vereinzelten Gewinne gegen den JavaBot zeigen, dass der Prototyp auf gutem Weg ist ein guter Jassspieler zu werden. Mit weiteren Optimierungen sollte es möglich sein nochmals besser zu werden.

2.7.2 Benutzerfeedback

Im Rahmen eines Benutzerfeedbacks wurde mehreren erfahrenen Jassspielern die Möglichkeit gegeben gegen den besten Prototyp des JassBots anzutreten. Dabei jassten die Spieler mit einer Instanz des besten Bots im Team gegen das gegnerische Team aus zwei Instanzen. In diesem Feedback werden diverse Situationen geschildert, in welchen sich die Bots anders verhalten haben, als die Testpersonen dies erwartet hätten.

Die Jassspieler waren grundsätzlich erfreut über die Möglichkeit mit einem auf Machine Learning basierenden JassBot zu spielen und waren auch von dem einen oder anderen guten Zug positiv überrascht.

Die im folgenden aufgeführten Beobachtungen sind als Verbesserungsvorschläge zu verstehen. Einige dieser nicht optimalen Entscheidungen würden auch von Gelegenheitsjassern gemacht werden. Da in den für das Training verwendeten Daten vermutlich auch viele Spiele mit solchen Spieler existieren, kann das Verhalten der Bots teilweise darauf zurückgeführt werden:

Verschenken von Trumpfkarten
Der Bot spielt zu Beginn weiter Trumpfkarten aus, obwohl die Gegner keine Trümpfe mehr besitzen. Dabei entstand auch die Situation, dass nur noch die zwei Bots, welche im Team spielten, Trümpfe besassen und sich dann gegenseitig gezwungen haben die Trumpfkarten abzulegen. So werden unnötig Trumpfkarten verschenkt, welche für spätere Stiche nützlich hätten sein können.

Partner abgestochen
Es kam auch vor, dass der Bot seinen Teamkollegen abgestochen hat. Das bedeutet der Bot, welcher als zweiter im Team eine Karte spielt hat den Stich genommen, obwohl der Stich bereits seinem Team gehörte. Hier wurde erneut eine später nützliche Karte verschwendet.

Punkte verschenkt
Falls klar ist, dass der aktuelle Stich nicht an das eigene Team gehen kann, so sollte eine niedrige Karte ausspielen, welche sowohl taktisch wie auch punktemässig keinen grossen Wert besitzt. In dieser Situation haben die Bots allerdings zum Teil eine relativ gute Karte gespielt und diese so unnötig verschenkt.

Zu wertvolle Karten gespielt
Zum Teil stachen die Bots mit sehr guten Karten auch relativ wertlose Stiche, welche in der Punktewertung nicht viel zählen und somit das Team einem Sieg nicht näher bringen. Die guten Karten, welche so eingesetzt wurden, wären für wertvollere Stiche hilfreich gewesen.

Ungeeignete Kartenwahl
Bei den Trumpfarten UNDEUFE und OBEABE werden normalerweise zuerst die besten Karten gespielt, da es ansonsten schwierig sein kann wieder einen Stich für sich zu entscheiden und die guten Karten nutzen zu können. Dies war bei den Bots nicht immer der Fall und sie begannen teilweise mit schlechteren Karten, was dazu führte, dass sie ihre Karten verwerfen mussten.

Ungeeignete Startkarte
Teilweise, wenn ein Bot einen Stich eröffnete, wurde eine sehr gute Karte gespielt, mit welcher eventuell später ein wertvollerer Stich hätte gemacht werden könnte. In einem solchen Fall wurde wiederum eine gute Karte verworfen.

Nicht Berücksichtigung von Spezialregeln
Falls eine Farbe gespielt ist, muss ein Spieler so lange diese Farbe oder Trumpf spielen bis er diese Farbe nicht mehr besitzt. Es ist beliebt zu Beginn eines Spiels ein paar Runden mit dem Trumpf zu beginnen, um so den Gegnern möglichst viele Trümpfe zu entziehen. Dabei müssen alle immer Trumpf spielen solange sie Trumpfkarten besitzen. Die einzige Ausnahme ist der Puur[3], welcher die höchste Karte im Spiel ist. Diese muss nicht gespielt werden und kann später für einen Stich mit vielen Punkten genutzt werden. Diesen Vorteil haben die Bots ebenfalls meistens ausser Acht gelassen und so diese wertvolle Karte verschenkt.

Schmieren des gegnerischen Teams
Spielt der Teampartner die höchste noch im Spiel verbliebene Karte und hat den Stich somit auf sicher, kann der zweite Spieler im Team den Stich schmieren. Dies bedeutet es wird eine punktemässig wertvolle Karte gespielt, um sich so möglichst viele Punkte zu sichern. Es geschah teilweise, dass die Bots nicht nur ihr eigenes Team schmierten, sondern gelegentlich auch das gegnerische Team. Dadurch verschenkten sie unnötig Punkte.

Kommunikation mittels Karten
Es konnte zur Zeit auch noch keine Kommunikation über die Karten zwischen den spielenden Bots beobachtet werden. Mit dem legen bestimmter Karten kann dem Partner mitgeteilt werden, welche Farbe bevorzugt wird oder welche Farbe nicht gewünscht wird. Durch diese Kommunikation kann ein Team besser miteinander zusammenspielen und die gegenseitigen sowie gemeinsamen Vorteile kombinieren.

2.8 Endergebnisse

Im Rahmen dieser Arbeit wurden zwei Prototypen mit unterschiedlichen Ansätzen für dasselbe Problem entwickelt.

Bei der Umsetzung des Supervised Learning Prototypen wurden viele verschiedene Experimente zu verschiedenen Architekturen und Parametern durchgeführt. Diese Experimente halfen optimale Voraussetzungen für das Training mit Supervised Learning zu schaffen und schlussendlich einen Prototypen zu trainieren, dessen Leistung bereits knapp an den JavaBot heranreicht.

Der mit Reinforcement Learning trainierte Prototyp konnte viele Erkenntnisse aus der Entwicklung des Supervised Learning Prototypen übernehmen. Der Reinforcement Learning Prototyp stand hingegen vor seinen ganz eigenen Herausforderungen. So musste für diesen eine möglichst realitätsnahe Rewardberechnung ermittelt werden, um dem Netzwerk ein optimales Feedback geben zu können. Zudem musste bei der Berechnung der Rewards für das Trumpfnetzwerk darauf geachtet werden, dass diese nicht von der Spielqualität des Gamenetzwerkes abhängig sind.

Schlussendlich entstanden zwei Prototypen von welchen der Supervised Learning Prototyp bereits sehr gute Resultate erzielte. Die Verbesserung von diesem Prototyp kann mit Hilfe des Benutzerfeedbacks genauer angegangen werden.
Der Reinforcement Learning Prototyp sollte nach mehr Entwicklungszeit ebenfalls noch bessere Spielergebnisse erreichen können.
Der Hybridprototyp weist im ersten Entwurf eine Verschlechterung auf, welche jedoch genauer analysiert werden muss. Dieser Ansatz ist ebenfalls sehr vielversprechend.

Die Stabilität des JavaBots, welcher den Zühlke Jasswettbewerb gewonnen hatte, lässt jedoch sehr zu wünschen übrig. Für längere Turniere oder Trainingseinheiten gegen den JavaBot müsste dieser zuerst noch überarbeitet und stabilisiert werden.

2.9 Ausblick

Im Weiteren werden Überlegungen aufgeführt, die sich das Projektteam bereits gemacht hatte, aber zeitlich nicht mehr realisieren konnte. Daher sollen diese Ideen Teams, welche diese Arbeit fortführen wollen, zur Verfügung stehen.

2.9.1 Ausbau des Gamenetzwerk

Eine genauere Abbildung der Realität kann durch die Vergrösserung des Input Layers erreicht werden. Eine Verbesserung des Bots durch die neue Grösse ist jedoch nicht garantiert. Ausserdem würde für das Training mit Supervised Learning und einem grösseren Input Layer mehr Daten benötigt, als dem Projektteam bisher vorlagen.

Das Design dieser Ausbaustufe sieht vor, dass das neuronale Netzwerk aus dem Input Layer die folgenden Informationen erkennen kann:

- Handkarten des Bots
- Die jeweils gespielte Karte eines Spielers in einem Stich
- Die Reihenfolge, wann welche Karte gespielt wurde
- Welcher Trumpf gewählt wurde

Daraus ergibt sich folgende Zusammenstellung:

- 36 mögliche Handkarten
- 3 $\times$ 36 mögliche Karten, die für den aktuellen Stich auf dem Tisch liegen
- 8 Stiche à 4 Spieler mit 36 möglichen Karten, wobei der letzte Stich nicht separat geführt werden muss
- 6 mögliche Trumpfvarianten

Die Berechnung für einen Input Layer mit diesen Designüberlegungen ist unter 2.5 ersichtlich:

$$36 \ + \ 3 \times 36 \ + \ 8 \times 4 \times 36 \ + \ 6 = 36 + 108 + 1\,152 + 6 = 1\,302 \tag{2.5}$$

Somit würde der Input Layer 1 302 Neuronen aufweisen. Dies beeinflusst auch die restlich Grösse des Netzwerkes, wie die Tiefe, also Anzahl der Hidden Layers und die Anzahl der Neuronen pro Hidden Layer.

2.9.2 Bessere Realitätsabbildung

Diese Arbeit wurde nach den Spielregeln des Zühlke Jasservers erstellt. Da dieser Server in der Turnierkonfiguration keine Weise, Stöcke und Multiplikatoren[25] kennt[9], wurden diese auch nicht berücksichtigt. Falls diese Arbeit als Grundlage für einen JassBot für eine andere Plattform dienen soll, müssten diese zusätzlichen Regeln beachtet werden. Dafür müssten mehr Informationen an das Netzwerk geliefert werden können. Ausserdem ist eine Überarbeitung des Rewards beim Reinforcement Learning notwendig.

2.9.3 Datenanfrage

Da solche Netzwerke schnell grosse Dimensionen annehmen können, werden beim Supervised Learning auch entsprechend viele Daten für das Training benötigt. Swisslos hat auf Anfrage des Projektteams extra die Datenaufzeichnung für circa 2 Wochen aktiviert und die Daten kostenlos zur Verfügung gestellt. Für eine Fortsetzungsarbeit wird empfohlen ebenfalls frühzeitig bei Online-Jassportalen anzufragen, um sicher genügend Daten für das Training zur Verfügung zu haben.

2.9.4 Dropout

Damit ein Netzwerk besser auf generelle und weniger auf spezifische Situationen trainiert werden kann, könnte ein Netzwerk mit Dropout[22] implementiert werden. Dabei werden zufällig einzelne Neuronen abgeschaltet, sodass bei jedem Training ein etwas anderes Netzwerk trainiert wird und sich Neuronen weniger zusammenschliessen können.

2.9.5 Reinforcement Learning

Während dieser Arbeit lag der Fokus vor allem auf den grundlegenden Parametern wie der Architektur des Netzwerkes, der Aktivierungs-, der Optimierungs- und der Lossfunktionen. Dadurch konnten auch beim Reinforcement Learning bereits gute Resultate erreicht werden. Vor allem das Tuning des Rewards sowohl beim Gameals auch beim Trumpfnetzwerk könnte noch weitere Leistungssteigerungen bringen. Durch eine Verbesserung des Reinforcement Learning Ansatzes kann schlussendlich im Zusammenspiel mit dem Supervised Learning auch ein erfolgreicherer Hybridprototyp trainiert werden.

[9] https://github.com/webplatformz/challenge/wiki/Jass-Rules

2.9.6 Rekurrentes neuronales Netzwerk

Zurzeit wird das Kartenzählen noch von der Umgebung übernommen, in der sich das neuronale Netzwerk befindet. Ein Prototyp bei dem alle Aufgaben vom neuronalen Netzwerk übernommen werden, könnte mit Hilfe eines rekurrenten neuronalen Netzwerkes[16] realisiert werden. Dabei sind Neuronen nicht wie bei einem vorwärtsgerichteten Netzwerk (feedforward network) nur zum nächsten Layer verbunden, sondern können auch untereinander im gleichen Layer verbunden sein, wie dies Abbildung 2.9.1 dargestellt ist. Dadurch kann ein künstliches Gedächtnis geschaffen werden und das Netzwerk sollte in der Lage sein, bereits gespielte Karten selbst merken zu können.

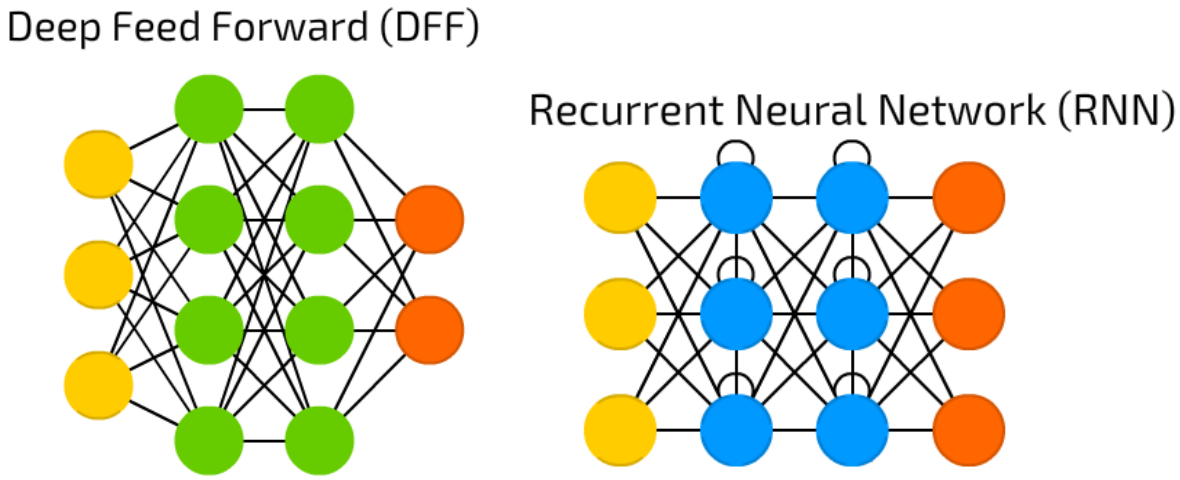


Abbildung 2.9.1: Vergleich zwischen DFF und RNN[26]

2.10 Schlussfolgerung

Das Entwickeln eines Reinforcement Learning Netzwerkes hatte viel mehr Zeit in Anspruch genommen als ursprünglich vermutet. Ein Grund dafür war, dass das Training nicht nur aus der Entwicklungsumgebung gestartet werden konnte, sondern zusätzlich noch sichergestellt werden musste, dass der Server lief, ein Turnier erstellt wurde und die richtige Anzahl Gegner verbunden war. Unter anderem auch deshalb wurden die meisten Designänderungen mit dem Supervised Learning Netzwerk getestet und dann eventuell in das Reinforcement Learning Netzwerk übernommen.

Die mit dieser Arbeit entstandenen Prototypen können bereits als gute Gegner für Gelegenheitsspieler verwendet werden. Auch wenn gute Jasser und Jasserinnen diese Bots problemlos besiegen können, wurde doch einiges erreicht. Mit der vorliegenden Arbeit und den entstandenen Prototypen wurde eine solide Grundlage geschaffen. Eine auf dieser Arbeit aufbauende Entwicklung könnte durchaus in der Lage sein, einen dem JavaBot gleichwertigen oder sogar besseren JassBot hervorzubringen. Einige vielversprechende Ideen, die aus Zeitgründen nicht während dieser Arbeit umgesetzt werden konnten, wurden im Kapitel 2.9 dokumentiert.

Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Quellenverzeichnis

[1] Tran The Anh. [ML – 10] Regularization – Overfitting and Underfitting. http://labs.septeni-technology.jp/technote/ml-10-regularization-overfitting-and-underfitting/, 2016. [Online; Zuletzt zugegriffen 17.12.2017].

[2] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In Leon Bottou, Olivier Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*. MIT Press, 2007. URL http://www.iro.umontreal.ca/~lisa/pointeurs/bengio+lecun_chapter2007.pdf.

[3] bluewin. Kapitel 4: Trumpf. https://jass.bluewin.ch/de/Jass-lernen/Jass-Grundlagen/Jass-Die-Einfuehrung/Kapitel-4-Trumpf. [Online; Zuletzt zugegriffen 17.12.2017].

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810. URL http://mcts.ai/pubs/mcts-survey-master.pdf.

[5] Jason Brownlee. Practical Machine Learning Problems. https://machinelearningmastery.com/practical-machine-learning-problems/, 2013. [Online; Zuletzt zugegriffen 17.12.2017].

[6] r3dDoX Florian Lüscher, Adrian Herzog. Jasschallenge Wiki. https://github.com/webplatformz/challenge/wiki. [Online; Zuletzt zugegriffen 17.12.2017].

[7] William Anton Rohm Gary Ericson. Flavors of machine learning. https://docs.microsoft.com/en-us/azure/machine-learning/studio/algorithm-choice, 2017. [Online; Zuletzt zugegriffen 17.12.2017].

[8] Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, pages 1756–1760, Lyon, France, August 2013. URL https://www-i6.informatik.rwth-aachen.de/publications/downloader.php?id=861&row=pdf.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. ISBN 9780262035613.

[10] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T. URL http://zmjones.com/static/statistical-learning/hornik-nn-1991.pdf.

[11] Keras. Getting started with the Keras functional API. https://keras.io/getting-started/sequential-model-guide/, . [Online; Zuletzt zugegriffen 17.12.2017].

[12] Keras. Getting started with the Keras Sequential model. https://keras.io/getting-started/functional-api-guide/, . [Online; Zuletzt zugegriffen 17.12.2017].

[13] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014. URL http://adsabs.harvard.edu/abs/2014arXiv1412.6980K.

[14] Dani Müller. *STÖCK - WYS - STICH*. Fona Verlag, 2016. ISBN 9783037810910.

[15] Nvidia. NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/. [Online; Zuletzt zugegriffen 17.12.2017].

[16] Christopher Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. [Online; Zuletzt zugegriffen 17.12.2017].

[17] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training Recurrent Neural Networks. *ArXiv e-prints*, November 2012. URL http://adsabs.harvard.edu/abs/2012arXiv1211.5063P.

[18] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv e-prints*, December 2017. URL https://arxiv.org/abs/1712.01815.

[19] David Silver. Technical perspective: Solving imperfect information games. *Commun. ACM*, 60(11):80–80, October 2017. ISSN 0001-0782. doi: 10.1145/3131286. URL http://doi.acm.org/10.1145/3131286.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. URL http://dx.doi.org/10.1038/nature16961. Article.

[21] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den

Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354 EP –, Oct 2017. URL http://dx.doi.org/10.1038/nature24270. Article.

[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[23] M. Stinchcombe and H. White. Approximating and learning unknown mappings using multilayer feedforward networks with bounded weights. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 7–16 vol.3, June 1990. doi: 10.1109/IJCNN.1990.137817. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5726775.

[24] Swisslos. Jass-Regeln. https://www.swisslos.ch/de/jass/informationen/jass-regeln/jass-grundlagen.html, . [Online; Zuletzt zugegriffen 17.12.2017].

[25] Swisslos. Schieber Jass-Regeln. https://www.swisslos.ch/de/jass/informationen/jass-regeln/schieber-jass.html, . [Online; Zuletzt zugegriffen 17.12.2017].

[26] Fjodor van Veen. The neural network zoo. http://www.asimovinstitute.org/neural-network-zoo/, 2016. [Online; Zuletzt zugegriffen 17.12.2017].

Glossar

Adam Eine kürzlich veröffentlichte Optimierungsfunktion, welche im Vergleich zum SGD effizienter ist.

AI Artificial Intelligence, englisch für Künstliche Intelligenz.

categorical crossentropy Eine belibte Lossfunktion in neuronalen Netzwerken für Klassifizierungsprobleme.

CNTK Ein Deep Learning Framework von Microsoft.

CUDA Eine API von Nvidia für die Nutzung von GPUs.

DGX Server Nvidia DGX-1 supercomputer des ICOM mit 8 GPUs, auf welchen mittels dedizierten Docker Containern Trainings- und Testprozesse für neuronale Netzwerke durchgeführt werden können.

Docker Eine Umgebung für die Virtualisierung mittels Containern.

Docker Hub Eine Plattform für den Bezug und die Bereitstellung von Docker Images.

Gradientenclipping Das zurücksetzen von Gradienten ab einem bestimmten Schwellwert (On the difficulty of training Recurrent Neural Networks, Kapitel 3.2[17]).

ICOM Institute for Communication, Institut des Fachbereich Elektrotechnik.

Keras Eine API, welche TensorFlow weiter abstrahiert.

KI Künstliche Intelligenz, deutsche Bezeichnung für Artificial Intelligence.

Leaky ReLU Eine modifizierte Version der Aktivierungsfunktion ReLu, bei der die negativen Werte nicht auf Null gesetzt werden sondern einen negativen Wert ergeben. Die negativen Werte sind ebenfalls linear angeordnet besitzen jedoch eine kleinere Steigung als die Werte im positiven Bereich.

Mean squared error Mittlerer quadratischer Fehler oder auch mittlere quadratische Abweichung bezeichnet in der Statistik, wie stark eine Schätzfunktion um den zu schätzenden Wert streut.

ReLU Rectified linear unit ist eine Aktivierungsfunktion, welche alle negativen Werte auf null setzt und die positiven Werte in der auftretenden Identität weiterleitet.

SGD Stochastic gradient descent ist eine Optimierungsfunktion, welche das Prinzip der stochastischen Ableitung verwendet.

TensorBoard Das browserbasierte Visualisierungswerkzeug von TensorBoard zur Darstellung von aufgezeichneten Messungen, Graphen und Metriken.

TensorFlow Ein Machine Learning Framework von Google.

Theano Eine Open Source Bibliothek für Machine Learning.

FrozenLake-v0

December 17, 2017

1 Vorstudie 1: FrozenLake

Zweck Diese Vorstudie dient zum allgemeinen Verständnis von Tensorflow.

Beschreibung Das gelöste Problem ist der FrozenLake aus der Sammlung des OpenAI Gyms. Die Aufgabe auf dem FrozenLake(4x4 Matrix) ist vom Startpunkt in der einen Ecke zum Zielpunkt in der gegenüberliegenden Ecke zu kommen. Da der See nicht komplett gefroren ist, gibt es einzelne Löcher, welche es zu umgehen gilt. Zusätzlich windet es auf dem See, so dass es sein kann, dass anstatt die gewählte Aktion eine andere zufällige ausgeführt wird.
Als Inputparameter dient ein Vektor mit 16 Werten, welcher den aktuellen Standort des Spielers darstellt.

Die folgende Analyse der Lösung von Arthur Juliani soll die Struktur und einige grundlegenede Befehle der Machine Learning Bibliothek TensorFlow erläutern.

Quellen https://gist.github.com/awjuliani/4d69edad4d0ed9a5884f3cdcf0ea0874

1.1 Lösung

```
In [7]: import gym
        import numpy as np
        import random
        import tensorflow as tf
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Nach dem Importieren wird die Umgebung (Environment) vom OpenAI Gym geladen und der TensorFlow Graph zurückgesetzt.

```
In [8]: env = gym.make('FrozenLake-v0')

        tf.reset_default_graph()
```

```
[2017-10-21 13:54:01,290] Making new env: FrozenLake-v0
```

1.1.1 Netzwerk erstellen

Im nächsten Schritt wird das neuronale Netzwerk (in TensofFlow **Graph** genannt) aufgebaut. In diesem Beispiel besteht dieser lediglich aus einem Input Layer (16 Nodes, Spielerposition) und einem Output Layer (4 Nodes, mögliche Aktionen). Komplexere Netzwerke können zusätzlich noch beliebig viele Hidden Layers besitzen.

Für den Input Layer (**inputs1**) werden Platzhalter verwendet, welche dann später durch die spezifischen Werte ersetzt werden. In der Variable **w** werden die Gewichte der Vernetzung zwischen den beiden Layern gespeichert. Diese werden hier zufällig mit Werten zwischen 0 und 0.1 initialisiert. **qout** ist der Output Vektor, der die erwartete Belohnung (Reward) für jeden der einzelnen Schritte enthält. Der Schritt mit dem höchsten erwarteten Reward wird mit der argmax() Funktion bestimmt.

```
In [9]: #These lines establish the feed-forward part of the network
        # used to choose actions
        inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32)
        w = tf.Variable(tf.random_uniform([16,4],0,0.01))
        qout = tf.matmul(inputs1,w)
        predict = tf.argmax(qout,1)
```

1.1.2 Loss Funktion erstellen

Eine Loss Funktion berechnet einen Verlustwert (loss) zu jeder Action. In diesem Fall ist der Verlustwert die quadrierte Differenz zwischen dem optimalen Schritt und dem gemachten Schritt.

Der Trainer (**trainer**) passt dann die Gewichte an, um diesen Wert möglichst zu minimieren. Die Learning Rate (**learning_rate**), welche für den Optimizer spezifiziert wird, definiert die Grösse des Betrags, um welchen die Gewichte bei jeder Optimierung angepasst werden sollten. Ist die Learning Rate klein, wird gelerntes nur langsam übernommen. Bei einer grossen Learning Rate kann es sein, dass überkorrigiert wird und dies beim nächsten Durchgang wieder korrigiert werden muss. Die Learning Rate definiert wie schnell altes Wissen durch neugelerntes ersetzt werden soll (0.1 = erst nach mehreren Erfahrungen, 1 = sofort).

```
In [10]: #Below we obtain the loss by taking the sum of squares difference
         # between the target and prediction Q values.
         nextq = tf.placeholder(shape=[1,4],dtype=tf.float32)
         loss = tf.reduce_sum(tf.square(nextq - qout))
         trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
         update_model = trainer.minimize(loss)
```

1.1.3 Netz trainieren

Bis jetzt wurde der Graph erst erstellt, es wurden noch keine Teile davon ausgeführt.

Um das Netz zu trainieren, wird eine Session erstellt und die Variablen initialisiert. In diesem Beispiel werden 2000 Spieldurchläufe (Episoden) gespielt, wobei pro Spiel so lange gespielt wird bis der Spieler einen Endzustand erreicht (auf dem Ziel steht oder durch das Eis gebrochen ist) oder 99 Schritten gemacht hat.

Zu Beginn einer Episode wird die Umgebung zurückgesetzt, was eine initiale Observation (den Inputvektor) zurückgibt.

Entscheidung fällen Mit sess.run(...) wird der Graph oder Teile davon ausgeführt.
Als erstes wird auf Zeile 23 der erwartete Reward der vier möglichen Aktionen (**qout**) berechnet und daraus die auszuführende Aktion (**predict**) gewählt. Für die Berechnung wird der Platzhalter (**inputs1**) durch einen Vektor mit der aktuellen Position des Spielers ersetzt.

In einzelnen Fällen wird die durch das Netz gewählte Aktion durch eine Zufällige ersetzt, um so auch ansonsten nicht ausprobierte Richtungen zu testen.

Mit der step() Funktion der Umgebung wird eine Aktion ausgeführt. Die Funktion gibt als Resultat die folgenden Parameter zurück:

- **observation (object):** an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.
- **reward (float):** amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- **done (boolean):** whether it's time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)
- **info (dict):** diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

(Source: OpenAI Gym Documenation)

Entscheidung bewerten Im zweiten Teil der Session wird die oben getroffene Entscheidung bewertet. Dies geschieht in dem die maximal zu erwartende Belohung für den nächsten Schritt (**maxq1**) berechnet wird. Gegen diesen Wert wird der Wert, der zu diesem Schritt geführt hat, verglichen (siehe loss Funktion oben).

Bellman Equation Der Targetreward, welcher später verwendet wird um die Abweichung der Vorhersage zu bestimmen, wird mit der Bellman Gleichung berechnet:

Q(s,a) = r + (max(Q(s',a'))

wobei * **r** der Reward des ausgeführten Schrittes ist * ein Faktor, der bestimmt wie sehr zukünftig erwartete Rewards berücksichtigt werden * **max(Q(s',a')** die maximale erwartete Belohnung im nächsten Schritt darstellt.

Je grösser der Unterschied zwischen dem erwarteten (**predict**) und dem realen Reward ist, desto stärker wird die Wertematrix **w** angepasst. Die Anpassungen werden mittels update_model aufgerufen und in der minimize() Funktion des Optimizers (siehe oben) berechnet und angewendet.

```
In [ ]: init = tf.global_variables_initializer()

        # Set learning parameters
        y = .99
        e = 0.1
        num_episodes = 2000
```

```
        #create lists to contain total rewards
        r_list = []

        with tf.Session() as sess:
            sess.run(init)
            for i in range(num_episodes):
                #Reset environment and get first new observation
                s = env.reset()
                r_all = 0
                d = False
                j = 0
                #The Q-Network
                while j < 99:
                    j+=1
                    #Choose an action by greedily (with e chance of random action)
                    # from the Q-network
                    a,allq = sess.run([predict,qout],
                                      feed_dict={inputs1:np.identity(16)[s:s+1]})
                    if np.random.rand(1) < e:
                        a[0] = env.action_space.sample()
                    #Get new state and reward from environment
                    s1,r,d,_ = env.step(a[0])
                    #Obtain the Q' values by feeding the new state
                    # through our network
                    q1 = sess.run(qout,
                                  feed_dict={inputs1:np.identity(16)[s1:s1+1]})
                    #Obtain maxQ' and set our target value for chosen action.
                    maxq1 = np.max(q1)
                    targetq = allq
                    targetq[0,a[0]] = r + y*maxq1
                    #Train our network using target and predicted Q values
                    _,w1 = sess.run([update_model,w],
                                    feed_dict={inputs1:np.identity(16)[s:s+1],
                                               nextq:targetq})
                    r_all += r
                    s = s1
                    if d == True:
                        #Reduce chance of random action as we train the model.
                        e = 1./((i/50) + 10)
                        break
                r_list.append(r_all)

        print("Percent of succesful episodes: " + str(sum(r_list)/num_episodes) + "%")
```

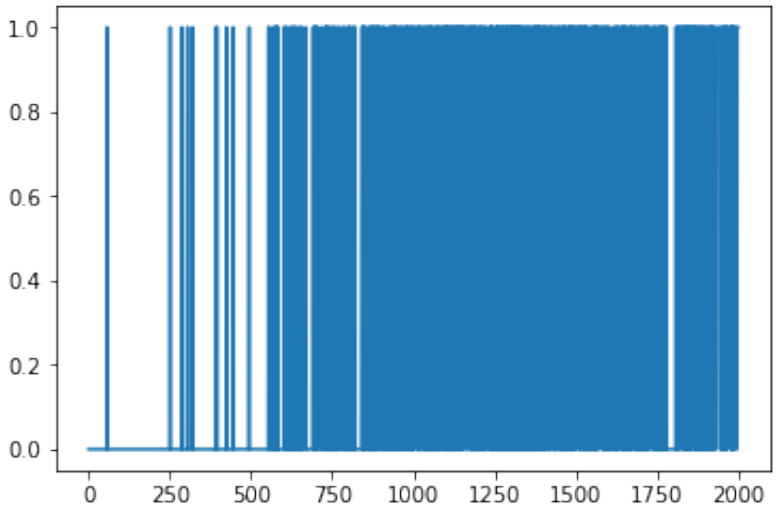
```
Percent of succesful episodes: 0.3145%
```

1.1.4 Auswertung

Das folgende Diagramm zeigt grafisch die gewonnenen (blau) und verlorenen (weiss) Episoden an.

```
In [12]: plt.plot(r_list)
```

```
Out[12]: [<matplotlib.lines.Line2D at 0x7fb659dc7860>]
```



Taxi-v1

December 17, 2017

1 Vorstudie 2: Taxi

Zweck Dieses Experiments soll die grundlegenden Funktionen zur Speicherung von Datenpunkten für die spätere Darstellung mittels TensorBoard zeigen.

Beschreibung Das Taxi Problem stammt ebenfall aus der Sammlung des OpenAI Gyms. Das Spielfeld besteht aus einer 5x5 Matrize auf welcher sich das Taxi befindet. Auf dem Spielfeld sind 4 Positionen markiert, welche mögliche Ein- oder Ausstiegsorte für den Passagier darstellen. Die Aufgabe des Taxi ist nun den Passagier auf dem richtigen Feld aufzunehmen (pick up action) und an einer bestimmten Position abzuladen (drop off action). Dies sollte natürlich bestenfalls auf dem kürzesten Wege passieren. Das erfolgreiche Abladen des Passagier gibt 20 Punkte während jeder Zug 1 Punkt abzug gibt. Unzulässige Aufnahme- oder Abladeversuche werden mit -10 Punkten bestraft.

TensorBoard Im folgende werden einige grundlegende Logfunktionen von TensorFlow erklärt.

Das skalare Ergebnis einer Operation loggen:

```
tf.summary.scalar("Loss", loss)
```

Falls mehrere Variablen geloggt werden, können die Logfunktionenen zu einer einzigen Operation zusammengefasst werden und müssen so nicht einzeln ausgeführt werden.

```
summary_op = tf.summary.merge_all()
```

Innerhalb der Session wird ein writer definiert, um die Logs zu speichern. Der angegebene Pfad im writer muss mit dem Pfad im TensorBoard übereinstimmen.

```
writer = tf.summary.FileWriter("/mnt/notebooks/logs", graph=tf.get_default_graph())
```

Auch die Logging Operation muss wie die andern Operationen mit einem run Befehlt ausgeführt werden.

```
_,W1,l,summary = sess.run([updateModel,W,loss,summary_op],
feed_dict={inputs1:np.identity(500)[s:s+1],nextQ:targetQ})
```

Anderweitig berechnete Werte können über dieses Statement direkt geloggt werden.

```
reward = tf.Summary(value=[tf.Summary.Value(tag="Reward", simple_value=l)])
```

Das Ergebnis der Logoperationen wird gespeichert:

```
writer.add_summary(reward, iteration) writer.add_summary(summary, iteration)
```

```
In [5]: import gym
        import numpy as np
        import random
        import tensorflow as tf
```

```
env = gym.make('Taxi-v2')

tf.reset_default_graph()

#These lines establish the feed-forward part of the network
# used to choose actions
inputs1 = tf.placeholder(shape=[1,500],dtype=tf.float32)
w = tf.Variable(tf.random_uniform([500,6],0,0.01))
qout = tf.matmul(inputs1,w)
predict = tf.argmax(qout,1)

#Below we obtain the loss by taking the sum of squares difference
# between the target and prediction Q values.
nextq = tf.placeholder(shape=[1,6],dtype=tf.float32)
loss = tf.reduce_sum(tf.square(nextq - qout))
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
update_model = trainer.minimize(loss)

#Log loss value
tf.summary.scalar("Loss", loss)
#Define additional summaries here

# merge all summaries into a single "operation"
# which we can execute in a session
summary_op = tf.summary.merge_all()

init = tf.global_variables_initializer()

#Set learning parameters
y = .99
e = 0.1
num_episodes = 10000

iteration = 0

with tf.Session() as sess:
    sess.run(init)

    # create log writer object
    writer = tf.summary.FileWriter("/mnt/notebooks/logs",
                                   graph=tf.get_default_graph())

    for i in range(num_episodes):
        #Reset environment and get first new observation
        s = env.reset()
        r_all = 0
        d = False
        j = 0
```

```
                #The Q-Network
                while j < 99:
                    j+=1
                    l=0
                    #Choose an action by greedily (with e chance of random action)
                    # from the Q-network
                    a,allq = sess.run([predict,qout],
                                      feed_dict={inputs1:np.identity(500)[s:s+1]})
                    if np.random.rand(1) < e:
                        a[0] = env.action_space.sample()
                    #Get new state and reward from environment
                    s1,r,d,_ = env.step(a[0])
                    #Obtain the Q' values by feeding the new state through our network
                    q1 = sess.run(qout,feed_dict={inputs1:np.identity(500)[s1:s1+1]})
                    #Obtain maxQ' and set our target value for chosen action.
                    maxq1 = np.max(q1)
                    targetq = allq
                    targetq[0,a[0]] = r + y*maxq1
                    #Train our network using target and predicted Q values
                    _,w1,l,summary = sess.run([update_model,w,loss,summary_op],
                                              feed_dict={inputs1:np.identity(500)[s:s+1],
                                                         nextq:targetq})

                    reward = tf.Summary(value=[tf.Summary.Value(tag="Reward",
                                                                simple_value=l)])

                    # write log
                    writer.add_summary(reward, iteration)
                    writer.add_summary(summary, iteration)
                    iteration += 1

                    s = s1
                    if d == True:
                        #Reduce chance of random action as we train the model.
                        e = 1./((i/50) + 10)
                        break
```

```
[2017-10-17 16:06:39,737] Making new env: Taxi-v2
```

Blackjack-v0

December 17, 2017

1 Vorstudie 3: Blackjack

Zweck Dieses Experiment gibt einen Einblick in die Verwendung von Keras.

Beschreibung Das gelöste Problem ist das Spiel Blackjack aus der Sammlung des OpenAI Gym. Die Regeln und auch das Ziel des Spieles stammt vom allgemein bekannten Karten-Glücksspiel Blackjack. Um zu gewinnen muss der Spieler mit den erhaltenen Karten so genau wie möglich die Punktzahl 21 erreichen, darf jedoch nicht höher sein als dieser Wert. Gleichzeitig muss er aber eine höhere Summe besitzen als die Bank, sprich der Dealer.

Die Grundlage der nachstehenden Lösung stammt von Simen Eide. Für die Analyse wurde diese jedoch optimiert und für die aktuellen Versionen von Python und Keras aktualisiert.

Quellen Spielumgebung mit Beschreibung
https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py
Originale Lösung
https://gist.github.com/simeneide/f3d33868c8fc36ed3934da4aa71847f0
Kerasspezifische Erklärungen stützen sich auf der Beschreibung der Keras API
https://keras.io

1.1 Lösung

```
In [8]: import gym
        import numpy as np
        import random
        import tensorflow as tf
        import keras
        keras.backend.backend()
        from keras.models import Sequential
        from keras.layers import Dense, Activation
        from keras.regularizers import l2
        from keras.optimizers import SGD
        from keras.utils import np_utils
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Nach dem Importieren aller notwendigen Packages und Modulen wird die Umgebung (Environment) des Spieles geladen.

```
In [9]: env = gym.make('Blackjack-v0')

[2017-10-25 07:36:29,645] Making new env: Blackjack-v0
```

1.1.1 Funktion des Spielablauf

Zu Beginn wird die Umgebung mit env.reset() zurückgesetzt also ein leerer Tisch erstellt und die Startkarten verteilt. Daraufhin wird die Aktion (action) mittels der übergebenen Funktion (policy) bestimmt. Dessen Ergebnis kommt bei der Funktion step() der Umgebung zum Einsatz, welche wiederum die folgenden vier Rückgabeparameter liefert:

- **observation (object):** Der aktuelle Spielstatus beinhaltet die Informationen über die aktuelle Punktezahl des Spielers, die Karten des Dealers und ob der Spieler ein Ass mit dem Punktewert als '1' (False) oder '11' (True) einsetzt.
- **reward (float):** Die Belohnung (reward) gibt den Wert '+1' für einen Sieg des Spielers, '0' für ein Unentschieden und '-1' für ein Verlieren des Spielers zurück.
- **done (boolean):** Wird der Punktewert 21 überschritten oder entscheidet sich der Spieler keine Karte vom Dealer mehr zu beziehen wird der boolesche Wert 'True' zurückgegeben ansonsten 'False'. Damit wird der Spielzug beendet.
- **info (dict):** Dieser Parameter ist in diesem Spiel immer leer, aber wird sonst verwendet um zusätzliche Informationen zur Umgebung mitzuteilen, welche beispielsweise bei der Fehlersuche hilfeich sein können.

Die oben aufgeführte Beschreibung ist spezifisch für das hier verwendete Spiel Blackjack gemäss der Beschreibung der Spielumgebung.

```
In [10]: def play_game(policy):
             obs = env.reset()
             for _ in range(10):
                 action = policy(obs)
                 obs, reward, done, info = env.step(action)
                 if done:
                     # print('Our hand: {} \t Dealer: {} \t
                     #  reward: {}'.format(obs[0], obs[1], reward))
                     break
             return obs, action, reward
```

1.1.2 Hilfsfunktionen

- **q_policy(obs):** Anhand eines zufällig generierten Wertes entscheidet sich, ob die nächste Aktion zufällig oder vom neuronalen Netzwerk entschieden wird.
- **proc_input(obs_list, action_list):** Damit werden die Daten für das Gradientenupdate des Modells vorbereitet.

```
In [11]: # Using Linear Regression
         def proc_input(obs_list, action_list):
             x = np.zeros((obs_list.shape[0], 4))
             x[:, :-1] = obs_list
```

```
        x[:, 3] = action_list
        return x

    def q_policy(obs):
        if random.random() < ep:
            return int(random.random() > 0)
        else:
            x = np.zeros([2, 4])
            x[:, 0:3] = obs
            x[:, 3] = [0, 1]
            q = q_model.predict(x)
            return q.argmax()
```

1.1.3 Netwerk erstellen

Mit Keras ist das Erstellen des Netzwerks einfacher als direkt mit der Programmbibliothek TensorFlow. Bei der Erstellung des Netzwerks wird angegeben um welche Art von Netzwerk es sich handelt und wie dieses im Detail (Schichten, Funktionen) aufgebaut ist.

Aufbau des neuronalen Netzwerk Das hier verwendete Netzwerk ist sequentiell aufgebaut und besteht aus einer Eingangsschicht (Input Layer) mit 4 Eingängen, einer versteckten Schicht (Hidden Layer) mit 4 Neuronen und einer Ausgangsschicht (Output Layer) mit 1 Ausgang. Die hintereinanderliegenden Schichten sind jeweils vollvernetzt (fully connected). Die Neuronen der versteckten Schicht besitzen die Aktivierungsfunktion ReLU. Die Ausgabeschicht besitzt nur ein Neuron, woraus erkennbar ist, dass hier nicht nach einer Klassifizierung sondern nach einer realen Zahl gefragt wird.

Lernprozess Der Lernprozess besteht aus einer Verlustfunktion (loss), einem Optimierer (optimizer) und einer Liste von Metriken (metrics). Das Netzwerk wird gleich nach dem Definieren der Parameter kompiliert. Im untenstehenden Beispiel wurde für die Verlustfunktion sowie auch für die Metrik die Funktion des mittleren quadratischen Fehlers verwendet und als Optimierungsfunktion kommt die Methode Stochastic Gradient Descent (SGD) mit einer definierten Learnrate (lr) zum Einsatz. Obwohl diese drei Parameter schlussendlich zusammenspielen müssen, kann für jeden einzelnen entschieden werden welche Funktion bzw. Methode dafür verwendet wird. Für den Optimierer wird hier beispielsweise, wie so oft als erste Wahl, die Methode sgd (stochastic gradient descent) eingesetzt, was jedoch nicht zwangsweise die optimalste Wahl ist, da sich auch die Methode Adam anbieten würde und oftmals sogar bessere Resultate liefert. Die genauen Beschreibungen und Auswahlmöglichkeiten können der Keras Online Dokumentation (https://keras.io) entnommen werden.

```
In [12]: # Define q-model:
         q_model = Sequential()
         q_model.add(Dense(4, input_shape=(4,), kernel_initializer='uniform'))
         q_model.add(keras.layers.normalization.BatchNormalization())
         q_model.add(Activation("relu"))
         q_model.add(Dense(1, kernel_regularizer=l2(0.01)))
         sgd = SGD(lr=0.005)
```

```
q_model.compile(loss='mean_squared_error', optimizer=sgd,
                metrics=['mean_squared_error'])
```

1.1.4 Neuronales Netzwerk trainieren

Um das neuronale Netzwerk optimal zu trainieren, müssen mehrere Episoden mit mehreren Spielrunden durchgeführt werden, damit das Netzwerk fürs Training genügend Runden und deren Auswertungen zur Verfügung hat. In den einzelnen Spielrunden wird das Verhalten des aktuellen Netzwerk verwendet um zu spielen und möglichst zu gewinnen. Am Ende jeder Episode wird das neuronale Netzwerk mit den aufgezeichneten Spielrunden inklusive Auswertung trainiert(train_on_batch()). Dadurch lernt das Netwerk von den alten Episoden und kann die Gewichte der Neuronenverbindungen der aktuellen Auswertung anpassen.

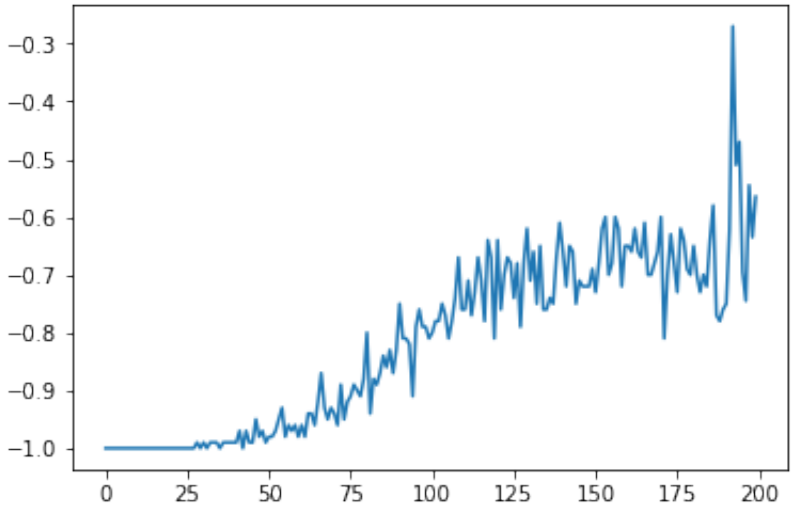
In [13]:
```
ep = 0.1

score = []
num_episodes = 200
num_rounds = 200
for episode in range(num_episodes):
    obs_list = []
    r = []
    action_list = []
    for k in range(num_rounds):
        obs, action, reward = play_game(q_policy)
        obs_list.append(obs)
        action_list.append(action)
        r.append(int(reward))
    score.append(np.mean(r))

    if (episode > 50) & (np.max(score) > 0.92) & (ep > 0.01):
        ep = 0.01
        print('Entering low-epsilon mode at episode {}'.format(episode))
    if (episode > 50) & (np.max(score) > 0.97) & (ep > 0):
        ep = 0.0
        print('Entering greedy-mode at episode {}'.format(episode))

    x = proc_input(np.array(obs_list), action_list)
    y = np.array(r)
    q_model.train_on_batch(x, y)
```

1.1.5 Auswertung

Das folgende Diagramm zeigt grafisch den Lernfortschritt und -erfolg über die gespielten Episoden an.

In [14]:
```
plt.plot(score)
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x7ffac4ce1a90>]
```


−0.3
−0.4
−0.5
−0.6
−0.7
−0.8
−0.9
−1.0
0
25
50
75
100
125
150
175
200

Anleitung für Installation und Betrieb der Test- und Entwicklungs-umgebung auf der HSR Infrastruktur und für den lokalen Betrieb des JassBot Prototyps

Inhaltsverzeichnis

Benötigte Installationen für die Entwicklungsumgebung an der HSR

Der virtuelle Server [Ubuntu 16.04.3 TLS], welcher die HSR zur Verfügung stellt, wird vom Projektteam als Test- und Entwicklungsumgebung verwendet. Die extrem rechenintensiven Prozesse der ML-Clients werden auf dem Nvidia DGX Server des ICOM betrieben.

Zu installierende und betreibende Systeme:

* Jass-Challenge-Server von Zühlke (Docker Container)

* Jupyter-Notebook mit TensorFlow, TensorBoard, OpenAI Gym und Keras (Docker Container)

* Docker

Docker wird als Grundlage für die einfachere Betreibung und Verwendung der Serverimages benutzt.

Die Docker-Container werden über die freien Pots des Hosts «40000 – 40010» veröffentlicht.

Installation von Docker

Docker besitzt seit 2017 zwei unterschiedliche Versionen. Dies sind Docker Enterprise Edition (EE) und Docker Community Edition (CE). Da es sich bei unserer Installation um eine Testumgebung für eine Bachelorarbeit handelt, reicht uns das Docker CE.

Falls der Installationsschritt nicht auf Anhieb funktioniert, müssen zuerst die folgenden vorbereitenden Massnahmen getroffen werden. Diese und auch deren detaillierte Erklärungen sind in den Dokumentationen von Docker zu finden. [1]

Vorbereitende Arbeiten

Erlauben eines Repository über HTTPS zu nutzen.

```
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
```

Hinzufügen des offiziellen GPG Schlüssel von Docker.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Mit dem folgenden Befehl wird das **stable** Repository gesetzt.

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu xenial stable"
```

Nach erneutem Update des Package-Index kann nach dem gewünschten Docker-Packet gesucht werden.

```
sudo apt-get update

apt-cache search docker-ce
```

Erscheint die folgende Zeile, dann ist das Docker-Package verfügbar zur Installation:

```
docker-ce - Docker: the open-source application container engine
```

Installation

Für die Installation ist lediglich zu beachten, dass für diese Testumgebung Docker CE eingesetzt wird.

Update des Package-Index:

```
sudo apt-get update
```

Installation vom Software-Package Docker CE:

```
sudo apt-get install docker-ce
```

[1] https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/

Installation des Zühlke-Jass-Challenge-Server

Bezug der Software

Erstellen des lokalen git-Verzeichnis für die Server-Software und in das Verzeichnis wechseln.

```
sudo mkdir /home/hsr/git/jass-challenge-server

cd /home/hsr/git/jass-challenge-server
```

Clonen des Jass-Challenge-Server von Zühlke

```
git clone https://github.com/webplatformz/challenge.git
```

Dockerfile bearbeiten

Das Dockerfile von Zühlke ist grundsätzlich bereits vorbereitet für eine Erstellung eines Docker Containers falls jedoch noch zusätzliche Parameter wie beispielsweise das exportieren von Transaktionen verwendet werden sollen, dann ist eine Bearbeitung des Dockerfiles noch noch notwendig.

```
cd challenge

nano Dockerfile
```

Parameter-Beschreibungen zu den Environment-Variablen sind im Git-Hub-Repository des Zühlke-Jass-Challenge-Server ersichtlich.

Zum Beispiel kann beim aktivieren des Parameters «TOURNAMENT_LOGGING» mittels «true» alle Transaktionen in das Verzeichnis der Installation auf dem Container geschrieben werden. [2]

[2] https://github.com/webplatformz/challenge

```
  GNU nano 2.5  File: Dockerfile

FROM node

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

ENV TOURNAMENT_ROUNDS 5
ENV TOURNAMENT_COUNTING true
ENV TOURNAMENT_LOGGING true

EXPOSE 3000
CMD [ "npm", "start" ]
```

Erstellen eines Docker-Container. Mit dem «.» wird das aktuelle Verzeichnis angegeben. An dieser Stelle des «.» kann auch das geklonte REPO-Verzeichnis angegeben werden.

Container erstellen und starten

Erstellen eines Docker Image vom Jass-Challenge-Server.

```
cd challenge

sudo docker build . -t jasschallenge
```

Starten des Containers mit der Veröffentlichung der Applikation auf einem von extern zugänglichen Port 40001. (Port-Mapping: <Hostport>:<DockerContainerPort>)

```
sudo docker run -d --rm --name cs -p 40001:3000 jasschallenge
```

Exportieren und Bearbeiten der Transaktionen

Um die Transaktionen der Spiele vom Container auf den Host zu exportieren, können die folgenden Befehel verwendet werden.

Kopieren von Dateien aus dem Container auf den Host:

```
sudo docker cp cs:/usr/src/app /home/hsr/data/generate-jcs/
```

Die benötigten Dateien mit den Transaktionen haben die folgende Form:

```
A vs B.1507297185788.json
```

Wechseln ins Verzeichnis der Dateien und rausfiltern der zu analysierenden Dateien:

```
cd /home/hsr/data/generate-jcs/app
```

```
rm -rf !(*vs*.json)
```

Liste der gefilterten Dateien sortiert nach Änderungsdatum:

```
ls -l -S
```

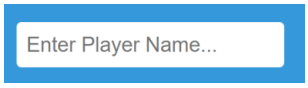
Output:

```
-rw-r--r-- 1 root root 20333 Oct  6 20:17 A vs B.1507297185788.json

-rw-r--r-- 1 root root 20310 Oct  6 20:55 A vs B.1507313906499.json
```

Webzugriff auf das Interface des Jass-Challenge-Server

Mittels eines Webbrowsers über den Link **«http://152.96.56.46:40001/»**
oder **«http://sinv-56046.edu.hsr.ch:40001/»** ist das Interface des Jass-Server erreichbar.

Startbildschirm:



Spieltisch nach einer gespielten Runde:

Testspiel auf dem Zühlke-Jass-Challenge-Server mit vordefinierten Bots

Der Challenge-Server liefert Random-Jass-Bots für Tests mit, welche wie folgt gestartet werden können.

Einstieg in den laufenden Container des «Zühlke-Challenge-Jass-Server» um die Bots zu starten.

```
sudo docker exec -it cs bash

npm run bot:start
```

Um ein Tournament zu starten sollt gemäss der Anleitung unter dem Link
https://github.com/webplatformz/challenge-client-java vorgegangen werden.

Installation des Jupyter Notebook Container

Bei dem Verwendeten Juypter Notebook Container handelt es sich um einen öffentlich zugänglichen Container auf Docker Hub,[3] welcher unter der Bezeichnung mimoralea/openai-gym[4] zu finden ist. Das spezielle an diesem Container sind die vorhandenen Software-Pakete, Python2 und Python3, OpenAI Gym, TensorFlow 1.00 mit TensorBoard, Keras und der webbasierten IDE «Jupyter Notebook».

Nach der Installation von Docker ist diejenige eines Containers von Docker Hub ziemlich simpel.

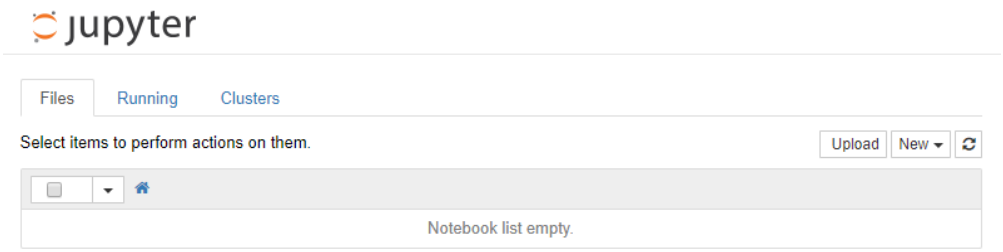
```
sudo docker pull mimoralea/openai-gym
```

Erstellen und starten des Container auf den Host-Ports «40007» und «40008».

```
sudo docker run -d -p 40008:8888 -p 40007:6006 -v /home/hsr/data/notebooks:/mnt/notebooks mimoralea/openai-gym:v1
```

Zugriff auf das Jupyter Notebook

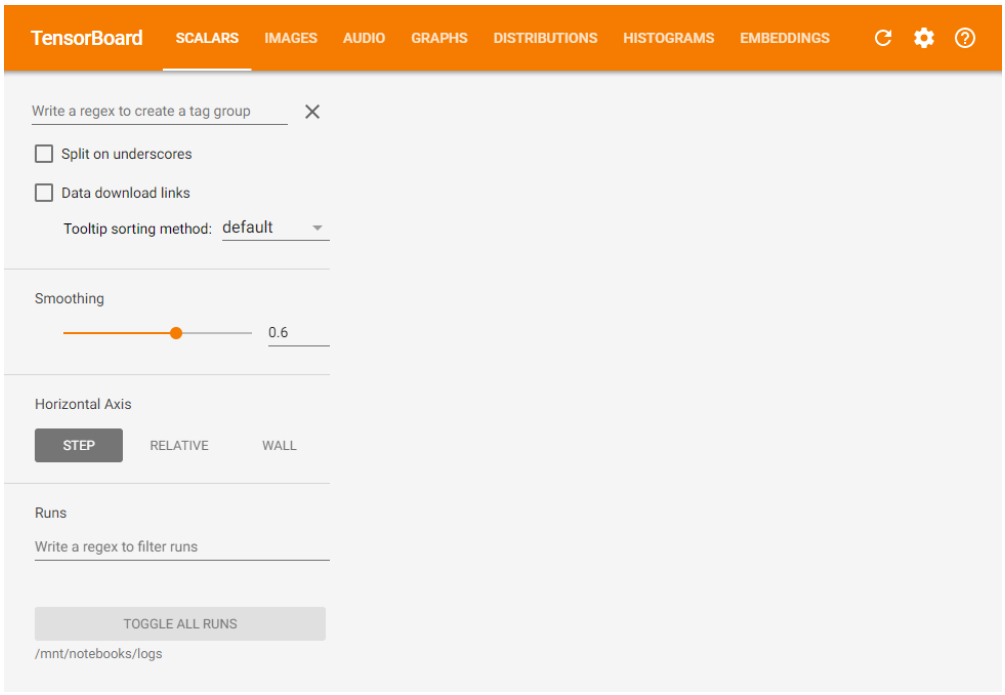
Mit dem Webbrowser via dem Link **«http://152.96.56.46:40008/»**
oder **«http://sinv-56046.edu.hsr.ch:40008/»** kann das Jupyter Notebook auf dem HSR Testserver aufgerufen und bearbeitet werden.



[3] https://hub.docker.com/
[4] https://hub.docker.com/r/mimoralea/openai-gym/

Zugriff auf das TensorBoard

Über den Link **«http://152.96.56.46:40007/»** oder **«http://sinv-56046.edu.hsr.ch:40007/»** kann das TensorBoard angezeigt werden.



TensorBoard manuell starten

Falls TensorBoard nicht automatisch startet oder nach der Installation von TensorFlow auf dem lokalen Computer gestartet werden möchte, kann mit folgendem Befehl TensorBoard gestartet werden.

```
tensorborad --logdir=<log_dir_path>
```

Nachdem TensorBoard gestartet ist, können die Auswertungen unter **http://localhost:6006/** betrachtet werden.

Installation des JavaBots

Bezug der Software

Erstellen des lokalen git-Verzeichnis für die JassBot-Software und in das Verzeichnis wechseln.

```
sudo mkdir /home/hsr/git/

cd /home/hsr/git/
```

Clonen des JassBots

```
git clone https://bitbucket.org/Zeglect/jasschallenge-client.git
```

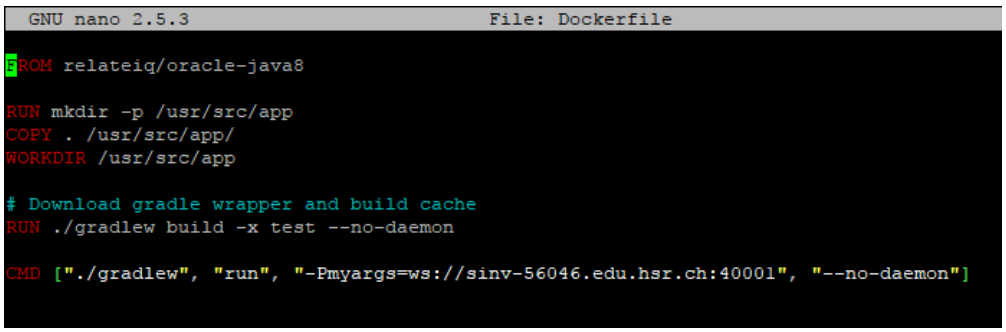
Dockerfile bearbeiten

In der aktuellen Version auf dem Repository sind zwei Junit Tests nicht erfolgreich. Da dies ein builden des Clients verhindert, müssen die Tests vor dem Build im Dockerfile deaktiviert werden.

```
cd jasschallenge-client

nano Dockerfile
```

Für die Testdeaktivierung muss die Zeile "**RUN ./gradlew build --no-daemon**" durch den Befehl "**RUN ./gradlew build -x test --no-daemon**" ersetzt werden.

Ausserdem muss je nachdem wo der Server betrieben wird zusätzlich in der letzten Zeile die Adresse und eventuell der Port des Servers angepasst werden (Standardmässig: **ws://cs:3000**).



Container erstellen und starten

Erstellen eines Docker Image vom Jass-Challenge-Server.

```
cd /home/hsr/git/jasschallenge-client

sudo docker build . -t jassclient
```

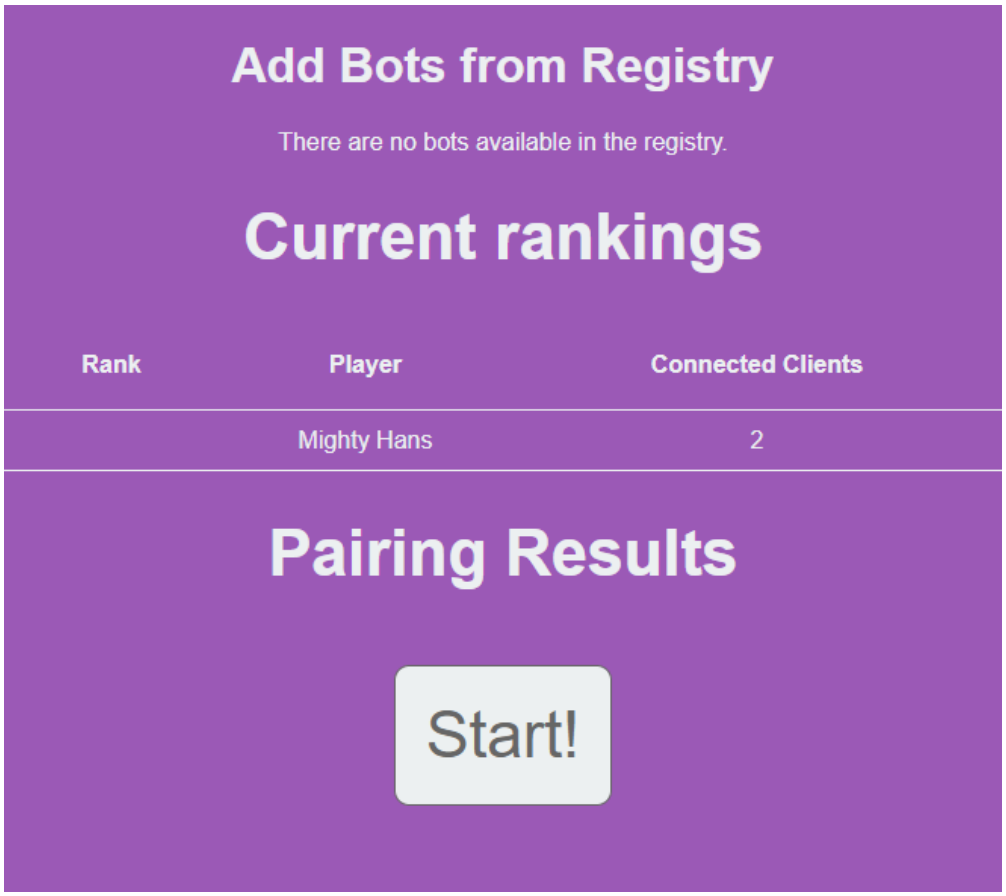
Mit dem «.» wird das aktuelle Verzeichnis angegeben. An der Stelle des «.» kann auch das geklonte REPO-Verzeichnis angegeben werden.

Starten des Containers

```
sudo docker run -it jassclient
```

Mehrere JavaBots betreiben

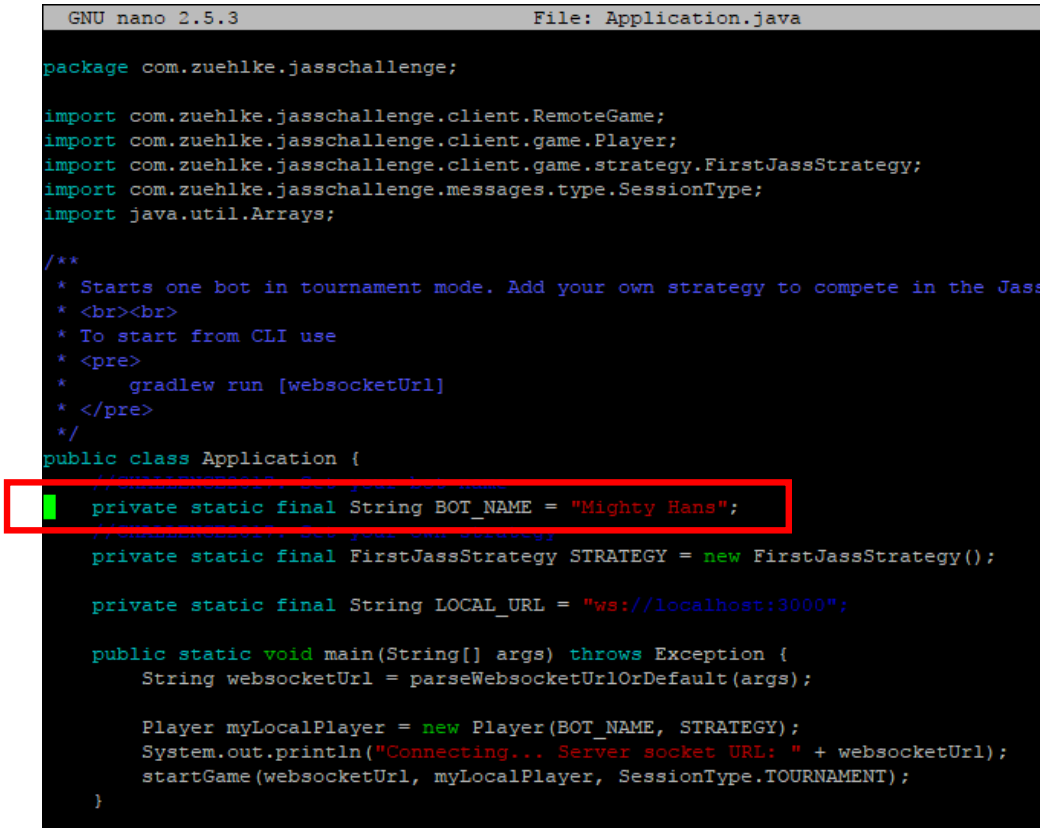
Es können maximal zwei Instanzen des JassBots mit dem gleichen Spielernamen gestartet werden. Beim Starten von mehreren Bots gemäss «Docker-RUN»-Befehl von vorherigem Abschnitt, wird einfach einer wieder beendet.



Der Spielername des JassBot kann im Verzeichnis «/home/hsr/git/jasschallenge-client/src/main/java/com/zuehlke/jasschallenge» in der Datei «Applikation.java» geändert werden.

```
cd /home/hsr/git/jasschallenge-client/src/main/java/com/zuehlke/jasschallenge

nano Application.java
```

Hier kann nun die statische final Variable «BOT_NAME» angepasst werden.



Anschliessend kann erneut ein neues Docker-Image erstellt und gestartet werden wie im vorherigen Abschnitt.

```
cd /home/hsr/git/jasschallenge-client

sudo docker build . -t <NEUER IMAGE NAME>
```

<NEUER IMAGE NAME> = Neuer Name des zu erstellenden Docker-Image, damit nicht das alte Image überschrieben wird.

Wenn nun so zwei Images mit unterschiedlichem Spielernamen existieren können von jedem Image zwei Instanzen gestartet werden.

Starten eines Docker-Containers

```
sudo docker run -it <NAME DES DOCKER-IMAGE>
```

Ausstieg aus einem interaktiven Container erfolgt mit der Tastenkombination und Reihenfolge «Ctl + P» und «Ctl + Q».

Installation des Machine Learning Jassbots

Diese Anleitung beschreibt wie der Python-basierte JassBot installiert und trainiert werden kann.

Installationsdateien beziehen

Es gibt zwei Arten die benötigten Dateien zu beziehen, welche im Folgenden genauer erläutert werden.

GitHub

Der JassBot ist in einem öffentlichen Repository[5] auf GitHub verfügbar. Die bereits trainierten Netzwerke sind in einem privaten Repository[6] nur für berechtigte Personen zugänglich. Es kann aber auch ohne diese Dateien nur mit dem JassBot ein eigenes Netzwerk trainiert werden.
Der aktuelle Git-Client kann unter folgendem Link bezogen werden: https://git-scm.com/download/

Clone via CLI

Kommandozeileneingabe starten und in das gewünschte Verzeichnis wechseln.

```
git clone https://github.com/scriptcoffee/JassBot.git JassBot

cd JassBot

git checkout rl-v0.3
```

Danach sollte im JassBot-Verzeichnis noch die Unterordner für die Logs erstellt werden:

```
mkdir logs

cd logs

mkdir config
```

Zip-Datei

Für den JassBot kann aus der abgegebenen Zip-Datei der Ordner **rl-v0.3** im Verzeichnis **Sourcecode/ReinforcementLearning/** extrahiert werden.

Installation

Es folgt die Installation von Python 3 und dem Python-basierten JassBot.

Ubuntu

```
apt-get update

apt-get install python3 python3-pip

pip3 install -r requirements.txt
```

Windows

Die aktuellste Python 3.6 Version für 64Bit kann unter folgender URL heruntergeladen werden:
https://www.python.org/downloads/windows/

[5] https://github.com/scriptcoffee/JassBot
[6] https://github.com/scriptcoffee/BA-Data

Bei der anschliessenden Installation sollte darauf geachtet werden, dass die Option **Add Python 3.6 to PATH** bereits zu Beginn angewählt wird. Danach kann die Installation mit den Standartparametern durchgeführt werden.

Das Microsoft Visual C++ 2015 Redistributable Update 3 kann unter folgendem Link heruntergeladen und anschliessend installiert werden:
https://www.microsoft.com/en-us/download/details.aspx?id=53587

Nun sollte auf der Kommandozeile im JassBot-Verzeichnis noch der folgende Befehl ausgeführt werden:

```
pip install -r requirements.txt
```

Falls der Befehl **pip** auf der Kommandozeileneingabe einen Fehler verursacht, dann muss das pip-Tool noch gemäss der Installation unter folgendem Link installiert werden:
https://pip.pypa.io/en/stable/installing/#installing-with-get-pip-py

Anschliessend muss nochmals der **pip install** Befehl ausgeführt werden, wie oben erwähnt.

Vortrainierte Netzwerke (privates Repository)

Für die Nutzung eines vortrainierten Netzwerkes und bei Zugriff auf das private Repository https://github.com/scriptcoffee/BA-Data, kann hier weitergelesen werden ansonsten sollte mit dem nächsten Kapitel «Verbindungsparamter anpassen» fortgesetzt werden.

Um bereits trainierte Netzwerke laden zu können, muss ein anderer Branch ausgecheckt werden. Dazu kann im JassBot Verzeichnis der folgende Befehl ausgeführt werden.

```
git checkout play
```

Die Modelle können auf die folgenden zwei Arten bezogen werden:

- In der Zip-Datei können die aktuellen Netzwerke unter ./**Daten/aktuelle Modelle** gefunden und von dort kopiert werden.
- Auf GitHub[7]sind ebenfalls alle Netzwerke abgelegt. Dort kann unter **TrainingLogs/** die Zip-Datei des gewünschten Netzwerkes heruntergeladen werden. Die Modelle befinden sich im **config** Ordner und tragen die Dateiendung h5

Im JassBot-Verzeichnis muss der Ordner «models» erstellt werden.

Die vortrainierten Netzwerke müssen anschliessend an den folgenden Ort kopiert werden:

```
models/trumpf_network_model.h5
models/game_network_model.h5
```

Dabei ist zu beachten, dass die Namen mit den oben genannten übereinstimmen.

Verbindungsparameter anpassen

In der Konfigurationsdatei `elbotto/tournament.py` können alle relevanten Parameter, wie die Server-URL, der Name, die Anzahl und der Typ der zu startenden Bot konfiguriert werden.

[7] https://github.com/scriptcoffee/BA-Data

Server URL anpassen

Standardmässig ist die URL des Servers auf localhost und den Standard Port 3000 gesetzt. Dies kann mit der Konstante **DEFAULT_SERVER_NAME** geändert werden.

```
DEFAULT_SERVER_NAME = "ws://127.0.0.1:3000"
```

Bots konfigurieren

Die Funktion **start_bots()** definiert wie viele und von welcher Art Bots gestartet werden. Standardmässig werden 3 Bots, welche zufällig Karten spielen, und ein Reinforcement Learning Bot, ohne vortrainiertes Netzwerk, gestartet.

Bots kompilieren und starten

Mit der Kommandozeile im JassBot- Verzeichnis die folgenden Befehle ausführen:

Ubuntu

```
mkdir logs

mkdir logs/config

python3 setup.py install

python3 elbotto/tournament.py
```

Windows

```
mkdir logs

cd logs

mkdir config

cd ..

python setup.py install

python elbotto/tournament.py
```

Zu beachten

TensorFlow und Threads

Der JassBot bietet die Möglichkeit gleichzeitig mehrere Bots in separaten Threads zu starten. Es ist jedoch nicht möglich mehrere Netzwerke gleichzeitig in unterschiedlichen Threads laufen zu lassen, da TensorFlow in dieser Konfiguration nicht stabil läuft. Sollten mehre Netzwerke gleichzeitig trainiert werden, müssen diese separat in unterschiedlichen Prozessen gestartet werden.

Manuelle Überprüfung von trainierten Netzwerken

Python-IDE

Um die Qualität von trainierten Netzwerken manuell zu überprüfen, wird empfohlen mit einer Python-IDE zu arbeiten. Falls noch keine installiert ist, so wird die frei verfügbare Version von PyCharm empfohlen, welche unter dem folgenden Link heruntergeladen und installiert werden kann. Es sollte jedoch zuvor die Installation von Python und dem requirements.txt aus dem vorherigen Unterkapitel

«Installation des Machine Learning Jassbots» durchgeführt werden.
https://www.jetbrains.com/pycharm-edu/

Bezug der manuellen Testdateien

Die manuellen Testdateien können wie die Installationsdateien im vorherigen Unterkapitel «Installationsdateien beziehen» via GitHub oder Zip-Datei bezogen werden.

Via **GitHub** muss anstatt der Branch **rl-v0.3** der Branch **sl-v0.3** bezogen werden, was bedeutet, dass die letzte Zeile unter «Clone via CLI» durch den folgenden Befehl ersetzen werden muss:

```
git checkout sl-v0.3
```

In der abgegebenen Zip-Datei entspricht dies dem Ordner **sl-v0.3**, welcher im Verzeichnis **Sourcecode/SupervisedLearning/** zu finden ist und extrahiert werden kann.

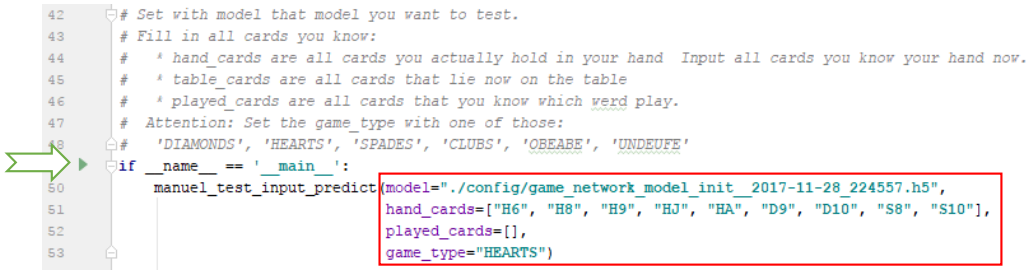
Öffnen der manuellen Tests

Starte die Python-IDE (z.B.: PyCharm) und öffne in der Startansicht oder über das Menü File → Open…, das JassBot-Projekt.

In der Projektstruktur «JassBot/elbotto/bots/training» befinden sich die zwei Dateien «manual_game_testing.py» und «manual_trumpf_testing.py» für die manuellen Tests.

Konfigurieren und starten der manuellen Tests für das «Gamenetzwerk»

Beim Öffnen der Datei «manual_game_testing.py» in der der Python-IDE (PyCharm), ist folgendes am Dateiende zu sehen:



Die Parameter im roten Rahmen sind oberhalb beschrieben und müssen von der Testperson manuell eingetragen und aktualisiert werden.
Um den Test zu starten und das ermittelte Ergebnis des Netzwerkes zu sehen, muss mit der Maus auf den grünen Run/Play-Button neben dem grünen Pfeil geklickt werden, wodurch das Ergebnis unterhalb angezeigt wird, wie in der folgenden Abbildung.

```
Follow you see the list of predictions:
 [[ 0.03009833  0.02967684  0.02812863  0.02474031  0.02543324  0.02716203
   0.02617721  0.02981923  0.02774704  0.02882694  0.02634907  0.02908394
   0.02893301  0.03059877  0.02772763  0.03110001  0.02676195  0.0249594
   0.02763025  0.02995488  0.03136003  0.02992567  0.02855938  0.02703219
   0.02471439  0.02750833  0.02935628  0.02585383  0.02771118  0.02501753
   0.02629865  0.02586791  0.02656069  0.02603494  0.02925063  0.0280397 ]]

The model predict the card 8 from CLUBS with a probability of 0.03136002644896507 as the best one.
```

Konfigurieren und starten der manuellen Tests für das «Trumpfnetzwerk»

Beim Öffnen der Datei «manual_trumpf_testing.py» in der der Python-IDE (PyCharm), ist folgendes am Dateiende zu sehen:

```
35
36      # Input your hand cards into the list and set 'pushed' to True if your partner moved the trump decision to you.
37      # Set with model that model you want to test.
38      if __name__ == '__main__':
39          manuel_test_input_predict(model="./config/trumpf_network_model__2017-12-04_180518.h5",
40                                    hand_cards=["H6", "H8", "H9", "HJ", "HA", "D9", "D10", "S8", "S10"],
41                                    pushed=False)
42
```

Die Parameter im roten Rahmen sind oberhalb beschrieben und müssen von der Testperson manuell eingetragen und aktualisiert werden.
Um den Test zu starten und das ermittelte Ergebnis des Netzwerkes zu sehen, muss mit der Maus auf den grünen Run/Play-Button neben dem grünen Pfeil geklickt werden, wodurch das Ergebnis unterhalb angezeigt wird und wie in der folgenden Abbildung dargestellt wird.

```
The prediction is:
 hearts: 0.5183934569358826
 diamonds: 0.06865750253200531
 clubs: 0.011051014997065067
 spades: 0.03648940846323967
 OBEABE: 0.045069266110658646
 UNDEUFE: 0.06094620004296303
 SCHIEBE: 0.2593930959701538
```

Angaben der erlaubten Karten

Die Karten sind wie gewohnt durchnummeriert von **6 bis 10**, danach folgt ein **J** für den Junker, **Q** für die Queen, **K** für King und **A** für das Ass. Des Weiteren ist zu beachten, dass alle Angaben für die manuellen Tests nach der englischen Bezeichnung des französischen Blattes zu machen sind. Dies bedeutet folgendes:

Geforderte Eingabe	Englische Bezeichnung	Französisches Blatt	Schweizer Blatt
H	Hearts	Herz	Rosen
D	Diamonds	Karo/Ecke	Eichel
C	Clubs	Kreuz	Schellen
S	Spades	Pik/Schaufeln	Schilten