InformatiCup2020: Dokumentation

Team Informatik ohne Grenzen

Markus Gersdorf, Paul Gronau,
Torben Logemann, Marcel Peplies

Wintersemester 2019/20

Pandemie

Stand:
19. Januar 2020

Zusammenfassung

In diesem Projekt wurde im Rahmen des 'InformatiCup 2020' der Gesellschaft für Informatik ein neuronales Netz implementiert, das als verteiltes System trainiert. Aufgrund von zeitlichen Einschränkungen in der Trainingszeit ist die Performance des neuronalen Netzes eingeschränkt und müsste in weiteren Schritten optimiert werden. Die Aufgabe war das Erstellen einer Software, die in einer bereitgestellten Simulation rundenbasiert Aktionen ausführt, die die Menschheit vor einer Pandemie rettet. Dafür wurde zunächst das gegebene Kommandozeilenwerkzeug (ic20 Tool) in Bezug auf die Kommunikation mit dem Pythonprogramm und den Ausgaben des Kommandozeilenwerkzeugs analysiert. Anschließend wurden mögliche Lösungsstrategien jeweils in ihrer Funktionsweise samt Struktur erklärt und mit ihren Vor- und Nachteilen diskutiert. Auf dieser Grundlage wurde die Entscheidung getroffen, ein neuronales Netz als Lösungsstrategie zu implementieren. Dazu wird die Logik des Netzes im Zusammenspiel mit ihren Komponenten erläutert. Für die Vollständigkeit sind Begriffe und Zugehörigkeiten im Rahmen von neuronalen Netzen in einem Exkurs erklärt. Für die Entwicklung des Netzes wurden bewährte Technologien wie die Containerisierungssoftware 'Docker' sowie die Arbeitsablaufgestaltung GitHub-Actions verwendet. Die Verteilung der Software fand auf von der Carl von Ossietzky Universität Oldenburg zur Verfügung gestellten Servern statt. Damit konnte die Strategie des verteilten Lernens für das neuronale Netz umgesetzt werden. Graphisch dargestellt werden der Verlauf der Krankheitsausbreitung sowie die Resultate der Entscheidungen mithilfe der Google-Maps-API. Am Ende der Dokumentation werden die Umsetzung der Tests sowie die Arbeitsorganisation aufgegriffen.

Abstract

In this project a neural network was implemented in the context of the 'InformatiCup 2020' (Gesellschaft für Informatik) which trains as a distributed system. Due to time restrictions in the training time, the performance of the neural network is limited and would have to be optimized in further steps. The task was to create a software, which executes actions in a provided round based simulation which saves humanity from a pandemic. For this purpose, the given command line tool (ic20_Tool) was first analyzed with regard to communication with the Python program and the output of the command line tool. Subsequently, possible solution strategies were explained in their function and structure, and their advantages and disadvantages were discussed, which led to the decision to implement a neural network as a solution strategy. The logic of the network in interaction with its components is explained. Terms and affiliations within the framework of neural networks are explained in an excursus. For the development of the network, proven technologies like the containerization technology 'Docker' and the workflow design gitHub-Actions were used. The software was distributed on servers provided by the Carl von Ossietzky University of Oldenburg, thus implementing the distributed learning strategy for the neural network, graphically illustrating the course of disease progression and the results of decisions using the Google Maps API. At the end of the documentation, the implementation of the tests and the organisation of the work were discussed.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Quick Start Guide . . . 8

2 Grundlagen **10**
2.1 Heuristiken . . . 10
2.1.1 Struktur und Funktionsweise . . . 10
2.1.2 Stärken und Schwächen . . . 11
2.2 Entscheidungsbäume . . . 11
2.2.1 Struktur und Funktionsweise . . . 11
2.2.2 Stärken und Schwächen . . . 11
2.3 Genetische Algorithmen . . . 12
2.3.1 Struktur und Funktionsweise . . . 12
2.3.2 Stärken und Schwächen . . . 12
2.4 Neuronale Netze . . . 13
2.4.1 Struktur und Funktionsweise . . . 13
2.4.2 Stärken und Schwächen . . . 13
2.5 Arten des maschinellen Lernens . . . 14
2.5.1 Überwachtes Lernen ('Supervised Learning') . . . 14
2.5.2 Unbeaufsichtigtes Lernen ('Unsupervised Learning') . . . 14
2.5.3 Bestärkendes Lernen ('Reinforcement Learning') . . . 14
2.6 Zwischenfazit . . . 16

3 Analyse **17**
3.1 Schnittstellenanalyse: ic20_Tool . . . 17
3.2 Qualitative Datenanalyse: ic20_Tool . . . 18
3.2.1 Anforderungen an die Analyse des ic20_Tool-Tools . . . 18
3.2.2 Datenstrukturanalyse: Output . . . 19
3.2.3 Datenstrukturanalyse Input . . . 23
3.3 Quantitative Datenanalyse . . . 26
3.3.1 Keine Berücksichtigung der Aktion 'closeConnection' . . . 26
3.3.2 Krankheitsausbreitung . . . 26
3.3.3 Hypothesen . . . 27
3.4 Anforderungen laut Aufgabenstellung . . . 28
3.4.1 Funktionale Anforderungen . . . 28
3.4.2 Definition der Optimierungsfunktion . . . 28

4 Aufbau Entscheidungskomponente **30**
4.1 Aufbau neuronales Netze . . . 30
4.1.1 Aktivierungsfunktion . . . 30
4.1.2 Initiale Konfiguration des Netzes . . . 30
4.1.3 Q-Learning . . . 30
4.1.4 Reward Function . . . 31
4.2 Unterschiedliche Netztypen . . . 32
4.2.1 Ansatz1: Krankheitsnetz . . . 33
4.2.2 Ansatz1: Stadtnetz . . . 34

4.2.3 Ansatz2: Kombiniertes Netz 34

5 Verwendete Technologien **36**
5.1 Containerisierung und dessen Orchestrierung 36
5.2 Versionsverwaltung 36
5.3 Programmiersprache mit Frameworks 37

6 Deployment **38**
6.1 Zugriff 38
6.2 Monitoring 38
6.3 Trainingsdeployment 39
6.4 Ausführungsdeployment 40
6.5 Verwaltung der Software auf den Trainingsmaschinen 40

7 Visualisierung **42**

8 Testdokumentation **44**
8.1 Aufbau 44
8.2 Umsetzung der Tests 44

9 Arbeitsorganisation **45**
9.1 Kommunikation 45
9.2 Planung 45
9.2.1 Prototypen 46
9.2.2 Fertiges Programm 46
9.2.3 Abgabe 46
9.3 GitLab issue board 46

10 Auswertung **47**

11 Fazit und Ausblick **48**
11.1 Fazit 48
11.2 Ausblick 48

12 Anhang **49**

Literaturverzeichnis **50**

Quellcodeverzeichnis

1 Container von DockerHub herunterladen und ausführen 8
2 Container lokal bauen 9
3 Starten der Visualisierung 9
4 Monitoring Skript 38
5 Monitoring Skript Output per E-Mail 38

Abbildungsverzeichnis

1 Request-Response Flow über den Webclient und -server 8
2 Neuronales Netzwerk fuer Regression mit zwei *hidden layers* [Ric19] 13
3 Messageaufbau 19
4 Outcome 19
5 City 20
6 City Events 20
7 Outbreak Event 21
8 Beispiel für die Medizinausgabe (Impfstoff). 21
9 Beispiel für Stadteingaben beeinflussendes Event 21
10 Globale Events 22
11 Large Scale Panic 22
12 Globales Event: vaccineInDevelopment 22
13 Globales Event: vaccineAvailable 23
14 Aufbau eines Pathogens 23
15 Ausprägungsrepräsentation 23
16 Optimierungsfunktion 29
17 Actor-Critic [SB98] 32
18 Stadt und Krankheitsnetz 33
19 Aufbau des Krankheitsnetzes 34
20 Aufbau des Stadtnetzes 34
21 Gemeinsames Netz 35
22 Block Diagramm zum Trainingsdeployment 40
23 Aufbau unsers Docker Deployments (in Anlehnung an [Doc20]) 41
24 Sequenzdiagramm des Datenaustausches zwischen Visualisierung, Webserver und -client 42
25 Karte von der Visualisierung 43
26 Entwicklung des Modells 47

1 Einleitung

Die Aufgabe des diesjährigen InformatiCups war die effiziente Auswahl und effektive Anwendung von möglichen Gegenmaßnahmen gegenüber gefährlichen Krankheiten zu evaluieren. Für die Evaluierung stand eine Simulation von Städten bereit, in welcher bestimmte Aktionen pro Runde ausgeführt werden. Es können Krankheiten in verschiedenen Städten ausbrechen, indem sich Keime über Flug- oder Landverbindungen verbreiten. Das Ziel der Simulation ist, möglichst viele Keime in möglichst kurzer Zeit zu vernichten und die Ausbreitung der Keime einzudämmen, um die Anzahl der ausbrechenden Krankheiten zu minimieren und die Menschheit vor der Auslöschung durch eine Pandemie zu bewahren. Die möglichen Gegenmaßnahmen, um dies zu bewerkstelligen sind: Quarantäne anordnen, Flughafen schließen/ Flugverbindung sperren, Impfstoff/Medikament entwickeln/verteilen und politischen Einfluss geltend machend, Neuwahlen ausrufen, Hygienemaßnahmen durchführen, Informationskampagne starten. Teilweise gibt es Abhängigkeiten zwischen den Maßnahmen, bspw. muss ein Impfstoff/Medikament erst entwickelt werden, bevor es nach X Runden zur Verfügung steht und verteilt werden kann. In einer Runde können zwar mehrere Gegenmaßnahmen getroffen (oder initiiert) werden, dabei kosten die Aktionen jedoch Punkte, die für einige Aktionen von der angegebenen Anzahl der Runden, für die die Gegenmaßnahme halten soll, abhängig sind. Die Punkte werden von einem Punktekonto abgezogen, das mit 40 Punkten startet und auf das pro Runde 20 weitere Punkte addiert werden.
Die Simulation ist in Form eines Webclients als Binary-Datei zur Verfügung gestellt, welche POST-Anfragen mit JSON-Body an einen Webserver stellt (siehe Abbildung 1). Die Anfragen enthalten den kompletten Weltzustand mit allen Städten, aktueller Rundenanzahl, Punkteanzahl sowie getriggerten Aktionen vom Webserver und vom Client. Der Webserver muss als Antwort, ebenfalls im JSON-Format, auf die Anfrage die Aktion schicken, die ausgeführt werden soll. Wenn der Webserver mit einer nicht rundenendenen Aktion antwortet, ändert sich die Runde in der nächsten Anfrage nicht, es wird lediglich die getriggerte Aktion zu den ausgeführten Aktionen hinzugefügt. Dadurch kann der Webserver auch mehrere Aktionen in einer (Spiel-)Runde ausführen. Sollen mehrere Aktionen durchgeführt werden, werden dementsprechend unterschiedliche einzelne Anfragen geschickt, auf die wiederum einzelne Antworten entgegnet werden. Unsere Aufgabe ist es diesen zustandslosen Webserver zu schreiben, der mit den dargestellten möglichen Aktionen versuchen soll, alle Keime möglichst schnell zu eliminieren.

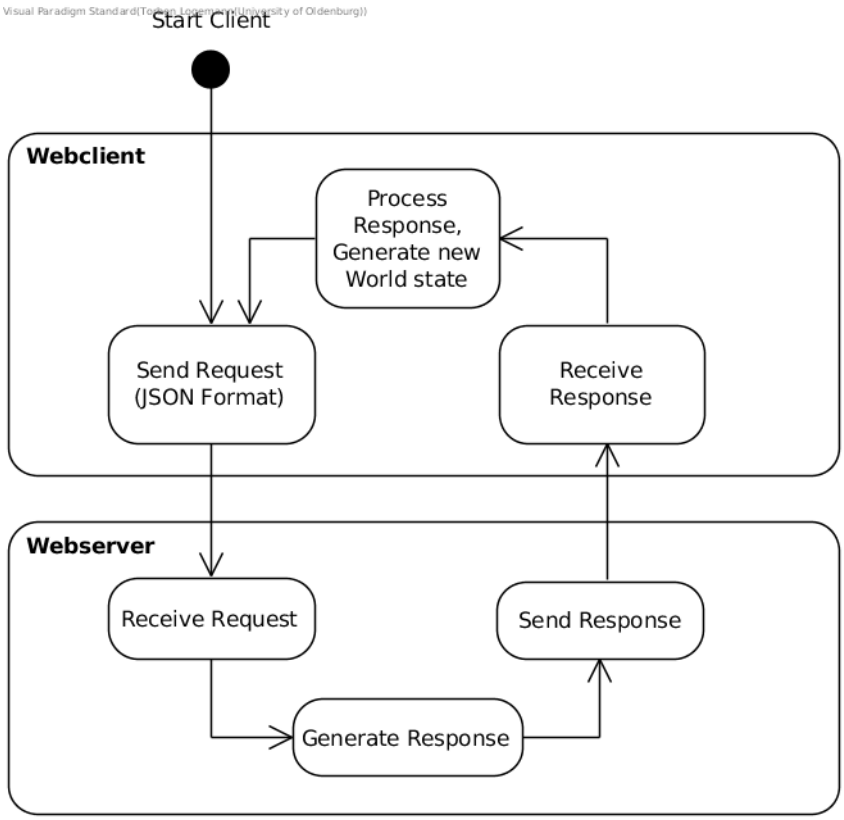

Abbildung 1: Request-Response Flow über den Webclient und -server

1.1 Quick Start Guide

In diesem Abschnitt werden in aller Kürze die Schritte erläutert, die notwendig sind, um den aktuellen Dockercontainer herunterzuladen. Außerdem wird beschrieben, wie der Container im Docker Hub hinzugefügt wird.

Container von Docker Hub herunterladen

Um die in Quellcode 1 stehenden Kommandos auszuführen, muss man sich zuerst bei Docker Hub anmelden. Danach kann der Container heruntergeladen und gestartet werden. Mit der Option *-it* wir die Visualisierung gestartet

```
# login to docker
docker login --username=pgronau --password=5e3a7fe8-2d31-4a8e-ba1f-5782214a8556
# pull last docker container
docker pull pgronau/infocup:solution
# run docker container
docker run -itd -p 50123:50123 --name sol1 pgronau/infocup:solution
# alt. run docker container with visualization
docker run -itd -p 50123:50123 --name sol1 pgronau/infocup:solution -vi
```

Quellcode 1: Container von DockerHub herunterladen und ausführen

Container lokal bauen

Wenn der Container lokal gebaut werden soll, dann können folgende Befehle (siehe Algorithmus Quellcode 2) ausgeführt werden. Die Befehle müssen innerhalb des Wurzelverzeichnisses des Projekts ausgeführt werden.

```
# Build solution image
docker build -t solution -f zSolution.Dockerfile .
# Run container based on image
docker run -itd -p 50123:50123 --name sol1 solution:latest
```

Quellcode 2: Container lokal bauen

Starten Visualiserung

Starten der Visualisierung wie in Quellcode 3 beschrieben.

```
# starten der Visualisierung
docker run -itd -p 50123:50123 --name sol1 solution:latest -vi
```

Quellcode 3: Starten der Visualisierung

2 Grundlagen

In diesem Kapitel werde die möglichen Lösungsansätze kurz mit ihren Vor- und Nachteilen vorgestellt, in Unterabschnitt 2.6 miteinander verglichen und anschließend die für die Lösung der Aufgabenstellung beste Option ausgewählt. In Unterabschnitt 2.1 werden Heuristiken vorgestellt. Im Anschluss daran werden Entscheidungsbäume und genetische Algorithmen vorgestellt. In Unterabschnitt 2.4 wird der Aufbau von neuronalen Netzen beschrieben und im Anschluss daran die verschiedenen Kategorien des maschinellen Lernens vorgestellt.

2.1 Heuristiken

Der Begriff *Heuristik* kommt aus dem griechischen Sprachgebrauch und bedeutet übersetzt auffinden oder entdecken. Anwendungsgebiete sind unter anderem in der Psychologie, Wirtschaftswissenschaften, Mathematik und Informatik. Heuristiken dienen als Werkzeug, um mit unvollständigen Daten, begrenztem Wissen und wenig Zeit trotzdem eine gute Entscheidung zu finden. [Kat17].

2.1.1 Struktur und Funktionsweise

Die Struktur sowie die Funktionsweise lassen sich anhand der Eigenschaften/Charakteristiken von Heuristiken beschreiben. Diese drei Punkte treffen auf alle Heuristiken zu:

1. Teilbetrachtung des Lösungsraumes

Um ein schnelles Ergebnis zu liefern, wird ein Teil des Lösungsraumes verworfen. Dabei kann es natürlich passieren, dass die gesuchte Lösung bzw. die optimale Lösung innerhalb des verworfenen Bereichs liegt. Damit ist die Zuverlässigkeit zum Finden einer optimalen Lösung mit einer Heuristik nicht mehr gegeben [Han92].

2. Suchalgorithmus im Restbaum

Die Suche nach einer Lösung wird im verbleibenden Restbaum vollzogen. Dabei wird keine zufällige Suche vollzogen, sondern anhand von definierten und intelligenten Regeln gesucht [Han92].

3. Konvergenz gegen eine Lösung

Bei einer Heuristik kann nicht garantiert werden, dass es zu einer Annäherung an eine bestimmte Lösung kommt. Es besteht also keine Lösungsgarantie [Han92].

Zu den Heuristiken können noch weitere Eigenschaften gezählt werden. Zum einen die subjektiven **Stoppregeln** und des Weiteren die **Steuerungsmöglichkeiten**. Da diese aber auch als Eigenschaften von anderen Methoden zählen und somit nicht als eindeutige Charakteristik einer Heuristik zählen, wird auf diese nicht weiter eingegangen [Han92].

2.1.2 Stärken und Schwächen

Zu den Vorteilen von Heuristiken lässt sich zum einen die Einfachheit der Implementierung zählen. Dadurch wird die Heuristik zu einem sehr begehrten Werkzeug aus der Praxis. Des Weiteren liefern Heuristiken eine schnelle Entscheidung. In Verbindung mit der einfachen Implementierung ist der Praxiserfolg gut nachzuvollziehen [Ork].

Jedoch bringt auch die Heuristik wie jedes andere Werkzeug Nachteile mit sich. Es ist unklar, ob die optimale Entscheidung gefunden wurde oder wie weit die Entfernung zu dieser noch ist. Als zweiter Nachteil einer Heuristik gilt die Problematik, dass bei großen Problemklassen keine Heuristik eingesetzt werden kann [Ork].

2.2 Entscheidungsbäume

Bei Entscheidungsbäumen handelt es sich um ein Tool mit dem komplexe Entscheidungen getroffen werden können. Es handelt sich dabei, wie der Name schon zeigt, um eine baumartige Struktur mit vielen Verästelungen. Entscheidungsbäume können zur Lösung von logischen oder mathematischen Probleme genutzt werden [CF12].

2.2.1 Struktur und Funktionsweise

Ein Entscheidungsbaum beginnt, wie ein echter Baum, mit einer Wurzel. Diese Wurzel stellt den ersten Knoten dar. Der Input läuft nun von der Wurzel durch die Knoten in die Blätter des Baums. Letztendlich stellen die Blättern verschiedene Entscheidungen dar. Bei jeder Abzweigung, also an jedem Knoten wird eine Teilmenge der Entscheidungen ausgeschlossen. Bei einem Durchlauf des Baums kommt man dann bei einer Entscheidung an. Um von der Wurzel zu den Entscheidungen zu kommen, werden die Äste an den einzelnen Knoten benötigt. Der richtige Weg durch den Entscheidungsbaum, entlang der Knoten und Äste, ergibt sich durch die Gewichtung und Bewertung der Äste. Diese Gewichtung kann beispielsweise die Auswahl der Krankheit sein, für die eine Impfstoffentwicklung den größten Effekt mit sich bringt [CF12].
Bezogen auf das Projekt spiegelt das Abbild der Welt, welches von dem ic20_Tool-Tool ausgegeben wird, den Input an die Baumwurzel wider. Ausgehend von den Werten, die sich im Weltabbild befinden, werden die Knoten entlang der Äste bis unten zu den endgültigen Entscheidungen gegangen. Als mögliches Beispiel:
Ein möglicher Knoten im Entscheidungsbaum könnte prüfen, ob es notwendig ist, eine Stadt unter Quarantäne zu setzen. Aus dem Input geht aber hervor, dass diese Stadt von keiner Krankheit infiziert ist und somit eine Quarantäne nicht notwendig ist. So werden die einzelnen Entscheidungen ausgeschlossen bzw. auch eingeschlossen und die Entscheidung mit der voraussichtlich höchsten Effektivitätsrate verwendet.

2.2.2 Stärken und Schwächen

Einer der signifikantesten Vorteile eines Entscheidungsbaum ist, dass die formalen Regeln die Möglichkeit gewähren, die Entscheidung einfach nachzuvollziehen. Bei falsch kategorisierten Daten können die formalen Regeln durchgegangen werden und dementsprechend angepasst werden [CF12].

Die Nachteile, die gegen Entscheidungsbäume sprechen sind unter anderem, dass ein Baum sehr schnell in die Breite wächst, wenn eine Eigenschaft viele verschiedene Ausprägungen hat. Dadurch entsteht ein höherer Aufwand und die Effizienz der Suche nimmt ab. Ein Entscheidungsbaum dient lediglich für abzählbare Datentypen [CF12].

2.3 Genetische Algorithmen

Genetische Algorithmen gehören zum Teilgebiet der evolutionären Algorithmen, welche angelehnt an die natürliche Evolution sind. Die Algorithmen starten mit einer beliebigen Anfangslösung und gelangen durch wiederholtes Selektieren und Verändern der Lösung zu einem Ergebnis.

2.3.1 Struktur und Funktionsweise

Als erstes wählt der Algorithmus eine beliebige Startpopulation aus. Auf Grundlage der Startpopulation werden sogenannte Chromosome ausgewählt. Die Fitnessfunktion bestimmt den Fitnesswert und somit auch die Wahrscheinlichkeit, die ein Chromosom hat, um in die nächste Generation übernommen zu werden. Die Auswahl wird über die folgende Gleichung 1 bestimmt [Har07]:

$$p(c_i) = \frac{f(c_i)}{\sum_{j=1}^{n} f(c_j)} \tag{1}$$

Dabei sei $c_i \in \{c_1, c_2, ..., c_n\}$ ein Chromosom und $f(c)$ bestimmt die Fitness für ein Chromosom. Im nächsten Schritt, wenn die besten Chromosomen ausgewählt wurden, gilt es diese zu verändern. Durch das Erstellen unterschiedlicher Mutationen können bessere Lösungen erzeugt werden. Für die Algorithmen muss eine Abbruchbedingung festgelegt werden, die den Algorithmus nach einer festgelegten Zeit oder sollte dieser keine besseren Lösungen mehr finden terminiert [Har07].

2.3.2 Stärken und Schwächen

Genetische Algorithmen werden im Allgemeinen für die Lösung von $\mathbb{NP}$ schweren Problemen verwendet. Darunter fällt unter anderem das Rucksackproblem. Ein weiteres bekanntes Problem, welches durch genetische Algorithmen gelöst werden kann ist das traveling salesman problem (TSP) [Har07].

Ein Nachteil der Algorithmen ist, dass diese nicht immer die optimale Lösung liefern. Wenn sich dem lokalem Maximum der Zielfunktion genähert wird und die genetischen Operatoren keine ausreichend starken Veränderungen hervorbringen, kann das globale Maximum nicht gefunden werden. Ein weiterer Nachteil ist die lange Laufzeit der Algorithmen, solange keine $\mathbb{NP}$ schweren Probleme gelöst werden oder eine Näherung der optimalen Lösungen akzeptabel ist [Har07].

2.4 Neuronale Netze

Ein neuronales Netz ist eine Datenstruktur mit dem Lösungen aus dem Gebiet der künstlichen Intelligenz und des Machine Learning umgesetzt werden können. Das Netz bekommt dabei eine Menge von Eingaben und kann für eine Eingabe eine Menge von Ausgaben erzeugen. Abhängig von den Informationen, die ein Netz zu seinen Entscheidungen bekommt, lernt es, welche Entscheidungen für eine Menge an Eingaben gut und welche eher schlecht ist.

2.4.1 Struktur und Funktionsweise

Ein neuronales Netz wird durch einen gerichteten Graphen repräsentiert. Die Knoten werden in einem neuronalem Netz als Neuronen bezeichnet. Der Graph wird unterteilt in unterschiedliche Layer. Es gibt einen Eingabevektor und einen Ausgabevektor. Zwischen den beiden Vektoren wird ein Netz aufgespannt bestehend aus x, wobei $x \in \mathbb{N}$ sogenannten Hidden Layern. Die Anzahl der Layer bestimmt die Lerndauer und die Güte des angenäherten Ergebnisses. Alle Neuronen einer Ebene werden mit allen Neuronen der nächsten Ebene verbunden.

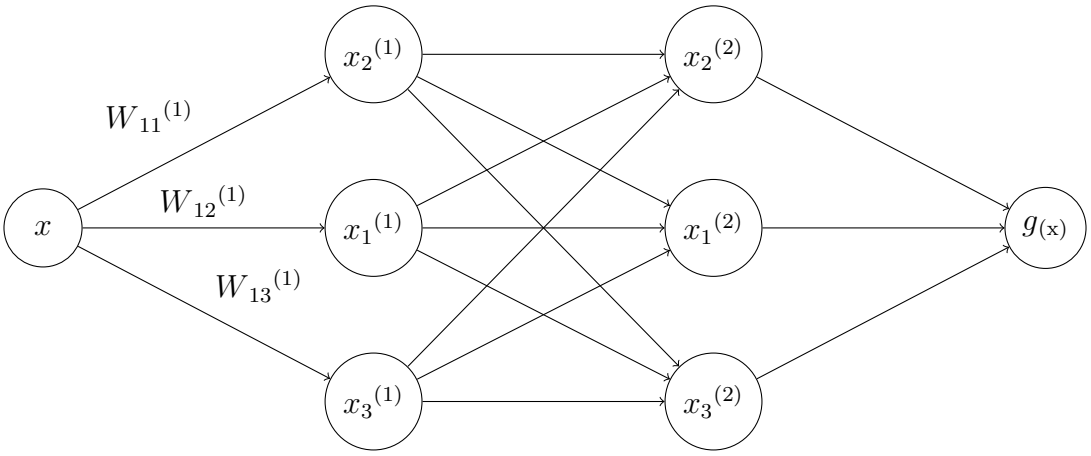

Abbildung 2: Neuronales Netzwerk fuer Regression mit zwei *hidden layers* [Ric19]

In Abbildung 2 ist ein neuronales Netz dargestellt bestehend aus einem Eingabeneuron und einem Ausgabeneuron. Beide Hidden Layer haben jeweils eine Breite von drei Neuronen [Kaf17].

2.4.2 Stärken und Schwächen

Vorteile eines neuronales Netzes sind, dass diese bei verrauschten oder unvollständigen Datensätzen eingesetzt werden können. Es ist möglich, für komplexe und nicht mathematisch beschreibbare Probleme gute Lösungen zu finden. Bei ausreichend komplexen Problemen ist es möglich, bessere Lösungen zu finden als mit statischen Lösungsansätzen [Kaf17, RN04].

Eine Schwäche eines neuronalen Netzes ist unter anderem, dass schwer nachvollzogen werden kann, wie ein Ergebnis zustande kommt. Desweiteren brauchen neuronale Netze

viele Trainingsdaten und die Ergebnisse hängen von der Qualität der Trainingsdaten ab. Eine weitere Schwäche ist, dass zu keinem Zeitpunkt bekannt ist, ob das gefundene Maximum ein lokales oder globales ist [Kaf17, RN04].

2.5 Arten des maschinellen Lernens

Maschinelles Lernen ist ein Teilgebiet der künstlichen Intelligenz ('Artificial Intelligence'). Dabei werden Algorithmen und statistische Daten zusammengeführt, damit der Computer in der Lage ist, eine konkrete Aufgabe ohne spezielle Instruktionen zu lösen.

2.5.1 Überwachtes Lernen ('Supervised Learning')

Wenn von überwachtem Lernen gesprochen wird, dann wird ein vorhandenes Modell mit Trainingsdaten trainiert bei denen zu jedem Input ein erwartetes Ergebnis vorhanden ist. Mit den Informationen aus dem Input, dem berechnetem Output des Modells und dem erwartetem Ergebnis kann das Modell trainiert werden. Hierbei ist die einfachste Form des Lernens die binäre Klassifikation neuer Daten (z.B. gesund oder krank). Aufgrund vorangegangener Ereignisse bzw. vorhandener Daten kann das Modell neue Daten einordnen und Voraussagen treffen. Konkrete Probleme sind hier:

- Handschrifterkennung
- Aktienpreise
- Spamerkennung
- Spracherkennung

Alle Algorithmen des überwachten Lernens fordern Probleme mit der Struktur, dass es einen Inputvektor X gibt und einen Ergebnisvektor Y. Der Eingabevektor besteht aus Daten (z.B. Bild, Ton, Text usw.) welche umgewandelt werden in einen reelzahligen Vektor dessen Dimension d unter Umständen sehr groß werden kann, wobei $d \in \mathbb{N}$. Dabei wird erwartet, dass ein Zusammenhang zwischen X und Y besteht, was ausgedrückt werden kann als; $Y = f(x)$.

2.5.2 Unbeaufsichtigtes Lernen ('Unsupervised Learning')

Anders als beim überwachtem Lernen gilt es hier, Muster in Daten zu erkennen. Anhand der erkannten Muster können neue Daten Gruppen zugeordnet werden. Dabei gibt es keine Belohnungsfunktion oder ähnliches.

2.5.3 Bestärkendes Lernen ('Reinforcement Learning')

Reinforcement Learning ist ein Teilgebiet des maschinellen Lernens. Der Unterschied zu Supervised Learning besteht darin, dass es hier keine genaue Aussage zu der Entscheidung des Netzes gibt. Es gibt nur ein Endresultat wie z.B. das Spiel wurde gewonnen oder nicht. Dabei sind keine Entscheidungen der Zwischenschritte vorhanden. Das Problem hierbei ist, dass oft Dinge ausprobiert werden müssen und das Lernen dadurch sehr langsam ist. Dabei kann der Systemaufbau beschrieben werden als ein Agent, der mit einer Umwelt in

Interaktion tritt. Der Agent wirkt auf die Umwelt ein und aufgrund der sich ändernden Umwelt kann die Handlung des Agenten über einen sogenannten 'Reward', also eine Belohnung bewertet werden. Ziel ist es, dass der Agent lernt, welche Handlung bei einer Abbildung der Umwelt zum Ziel führt. Formal kann dies wie folgt dargestellt werden, der Agent S bildet auf einen Aktionsraum A mit einem aktuellen Zustand s_t ab:

$$Agent : S \rightarrow A(s_t) \tag{2}$$

Die Umwelt kann als stochastischer Ausdruck beschrieben werden. In einer Situation A wird ein Zustand s_t abgebildet auf den Reward r_t:

$$Umwelt : S \times A \rightarrow P(S \times r_t) \tag{3}$$

Ein Agent kann nicht immer einen Zustand komplett erfassen, weswegen im Folgendem von **Situationen** gesprochen wird. Einem Agenten ist es nicht möglich, eine Situation zu beurteilen, weswegen von der Umwelt ein Reward verteilt werden muss, mit dem der Agent lernt. Der Reward ist immer ein skalierter Wert.
Um das neuronale Netz möglichst gut zu trainieren reicht es nicht immer, den besten Weg zu gehen, da eventuell andere Wege, die zu einem besserem Ergebnis führen, so niemals erforscht würden. Daher gibt es zwei unterschiedliche Ansätze, die parallel laufen: den **Exploitation-Ansatz** und den **Exploration-Ansatz**.
Bei dem Exploitation-Ansatz wird wird eine **Greedy Policy** ausgeführt, welche immer den Weg mit dem höchsten Reward auswählt. Der Exploration-Ansatz hat hingegen das Ziel, das gesamte System möglichst umfangreich zu erforschen und wählt viele zufällige Züge.
Beide Ansätze können kombiniert werden, um das Ergebnis besser zu approximieren. [Ert16].

2.6 Zwischenfazit

Es existiert eine Menge an unterschiedlichen Lösungsstrategien zum Bewältigen der Aufgabe. In Abhängigkeit der Komplexität des Problems und der Menge an Entscheidungsmöglichkeiten lassen sich mehrere Strategien wählen. Die in Unterabschnitt 2.1 vorgestellten Heuristiken führen dazu, dass schnell mit einem bewährten Ansatz eine Implementierung vorgenommen werden kann. Jedoch ist die Menge an Daten die verarbeitet werden so groß, dass dies eher ein ungeeigneter Ansatz gewesen wäre.
Die Entscheidungsbäume hätten eine schöne Repräsentation der Entscheidungsstruktur, die für die Lösung der Aufgabe nötig ist, wären aber zu groß geworden und Algorithmen hätten nicht performant genug gute Lösungen geliefert.
Aufgrund der hier aufgeführten Argumente fiel die Entscheidung auf die Implementierung eines neuronalen Netzes. Das Problem welches gelöst werden soll befindet sich im Kontext von 'Reinforcement Learning'. Dazu gibt es in [FvHM18] einen Lösungsansatz, welcher in diesem Projekt umgesetzt werden soll. Ähnlich wie bei Forschungsproblemen, welche die Spiele Schach oder Backgammon lösen kann das hier bearbeitete Problem gelöst werden. Abhängig vom Spielausgang kann eine Aussage getroffen werden, ob die Entscheidungen gut oder schlecht waren. Das Problem beim Umsetzen dieser Lösung ist, dass die Modelle langsam lernen und viele Iterationen benötigen um gute Entscheidungen treffen zu können.

3 Analyse

In Unterabschnitt 3.1 werden die Eigenschaften des uns zur Verfügung gestellten ic20_Tool Tools beschrieben. Damit soll die Grundlage für das Verständnis der Funktionen des Tools auf der Anwendungsebene gelegt werden. In Unterabschnitt 3.2 werden die Datensätze des Tools analysiert. Das Analysieren des Systemverhaltens ermöglicht es abzuschätzen, wie mit dem Tool interagiert werden muss, um einen Lösungsansatz zu generieren. Darauf aufbauen wird das Tool betrachtet, um den Lösungsansatz im ersten Schritt zu konkretisieren um dann folgend das System zu optimieren. Im letzten Unterabschnitt 3.3 wird beschrieben, welche Funktionen keine Auswirkungen auf die nächsten Spielrunden haben und deshalb nicht mit in die Entscheidungen integriert werden müssen.

3.1 Schnittstellenanalyse: ic20_Tool

Bei dem uns zur Verfügung gestellte Tool handelt es sich um ein in der Kommandozeile ausführbares Programm. Das uns zur Verfügung gestellte Tool kann mit den Flag -o, -s, -t, -u Parameter übergeben bekommen.

–**log-file-path** Pfad an den Ausgabeinformationen des Tools geschrieben werden. Im Standardzustand wird nach std-out geschrieben.
Die Ausgabeinformation kann verwendet werden, um die Zustände zu sichern, welche im Laufe eine Spiels aufgetreten sind. Diese Information wurde genutzt, um die Datenstruktur zu analysieren, die von dem verwendeten Tool gesendet wurde.

–**random-seed** Random Seed. Default: Unixtimestamp in nanoseconds.
Das Setzen des Seeds bietet einen Vorteil und einen Nachteil. Ein Benchmark setzt eine zwischen verschiedenen Versuchsdurchläufen vergleichbare Umwelt voraus. Dies kann durch das Setzen eines Seeds ermöglicht werden. Dabei muss jedoch beachtet werden, dass ein Algorithmus welcher bei einem bestimmten Seed eine gute Performance aufweist nicht zwangläufig auch bei anderen Seeds gut performt. Aus diesem Grund müssen für einen Benchmark mehrere Seeds verwendet werden. Des Weiteren muss der Unterschied der durch die verschiedenen Seeds generierten Zustände sich ausreichend stark unterscheiden, um eine Allgemeingültigkeit des Benchmarks zu ermöglichen.

–**request-timeout** Timeout. Dauer, die der Client wartet, bis er eine Antwort erhält.
Diese Funktion ermöglicht es, die Dauer bis zum automatischen Abbruch der Kommunikationsverbindung zu dem von uns erstellten Programmen zu erhöhen oder zu reduzieren. Diese Funktion wird genutzt, um die Kommunikationslatenzen unter anderem im Debugprozess zu reduzieren.

–**endpoint-url** Endpunkt an den sich das Kommandozeilentool verbinden soll.
Genutzt werden kann dieses Argument um die Verbindung zu dem von uns generierten Tool herzustellen. Zum einen ermöglicht es das Kommunizieren auf einem Computer. Es wäre jedoch auch möglich, ein verteiltes System im Netzwerk aufzubauen, bei dem die von uns erstellte Lösung auf einem anderen Computer bereitgestellt wird.

3.2 Qualitative Datenanalyse: ic20_Tool

In diesem Abschnitt wird das Tool und damit insbesondere seine Ausgabedaten analysiert. Dafür werden zunächst Anforderungen an die eigentlichen Analyse gestellt, in Hinblick auf die später potentielle Verwendung von Neuronalen Netzen.

3.2.1 Anforderungen an die Analyse des ic20_Tool-Tools

Zur Analyse der Datenstruktur wurden die gesendeten Daten geloggt, um eine Übersicht der erhaltenden Daten zu bekommen. Darauf folgend wurde das Verhalten der Daten betrachtet. Ziel ist es somit die Form eines neuronalen Netzes aus der Datenstruktur abzuleiten. Zu beachten gilt dabei:

- Ein FeedForward Network kann schwer mit Rekursion umgehen
- Ein FeedForward Network benötigt eine feste Eingabedimension
- Das Training wird erleichtert durch die Normalisierung der Eingabedimensionen (Alle möglichen Eingabewerte sollten sich in der selben Größenordnung befinden)

Dafür gilt es in der Datenanalyse folgende Eigenschaften zu erkennen:

- Rekursion
- Sammlungen im Sinne von Listen oder Mappings wiederkehrender Datenstrukturen
- Elementare Datentypen
- Abbildung von Daten in einem numerischen Format für den Netzin-/output
- Wertebereich der Datenpunkte zur Normalisierung des Netzin-/outputs
- Globale Werte, die durch lokal verstreute zusammengesetzt sind

Für einen Algorithmus, der neuronale Netze verwendet, aber die folgenden Eigenschaften in den externen Datenstrukturen bekommt, gilt folgendes:

Rekursion Kann in einem Feed Forward Network nicht abgebildet werden. Muss entweder substituiert werden, oder es gilt eine passende Abbruchbedingung zu definieren.

Sammlung Muss außerhalb des neuronalen Netzes als Schleife implementiert werden.

Elementare Datentypen Müssen auf einen numerischen Datentypen abgebildet werden, dienen als Netzin-/output.

Wertebereich Von besonderem Interesse sind die Min/Max Werte für die Bestimmung einer Normalisierungsfunktion.

Globale Werte Müssen durch die lokal verstreuten Werte zusammengesetzt werden, da in einem neuronalen Netz die Zusammengehörigkeit der Werte ohne Weiteres verloren gehen würde

3.2.2 Datenstrukturanalyse: Output

Zur Abschätzung mit welcher Art von Algorithmus das Spiel gewonnen werden kann, wird ein Überblick über die Struktur der eingehenden und ausgehenden Daten benötigt. Da bei dem ic20_Tool Tool JSON als Datenstruktur verwendet wurde, ist davon auszugehen, dass eine spezifische Syntax für den Aufbau der Repräsentation von Spielzuständen existiert.

Als Technik zur Syntaxanalyse bietet sich die Darstellung in der Backus-Naur Form an. In unserem Fall haben wir das JSON mit der erweiterten Backus-Naurs Form modelliert um eine Abschätzung des Verhaltens unseres zukünftigen Lösungsansatzes zu gewinnen.

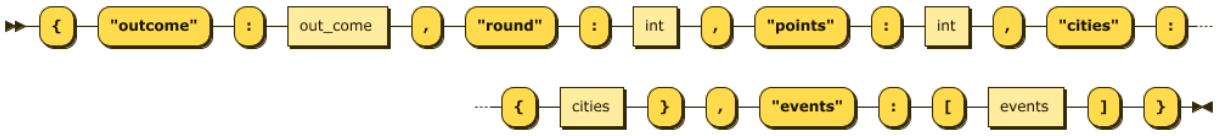

Abbildung 3: Messageaufbau

Die Message beginnt mit dem Schlüssel *outcome*, welcher einen String enthält. Darauf folgen zweite Einträge *round*, sowie *points*. Beide verwenden den elementaren Datentypen Integer. Der vierte Eintrag ist *cities* welcher aus einem assoziativen Datenfeld besteht. Der letzte Schlüssel lautet *events* und beinhaltet eine Liste von weiteren assoziativen Datenfeldern. Für den auszuwählenden Algorithmus können die Werte der Schlüssel *round* und *points* direkt übernommen werden. Ihr Informationswert ist direkt aus dem Schlüssel erkennbar; sie benötigen keine weitere Vorverarbeitung und können direkt in das Programm eingespeist werden.

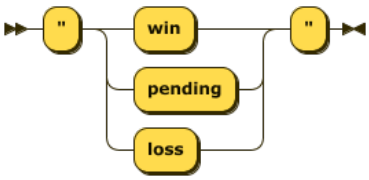

Abbildung 4: Outcome

Der *outcome* (Abbildung 4) besteht aus einem String mit drei verschiedenen Werten: *win*, *pending* und *loss*. Diese Werte können in dem Programm verwendet werden, um den Spielzustand zu erfassen. Bei *pending* hat das Spiel eine Aktion auszuwählen. Die Zustände *win*, sowie *loss* signalisieren das Durchlaufsende.
Der Abschnitt *City* (Abbildung 5) ist der komplexeste Abschnitt mit zehn unterschiedlichen Schlüsseln. Der erste Schlüssel *name* beinhaltet den Namen der Stadt, dieser Schlüssel kann verwendet werden, um von anderen Stellen im JSON auf die Informationen dieser Stadt zu verweisen. Die Einträge *latitude* und *longitude* liegen als float vor und signalisieren die Position der Stadt in einem Koordinatensystem. Auf Grundlage der Syntax und der Semantik der Koordinaten kann davon ausgegangen werden, dass ein geographisches Koordinatensystem zugrunde liegt. Der Eintrag *population* enthält vom Datentypen ein Integer und spiegelt die Einwohnerzahl wider. Der Schlüssel *connections* enthält eine Liste von anderen Städtenamen mit denen die Stadt verbunden ist. Die Städtenamen können

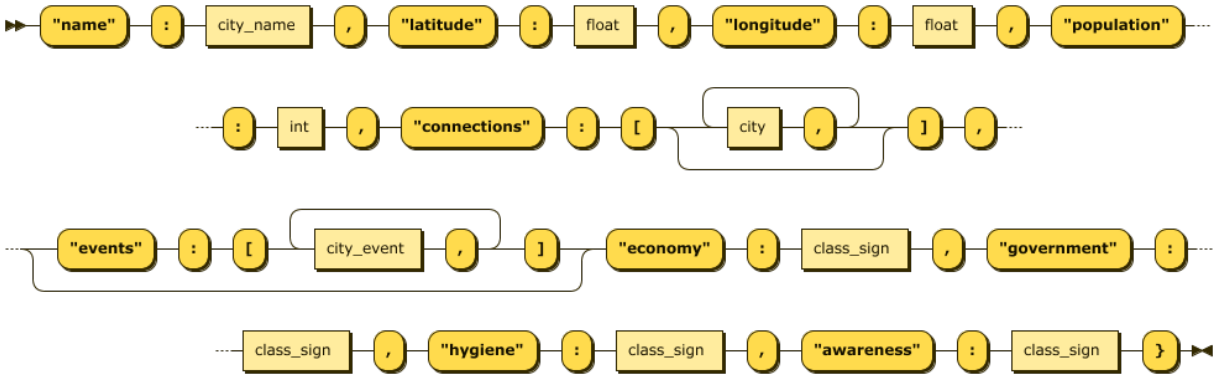

Abbildung 5: City

im JSON unter *cities* als Schlüssel verwendet werden, um auf die Werte der verbundenen Städte zu zuzugreifen. Dies ist von besonderer Bedeutung, da in diesem Bereich zyklische Abhängigkeiten zwischen den Städten bestehen können. Der Eintrag *events* teilt sich seinen Namen mit dem Schlüssel auf der Wurzelebene, beinhaltet jedoch andere Werte. Hier gilt es bei der späteren Verarbeitung darauf zu achten, dass nicht versehentlich über den falschen Schlüssel auf Daten zugegriffen wird. Aus diesem Grund sprechen wir im Folgenden von city_events. Inhaltlich ist dieser Eintrag optional und entfällt bei Städten, in denen keine Events vorliegen. Die letzten vier Schlüssel sind *economy*, *government*, *hygiene* und *awareness*. Diese sind in ihrer Struktur sehr ähnlich. *economy* repräsentiert den wirtschaflichen Zustand der Stadt. *government* signalisiert die politische Stabilität. *hygiene* ist der Zustand der Hygiene in der Stadt und *awareness* steht für die Aufmerksamkeit der Einwohner. Diese vier Schlüssel enthalten alle einen String welcher die Ausprägung repräsentiert (Abbildung 15).

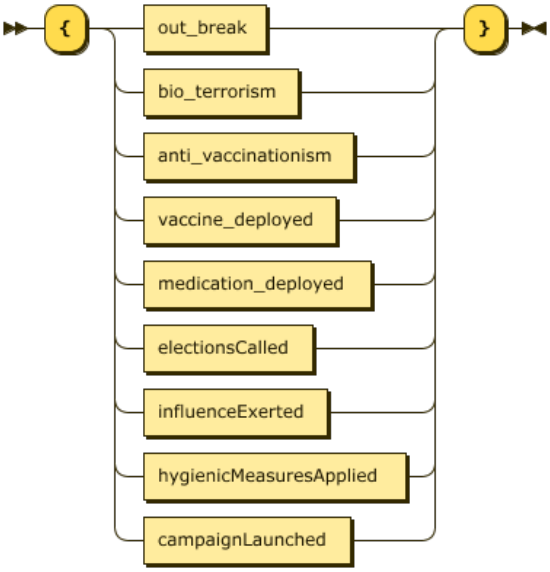

Abbildung 6: City Events

Es existieren elf verschiedene City Events. Diese können inhaltlich in zwei Kategorien unterteilt werden. Zum einen gibt es die Kategorie der unkontollierten Ereignisse, sie repräsentieren Ereignisse auf deren Entstehung nicht unmittelbar eingewirkt wurde.

Zu diesen Ereignissen gehören: out_break, bio_terrorism und anti_vaccination. Die zweite Kategorie beinhaltet Feedback zu den von dem Agenten ausgeführten Aktionen. Hierzu gehören: vaccine_deployed, medication_deployed, electionCalled, influenceExerted, hygienicMeasuresApplied, sowie campaignLaunched.
Bezüglich der Form müssen vier Kategorien unterschiedenen werden. Outbreak beinhaltet

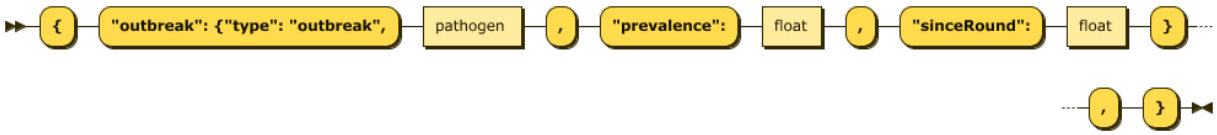

Abbildung 7: Outbreak Event

zum einen eine Krankheit mit ihren kompletten Eigenschaften, eine *prevalence* welche den prozentualen Anteil Infizierter als float darstellt, sowie *sinceRound* mit dem die Runde des Krankheitsausbruches angegeben wird. Das Event BioTerrorism ist vergleichbar mit dem Outbreak aufgebaut, jedoch heißt hier sowohl der Eventschlüssel, als auch der Wert für den Eventtypen *bioTerrorism* statt *outbreak*. Zu beachten gilt auch, dass die Rundenzahl nicht über den Schlüssel *sinceRound* übergeben wird, sondern als *round*.
Das Event *antiVaccinationism* besitzt als Eventschlüssel und als Type den Wert *anti-Vaccinationism*, des Weiteren wird die Runde angegeben ab welcher Runde das Event existiert.

Abbildung 8: Beispiel für die Medizinausgabe (Impfstoff).

Die beiden Events *medicationDeloyed* und *vaccineDeployed* teilen sich ihren Strukturaufbau (Abbildung 8). Zum einen wird eine Krankheit referenziert (siehe Krankheit), zum anderen die Runde in Form eines Integers angezeigt in der der Impfstoff oder das Medikament ausgeteilt wurde.

Abbildung 9: Beispiel für Stadteingaben beeinflussendes Event

Des Weiteren gibt es die Gruppe der Events, welche die Stadteigenschaften beeinflussen. Sie bestehen aus dem Namen des Events und einer Rundenangabe in welcher Runde dieses Event stattgefunden hat. Diese Events können verwendet werden, um zu identifizieren, dass ein Zug in einer Runde ausgeführt wurde (Abbildung 9).

Die letzte Kategorie der Servernachrichten sind die globalen Events. Diese Events sind allgemeingültig und beziehen sich nicht auf eine bestimmte Stadt. Diese Gruppe teilt sich syntaktisch in vier Gruppen (siehe Abbildung 10).
Die erste Gruppe ist *pathogenEncountered* und beseteht aus einem Eintrag. In diesem Event wird auf ein Pathogen verwiesen, welches aufgetreten ist, sowie einer Rundenzahl in der die Krankheit entdeckt wurde.

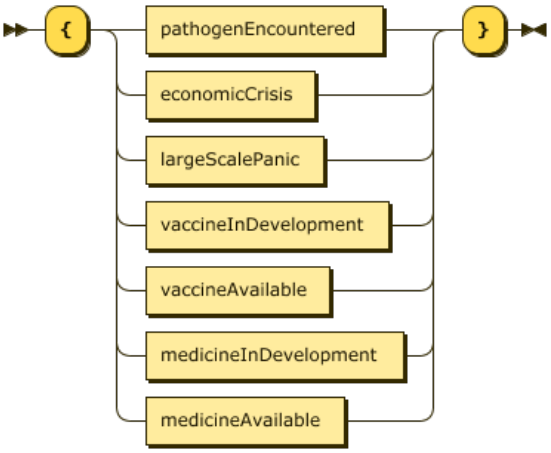

Abbildung 10: Globale Events

Die zweite Gruppe sind zwei Events, welche auf die Eigenschaften der Städte abzielen. *economicCrisis* und *lageScalePanic* bestehen einmal aus dem Event und einer Angabe in welcher Runde sie aufgetreten sind (vgl. Abbildung 11).

Abbildung 11: Large Scale Panic

Die dritte Gruppe gibt an, ob ein Medikament oder ein Impfstoff sich aktuell in der Entwicklung befindet. In diesem Event wird ausschließlich auf ein Pathogen verwiesen, die Information ab welcher Runde die Entwicklung begonnen hat ist nicht vorhanden (siehe Abbildung 12).

Abbildung 12: Globales Event: vaccineInDevelopment

Die vierte Gruppe gibt an, ab welcher Runde ein Medikament oder ein Impfstoff entwickelt wurde. Hier wird sowohl auf ein Pathogen verwiesen, als auch durch ein Integer die Rundenzahl angegeben (vgl. Abbildung 13).

Ein Pathogen besteht zum einen aus dem Namen des Pathogens, sowie vier verschiedenen Attributen, welche das Verhalten der Krankheit beschreiben. Die Eigenschaften sind *infectivity* welches die Infektionsrate angibt, *mobility* mit der die Ausbreitungsgeschwindigkeit beschrieben wird, *duration* mit dem die Dauer der Erkrankung beschrieben wird und *lethality* welches die Tödlichkeit eines Viruses anzeigt. Diese Eigenschaften benutzen das selbe Wertesystem wie die Eigenschaften der Städte (siehe Ausprägungsrepräsentation).

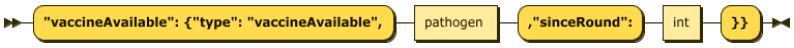

Abbildung 13: Globales Event: vaccineAvailable

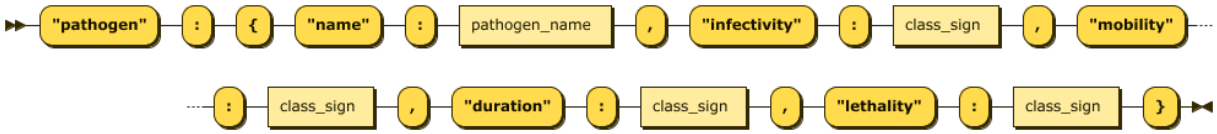

Abbildung 14: Aufbau eines Pathogens

Ausprägungen von Eigenschaften werden ordinalskaliert als String dargestellt. Als Repräsentation wird sie sowohl für die Eigenschaften einer Krankheit als auch für die Eigenschaften im Stadtobjekt verwendet. Diese Werte müssen bei Bedarf in ein ordinalskalierte Enumeration, oder auf einen numerischen Wertebereich abgebildet werden.

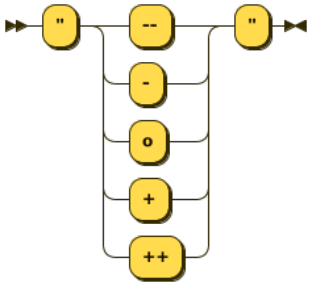

Abbildung 15: Ausprägungsrepräsentation

3.2.3 Datenstrukturanalyse Input

Um gegen eine Krankheit vorgehen zu können, gibt es eine Menge an Aktionen, die durchgeführt werden können. Diese Aktionen werden in Form eines JSON an das Kommandozeilenwerkzeug zurückgesendet. Jede Aktion die ausgeführt wird, kostet Punkte und es können auch mehrere Aktionen pro Runde gemacht werden. Eine Runde ist erst vorbei, sobald die Aktion 'endRound' geschickt wird.

1. Runde beenden

Die wichtigste Aktion ist, die Runden beenden zu können. Dafür wird die Aktion vom Typ 'endRound' an das ic20_Tool zurückgeschickt. Erst wenn diese Aktion geschickt wird, geht es in die nächste Runde. Die Aktion kostet 0 Punkte.

2. Quarantäne anordnen

Eine Stadt kann mit der Aktion 'putUnderQuarantine' unter Quarantäne gesetzt werden. Dies hat zur Folge, dass sich die Krankheiten für R Runden nicht aus dieser Stadt in andere Städte übertragen lässt. Die Aktion kostet 10 x die Anzahl der Runden, wie lange die Stadt unter Quarantäne ist + 20 Punkte.

3. Flughafen schließen

Für eine angegebene Rundenzahl R kann ein Flughafen einer Stadt geschlossen werden. Dadurch kann sich für R Runden keine Krankheit über die Flugverbindung dieser Stadt ausbreiten. Um den Flughafen zu schließen, werden 5 x die Anzahl der Runden wie lange der Flughafen geschlossen sein soll + 15 Punkte benötigt.

4. Flugverbindung sperren

Im Gegensatz zum Schließen des ganzen Flughafens können auch einzelne Flugverbindungen geschlossen werden. Zum Schließen wird eine Stadt benötigt, von der die Verbindung ausgehend ist, eine Stadt zu der die Verbindung geht und eine Anzahl an Runden für wie lange die Verbindung geschlossen sein soll.

5. Impfstoff entwickeln

Für jede Krankheit kann ein Impfstoff entwickelt werden. Dafür muss die Krankheit bereits ausgebrochen sein und der Name muss in der Aktion vorhanden sein. Nach 6 Runden steh der Impfstoff zur Verfügung. Die Kosten für die Entwicklung eines Impfstoffs sind 40 Punkte.

6. Impfstoff verteilen

Sobald ein Impfstoff zur Verfügung steht, kann er in einer Stadt verteilt werden. Dies hat zur Folge, dass alle Menschen in dieser Stadt, die noch nicht infiziert sind, sofort immun gegen die Krankheit werden. Der Impfstoff muss für jede gewünschte Stadt einzeln verteilt werden. Für jede Verteilung in einer Stadt werden 5 Punkte benötigt.

7. Medikament entwickeln

Gleich wie beim Impfstoff kann auch für jede Krankheit ein Medikament entwickelt werden. Dieses Medikament ist 3. Runden nach der Ausführung dieser Aktion verfügbar. Voraussetzung, dass ein Medikament entwickelt werden kann, ist, dass diese Krankheit bereits ausgebrochen ist. Um diese Aktion auszuführen, wird der Name der Krankheit benötigt. Die Kosten um das Medikament zu entwickeln sind 20 Punkte.

8. Medikament verteilen

Das entwickelte Medikament kann sobald es zur Verfügung steht, wie beim Verteilen des Impfstoffes, an eine Stadt ausgeteilt werden. Auch hierbei muss das Medikament einzeln an die gewünschten Städte verteilt werden. Zum Verteilen werden Kosten in Höhe von 10 Punkten benötigt.

9. Politischen Einfluss geltend machen

Durch diese Aktion wird der Wert der Eigenschaft 'Wirtschaft' einer Stadt neu gesetzt. Der Wert wird dabei nicht zwingend verbessert, sondern zufällig neu zugewiesen. Hierfür werden Kosten in Höhe von 3 Punkten veranschlagt.

10. Neuwahlen ausrufen

Wie bei 'Politischen Einfluss geltend machen', wird der Wert der Eigenschaft 'Regierung' einer Stadt zufällig neu zugeteilt. Diese Aktion kostet ebenfalls 3 Punkte.

11. Hygienemaßnahmen durchführen

Die Zuteilung eines neuen Eigenschaftswertes für die Eigenschaft 'Hygienestandards' wird neu und zufällig neu zugeteilt. Kosten belaufen sich auf 3 Punkte,

12. Informationskampagne starten

Als letzte mögliche Aktion aus der Menge, ist die zufällige Neuzuteilung des Wertes 'Achtsamkeit der Einwohner'. Die Kosten sind 3 Punkte.

Es ist wichtig zu beachten, dass jegliche Angabe zu Rundenzahlen mindestens 1 oder größer sein muss. Denn es ist nicht möglich, einen Flughafen für 0 Runden zu schließen. Zur Veranschaulichung der Antworten die an das ic20_Tool zurückgeschickt werden, hier ein Beispiel um die Stadt New York City für 4 Runden unter Quarantäne zu setzen.

```
1 {"type": "putUnderQuarantine", "city": "New York City", "rounds": 4}
```

3.3 Quantitative Datenanalyse

Dieses Kapitel deckt den Bereich ab, der in Form von Zusammenhängen, zahlenmäßigen Granularität oder auch statistischer Vorhersagbarkeit die Daten des ic20_Tools beschreibt[Ste00].
Dabei greifen wir auf eine Vielzahl von uns erhobenen und ausgewerteten Daten zurück, die beim Testen, Ausprobieren und Prüfen des ic20_Tools entstanden sind. Es wird unter anderem darauf eingegangen, wieso die Aktion 'closeConnection' nicht berücksichtigt wurde oder wie sich die Daten des ic20_Tools verhalten bzw. welche speziellen Auffälligkeiten wir festgestellt haben.

3.3.1 Keine Berücksichtung der Aktion 'closeConnection'

Durch umfangreiches Testen der Reaktionen des ic20_Tool-Tools auf die möglichen Aktionen haben wir uns dazu entschieden, die Aktion zum Schließen der Flugverbindungen 'closeConnection' nicht zu betrachten. Die Tests haben ergeben, dass der Nutzen dieser Aktion in keinem Verhältnis zu den Kosten (3 * Anzahl der Runden + 3) steht. Dazu führen wir das folgende Beispiel an:

In der allerersten Runde ist die Stadt Lima von dem Pathogen Rhinonitis infiziert. Die Stadt Lima hat unter anderem eine Flugverbindung nach London, welche bisher von keinem Pathogen infiziert ist. Wird nun die Aktion zum Schließen der Flughafenverbindung von Lima nach London genutzt, so ist die Stadt London in der zweiten Runde trotzdem von dem Pathogen Rhinonitis infiziert. Es ist ausschließbar, dass die Stadt London von einer anderen Stadt infiziert worden ist, denn in Runde 1 des Spiels ist die Stadt Lima die Einzige, die von dem Pathogen Rhinonitis infiziert ist.

Die Anzahl der Tests auf unterschiedlichen Seeds lässt ebenfalls den Fall ausschließen, dass es sich dabei um einen Einzelfall handelt.

3.3.2 Krankheitsausbreitung

Die Auswertung der Daten, welche vom ic20_Tool ausgegeben werden, lässt Rückschlüsse auf das Verhalten der Krankheiten zu. Bei besonderer Betrachtung der ersten Runden ließ sich feststellen, dass in der allerersten Runde jede Krankheit im Schnitt lediglich 1 bis 3 Städte infiziert hat. Im Durchschnitt gibt es in einem Spiel, gerade am Anfang, auch nur 2 bis 3 verschiedene Krankheiten. Werden die Daten aus der ersten Runde nun einmal abgespeichert:

- Rhinonitis (Infektiosität: o / Mobilität: - / Dauer: - / Tödlichkeit: –)
  - Infizierte Städte: 1
- Moricillus (Infektiosität: - / Mobilität: o / Dauer: + / Tödlichkeit: +)
  - Infizierte Städte: 1

Nach einigen Runde, ohne dass Entscheidungen getroffen worden sind, lassen sich die folgenden Daten für die selben Krankheiten erheben:

- Rhinonitis (Infektiosität: o / Mobilität: - / Dauer: - / Tödlichkeit: –)
  - Infizierte Städte: 9
- Moricillus (Infektiosität: - / Mobilität: o / Dauer: + / Tödlichkeit: +)
  - Infizierte Städte: 173

Dazu muss gesagt werden, dass dieses Beispiel lediglich der Veranschaulichung dient, aber trotzdem auf Daten des ic20_Tools beruht. Wie in dem Beispiel zu sehen, sind um ein Vielfaches mehr Städte durch das Moricillus Pathogen infiziert worden, als vom Rhinonitis. Die Tests belegen, dass die unterschiedliche Ausbreitung der Pathogene auf die Mobilität zurückzuführen ist. Auch wenn in einer Startrunde mehrere Pathogene mit hoher Mobilität vorhanden sind, haben Sie jeweils eine hohe Ausbreitung, da eine Stadt auch von mehreren Pathogene infiziert sein kann. Natürlich gilt dabei zu berücksichtigen, dass ein Pathogen, welches eine Mobilität von 'o' hat, in einer Runde mit einem anderen Pathogen, welches eine Mobilität von '-' hat oftmals mehr infizierte Städte hat, als wenn das andere Pathogen anstatt '-' eine Mobilität von '++' hätte.

Die Ausbreitung einer Krankheit kann auch wieder abnehmen. Dies wäre der Fall, wenn eine Krankheit sich zwar schnell auf verschiedene Städte verbreiten kann, aber die Tödlichkeit überwiegt. Das hätte zur Folge, dass die Infizierten so schnell sterben, dass keine weiteren Personen mehr angesteckt werden können. Somit nimmt die Anzahl der infizierten Städte über den Verlauf des Spiels hinweg wieder ab. Beobachtbar bei der Krankheit N5-10, dessen Eigenschaften dies bewirken.

- N5-10 (Infektiosität: + / Mobilität: ++ / Dauer: o / Tödlichkeit: +)

3.3.3 Hypothesen

Aus der Datenstrukturanalyse: Output ergibt sich, dass aus dem Top Level des JSONs direkt der Spielzustand ausgelesen werden kann. Bei *win* und *loss* kann unser Algorithmus das Spiel beenden und die Qualität seiner Lösung berechnen. Bei einem lernenden Algorithmus kann hier nach ggf. eine Anpassung stattfinden. Bei *pending* befindet sich der Algorithmus in einem laufenden Spiel und hat die optimale Entscheidung zu treffen.

Der Eintrag *round* kann verwendet werden, um zu erkennen, in welcher Runde er sich befindet. Diese Information kann bei der Entwicklung von Medizin genutzt werden um zu berechnen wie lange es dauert bis die Medizin zur Verfügung steht. Außerdem besteht die Möglichkeit die weitere Dauer einer Krankheit in einer Stadt abzuschätzen. Zusätzlich mag es ein optimales Verhalten in Bezug auf die Punkte geben, welche in einer Runde zur Verfügung stehen. Der letzte Punkt welcher naheliegt ist, dass von der Runde abhängig eine bestimmte Anzahl an Zügen zu einem optimalen Verhalten führt.

Die *points* können verwendet werden, um die Anzahl an Entscheidungen in der Runde zu begrenzen. Die Wahrscheinlichkeit für oder gegen eine Entscheidung zu erhöhen wenn viele oder wenige Punkte zur Verfügung stehen. Die Dauer von Entscheidungen zu begrenzen, wenn Entscheidungen getroffen werden sollen welche eine variable Punkteanzahl besitzen.

Die Cities als assoziative Datenstruktur wird der Kern des Algorithmusses sein, da hier die meisten Informationen die Spielrelevant sind gehalten werden. Die Schlüssel für *economy*, *government*, *hygiene* und *awareness* können direkt auf numerische Werte gemappt werden. Die *popluation* ist in der Kombination mit *prevalence* der Stadt ein wichtiger Indikator um die Bedrohung durch die Krankheit in der Stadt festzustellen. Die Ausbreitung von Krankheiten wird wahrscheinlich in geographisch nahe Städte stattfinden. Für diese Betrachtung kann über *lattitude* und *longitude* die geographische Distanz zwischen Städten bestimmt werden. Außerdem muss bei der Distanzberechnung zwischen Städten beachtet werden, dass unter der Betrachtung der Longitude die Distanz zwischen -150 und 150 geringer ist als zwischen -150 und 0. In dem selben Maß sind große Distanzen in der Longitude durch kleine Distanzen in der Latitude abgeschwächt Ähnlich wird es sich mit den *connections* verhalten. Krankheiten werden sich über diese ausbreiten. Mit dem Unterschied, dass große Distanzen überbrückt werden. Hier bietet es sich ggf. an, bei großer Distanz Überbrückungen zu verhindern.
Bei den City Events sind *outbreak* und *bioTerrorism* wichtig, um Gegenmaßnahmen gegen die Krankheitsausbreitung, sowie Todeszahlen zu reduzieren. *antiVaccinationism* weisst wahrscheinlich auf eine reduzierte Wirkkraft von Impfstoffen hin. Die weiteren Events können verwendet werden, um Veränderungen in der selben Runde zu überwachen.

3.4 Anforderungen laut Aufgabenstellung

3.4.1 Funktionale Anforderungen

Zustandsloser Webservice

> „Berücksichtigen Sie die Umsetzung der Produktivversion Ihrer Software als zustandslosen Webservice von Anfang an in Ihrem Softwareentwurf. “

Nutzen von HTTP-POST-Requests

> „Ihre Software muss einen Webservice bereitstellen, über den die Eingabe der Spielzustände per HTTP-POST-Requests erfolgt.“

Verwenden vorgegebener JSON in UTF-8

> „Im Body der Requests sind Objekte in JSON 2 (UTF-8) wie in dem folgenden Beispiel enthalten.“

3.4.2 Definition der Optimierungsfunktion

Als feste Beschränkung gilt das Einhalten des vorgegeben Punkterahmens. Das bedeutet, es können nicht mehr Punkte in einem Spielzug ausgegeben werden als zur Verfügung stehen. Als weiche Beschränkung gilt eine Bevölkerungszahl von über 50% des Startwertes. Optimiert wird die Anzahl der Runden die benötigt werden, um das Spiel zum Abschluss zu bringen.

min	$sum_r(T_s)$		
$hard\ constraint$	$k(t)$	$\leq$	$points_r$
$soft\ constraint$	pop_r	$\geq$	$0.5 * pop_{start}$

mit	s	$:=$	Seed
	r	$:=$	Runde
	pop_r	$:=$	Population in Runde r
	pop_{start}	$:=$	Population zu Beginn des Spiels
	$points_r$	$:=$	Punkte in Runde r
	t	$:=$	Transition von Zustand $z1$ nach Zustand $z2$
	T_s	$:=$	Menge aufeinander folgender Transitionen beginned beim Zustand vom Seed s
	$k(t)$	$:=$	Kosten für Transition t
	$sum_r(s)$	$:=$	Benötigte Rundenanzahl für Seed zur Beendigung des Spiels

Abbildung 16: Optimierungsfunktion

4 Aufbau Entscheidungskomponente

4.1 Aufbau neuronales Netze

Im Computer wird ein Neuronales Netz abgebildet, wie ein Graph welcher einen Eingabevektor und einen Ausgabevektor besitzt (siehe Abbildung 2). Beide Vektoren sind über sogenannte Hidden Layer miteinander verbunden, dabei ist die Anzahl der Hiddenlayer nicht spezifiziert. Jedes Tupel ist mit jedem Tupel der benachbarten Layer verbunden. Aufgrund der Eingabe, die um das Netz möglichst performant lernen lassen zu können normiert (Werte zwischen: -1 < x < 1) sein sollten. Das Netz Initialisiert alle Kanten mit zufälligen Werten. Wenn das Netz ein Ergebnis aufgrund der Eingabeparameter berechnet hat muss im diese Entscheidung im Training bewertet werden. Dabei muss geprüft werden, wie gut die Entscheidung war und über eine Trainingsfunktion eine Rückmeldung an das Netz gegeben werden. Das Netz verändert dabei die Kantengewichte der einzelnen Tupel. Dieser Vorgang wird mehrere Male wiederholt und eine optimale Lösung angenähert.

4.1.1 Aktivierungsfunktion

Genau wie in der Natur ist auch ein Neuron welches Teil eines Neuronalen Netzes ist immer zu einem gewissen Grad aktiv bzw. gereizt. Der *Aktivierungsgrad* bestimmt, ob ein Neuron aktiviert wird und somit weitere Neuronen aktiviert (vgl. [Ert16]).
In unserer implementierung haben wir uns bei dem Ansatz an der Publikation von Fujimoto [FvHM18] orientiert und in den Hidden Layern eine lineare Aktivierungsfunktion verwendet (ReLu). Im Output Layer haben wir hingegen auf die von Fujimoto verwendete tanh Aktivierungsfunktion verzichtet um eine Skalierung der Ausgabewerte mit den realen Werten zu erfmöglichen. Dies führt dazu, dass im letzten Layer eine Funktion vergleichbar mit einer Regressions vorhanden ist. Aus der folgenden geringeren Differenzierung der Ausgabeaktivierung und damit einer unschärferen Trennung der zu wählenden Aktionen folgt eine höhere Dauer bei der Konvergenz des Netzes.

4.1.2 Initiale Konfiguration des Netzes

Wichtig ist, dass das Netz mit unterschiedlichen Kantengewichten initialisiert wird. Wenn alle Kanten beim Start dasselbe Gewicht hätten, dann würde das Netzt nicht lernen können. Die Gewichte sollten aus dem Intervall [-0.5;0.5] stammen, wobei alle Werte nahe an 0 ausgelassen werden sollten. Die Auswahl der Anzahl der Layer kann im vornherein nicht genau bestimmt werden. Es gilt dabei, desto mehr unsichtbare Layer vorhanden sind, desto länger dauert das lernen. Mit mehreren unsichtbaren Layern wird das Ergebnis ggf. besser angenähert. Deshalb gilt so viele unsichtbare Layer, wie nötig aber so wenig wie möglich [Ert16].

4.1.3 Q-Learning

Bei der Action Value Funktion $Q_{\Pi}(s,a)$ werden so viele Aktionen nacheinander abgearbeitet, bis ein Endzustand s_{τ} erreicht wurde. Das Verfahren zur Ermittlung der optimalen Funktion wird als **Q-Learning** bezeichnet. Ducrh Q-Learning kann immer eine optimale Funktion $Q^{*}_{\Pi}(s,a)$ ermittelt werden, egal, wie diese initialisiert wurde. Beim **Q-Learning** wird am Anfang mit zufälligen Werten gearbeitet. Da das System mit zufälligen Werten initialisiert

wurde ist der Weg des Systems auch zufällig. Mit zunehmender Zeit lernt das System Wege und kann dann durch eine Mischung aus bekannten (*greedy*) und explorativen Teilzügen *exploration* Stückweise von dem bekannten Verhalten in Richtung eines optimalen Verhaltens abweichen[Ert16].

4.1.4 Reward Function

Um den Agenten im Kontext von überwachtem lernen ein Feedback zu Entscheidungen zu geben, die dieser getroffen hat existiert eine Belohnungsfunktion (engl. reward). Mit dieser wird dem Agenten vermittelt, ob eine Entscheidung gut oder schlecht war. Dies wird auch als **Belohnung** bzw. **Strafe** gesehen [RN04].

Wir haben die Belohnungsfunktion wie folgt genutzt, wobei n die Anzahl der Runden wiedergibt die gespielt wurden:

$$\frac{1}{n} \tag{4}$$

Die Gleichung 4 wird genutzt wenn ein Spiel gewonnen wurde. Die Belohnung ist dabei höher wenn weniger Runden gebraucht wurden um ein Spiel zu gewinnen.

$$-\frac{1}{n} \tag{5}$$

Die Gleichung 5 wird genutzt wenn ein Spiel verloren wurde. Es wird immer eine Negative Belohnung vergeben. Bei einer Vielzahl an Runden die benötigt werden um zu verlieren ist die Belohnung kleiner.

$$\lim_{n \to \infty} Reward = 0 \tag{6}$$

Die Funktionen sind entstanden aus den in Unterunterabschnitt 3.4.2 vorgegebenen Kriterien.

Actor Critic

Mit dem Spielen in der Simulation würde in Deep Q Ansatz mit der Zeit Aktions-/Wert-Paare in einem endlichen Aktionsraum- und Ausgaberaum aktualisieren. Das Problem ist, dass der Ausgaberaum zu groß ist, um in diesen durch das Trainieren von Aktions-/Wert-Paare einen messbaren Erfolg in dem besseren Spielen in angemesserner Zeit zu erzielen. Er ist potentiell unendlich groß, da zu jedem beliebigem Zeitpunkt, bei beliebigem Seed ein Zug zu einer potentiell beliebigen Zustand führt. Um trotzdem in diesem unendlichen Ausgaberaum lernen zu können, wird die Actor/Critic-Methode verwendet, in dieser gibt es das Actor- und das Critic Neuronale Netz. Der Actor wählt mit dem Weltzustand die beste Aktion aus, die es durch das bisherig gelernte ausgibt. Der Critic lernt mit dem aktuellen Weltzustand und mit dem Reward der Environment, zu den zuvor ausgeführten Aktionen des Actors, einen Aktionsscore, der wiederum sich selbst und aber auch den Actor beeinflusst. So lernt der Actor potentiell nicht immer die Aktion zu nehmen, die greedy für den nächsten Zug der beste ist, sondern lernt über mehrere Aktionen hinweg den *besten Weg einzuschlagen* (siehe dazu Abbildung 17).

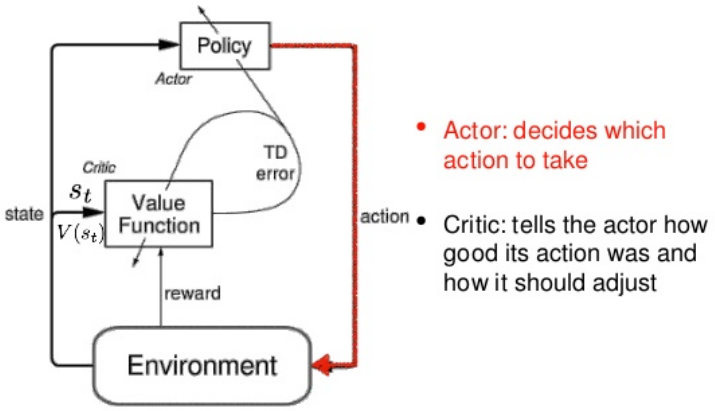

Abbildung 17: Actor-Critic [SB98]

Twin Deep Delayed Deterministic Policy Gradient

Ein aktuell viel verwendeter Ansatz zur Lösung von Entscheidungsproblemen ist der von Fujimoto [FvHM18] veröffentlichte Ansatz des Twin Deep Delayed Deterministic Policy Gradient. Unter anderem wurde dieser Ansatz zur Planung von Fahrwegen von Robotern verwendet[KHPK20]. Des weiteren wird er aktuelle wiederholt als State of the Art reinforcement Learning algorithmus bezeichnet (vgl. [HZAL18], [HZH+18], [JPS20]). In actor critic Umgebungen kann es genau wie bei Verstärkungsproblemen mit diskreten Aktionsräumen zu Wertüberschätzugen durch Fehler bei der Funktionsannäherung kommen [FvHM18].

4.2 Unterschiedliche Netztypen

Der erste Ansatz, den wir verfolgt haben, war alle (sinnvoll zu verwendenen) Größen des Request-JSON vom ic20_Tool Webclient in einen Eingabevektor für ein Neuronales Netz umzuwandeln. Bei diesem Vorgehen hat das Neuronale Netz allerdings

$$\begin{aligned}&\#_{\text{Städte}} * (\#_{\text{Eingabegrößen, Stadt}} + \#_{\text{Eingabegrößen, Krankheit}} * \#_{\text{Krankheiten}})\\ &= 258 * (10 + 6 * 10)\\ &= 18060 \text{ Eingabegrößen.}\end{aligned}$$

Neben der Größe des Netzes, war zu dem Zeitpunkt, als wir das große Neuronale Netz entwickelt haben, nicht klar, ob die Anzahl der Städte und Krankheiten gleich fix ist und wie hoch sie sind. Bspw. traten Krankheiten erst nach dem Verwenden von Zügen, verschieden von dem Beendungszug, auf. Daraufhin haben wir das großen Netz, im folgenden dargestellt, auf zwei kleinere aufgeteilt: Das Krankheits- und das Stadtnetz. Allerdings werden diese Netze jeweils für jede einzelne Krankheit bzw. Stadt durchlaufen. So muss die Größe der Städte und Krankheiten nicht bekannt sein oder fix sein. Außerdem trainieren die Netze ggf. schneller, da pro Spiel die Netze $\#_{\text{Städte}}$ oder $\#_{\text{Krankheiten}}$ oft durchlaufen werden. Beide

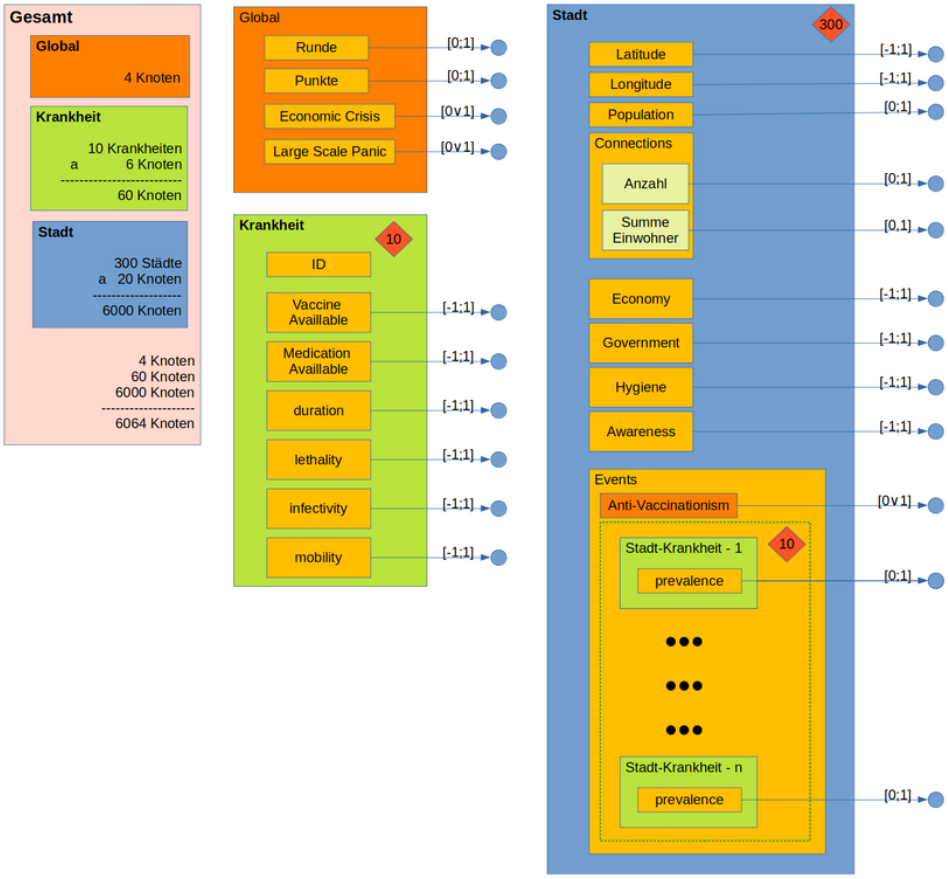

Abbildung 18: Stadt und Krankheitsnetz

Netze zusammen benötigen auch nur

$$\begin{aligned}&\#_{\text{Krankheit}} + \#_{\text{Eingabegrößen, Stadt}} + \#_{\text{Eingabegrößen, Stadt-Krankheit}}\\ &= 6 + 10 + 10\\ &= 26 \text{ Eingangsgrößen.}\end{aligned}$$

Im Vergleich verwenden die Netze im zweiten Ansatz also nur $\frac{26}{18060} \approx 0.14\%$ der Eingangsgrößen vom ersten Ansatz.

4.2.1 Ansatz1: Krankheitsnetz

Das Krankheitsnetz (siehe Abbildung 19) bekommen als Eingabegrößen die Eigenschaften der Krankheiten (duration, lethality, infectivity, mobility), die durch die Simulation (ic20_Tool) den Krankheiten zugeordnet sind. Weiterhin werden die Information, ob ein Impfstoff und ob ein Medikament für diese Krankheit existiert als weitere Eingabegrößen verwendet.

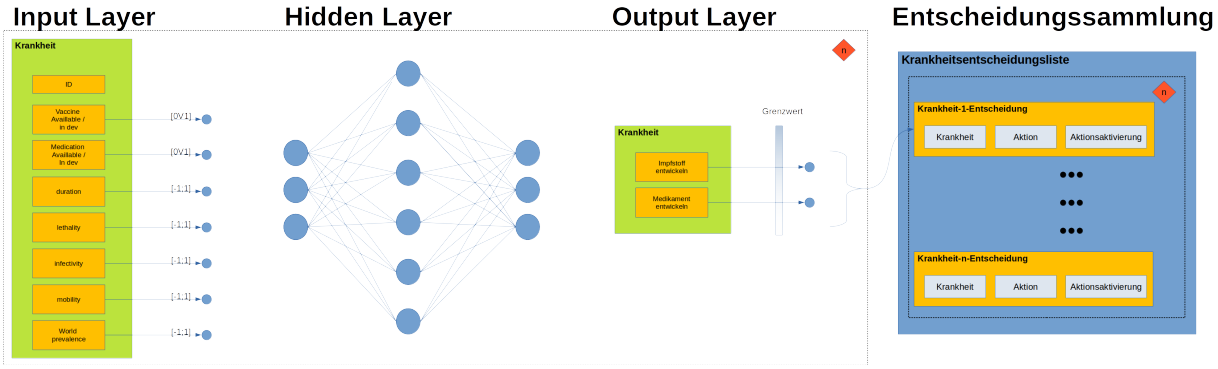

Abbildung 19: Aufbau des Krankheitsnetzes

4.2.2 Ansatz1: Stadtnetz

Das Stadtnetz (siehe Abbildung 20) bekommt die normierte (auf [0,1]) Anzahl der Einwohner zur Weltbevölkerung, Anzahl der Verbindungen auf x Verbindungen und Summe der Einwohner von jeder nächsten verbundenen Stadt auf die Weltbevölkerung. Weiterhin werden die Stadteigenschaften (Economy, Government, Hygiene, Awareness), die durch die Simulation (ic20_Tool) den Städten zugeordnet sind als Eingabegrößen verwendet. Die Information, in welcher Stadt welche Krankheit mit welcher Verbreitung in der Stadt ausgebrochen ist, wird mit Nicht ausgebrochen $= 0$ und Ausgebrochen $\neq 0$ in prevalence gekennzeichnet. Dabei entspricht Ausgebrochen $\neq 0$ in prevalence dem Wert der, durch die Krankheit getöteten, Einwohner in der Stadt normiert auf alle Einwohner der Stadt.

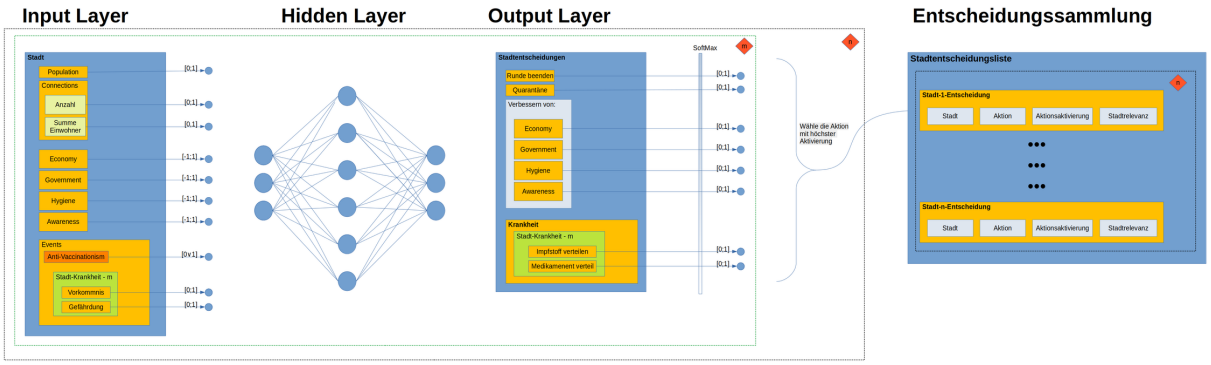

Abbildung 20: Aufbau des Stadtnetzes

4.2.3 Ansatz2: Kombiniertes Netz

Da durch das Stadtnetz für jede Stadt-Krankheitskombination trainiert wurde und in diesem deshalb auch für jede Krankheit die Stadt-Prävalenz als Eingabegröße verwendet wird, ist das Krankheitsnetz überflüssig geworden. Außerdem haben wir im Nachhinein festgestellt, dass die Netze zu viele Hidden Layer in dem getrennten Stadt-/Krankheitsnetzansatz hatten und deshalb weniger verwendet werden sollten. Kriesel hat in seiner Arbeit beschrieben, dass mit weniger Hidden Layern eine Lösung schneller erreicht werden kann, diese aber unter Umständen nicht so gut approximiert [Kri07]. Mit dieser Erkenntnis, haben wir dann also nochmal das Stadt- und Krankheitsnetz durch ein gemeinsames

Netz ersetzt, das Stadt- (blau), Krankheits- (grün) und gemeinsame (gelb) Eingabegrößen erhält. Außerdem haben wir die Anzahl der Hidden Layer heruntergesetzt. Weitere Unterschiede zu den Netzen davor ist, dass in dem gemeinsamen Netz die Stadt- und die Weltprävalenz als Eingabegröße (siehe Abbildung 21) einfließt und der Status, wie weit ein Medikament/Impfstoff entwickelt ist (vaccine/medication in development).
Als Ausgabegrößen (siehe Abbildung 21) hat das gemeinsame Netz die, in einigen Fällen gebrauchte, Anzahl an Runden und alle möglichen Spielzüge außer das Sperren von Flugverbindungen zwischen zwei Städten. Für jede Stadt-Krankheitskombination gibt das Netz also 12 Ausgabegrößen aus. Aus diesen effektiven $\#_{\text{Städte}} * \#_{\text{Krankheiten}} * \#_{\text{Züge}} = 258 * 10 * 11 = 28380$ möglichen Spielzüge mit $\#_{\text{Städte}} * \#_{\text{Krankheiten}} * \#_{\text{Ausgabegröße Runde}} = 258 * 10 * 1 = 2580$ potentiell benötigten Runden, wird dann über alle Städte und Krankheiten hinweg nach der jeweiligen Netzaktivierung (Wert des jeweiligen Ausgangsneurons) der Spielzug mit der höchsten Aktivierung gesucht, der Sinn macht: Bspw. muss ein Medikament/Impfstoff erst entwickelt werden, bevor es eingesetzt werden kann. Alle solche Spielzüge, die dabei als Unsinnig gewertet werden, werden mit dem gleichen Eingangs- wie Ausgangszustand (Eingabegrößen gleich Ausgabegrößen) mit einem schlechten Reward bestraft.

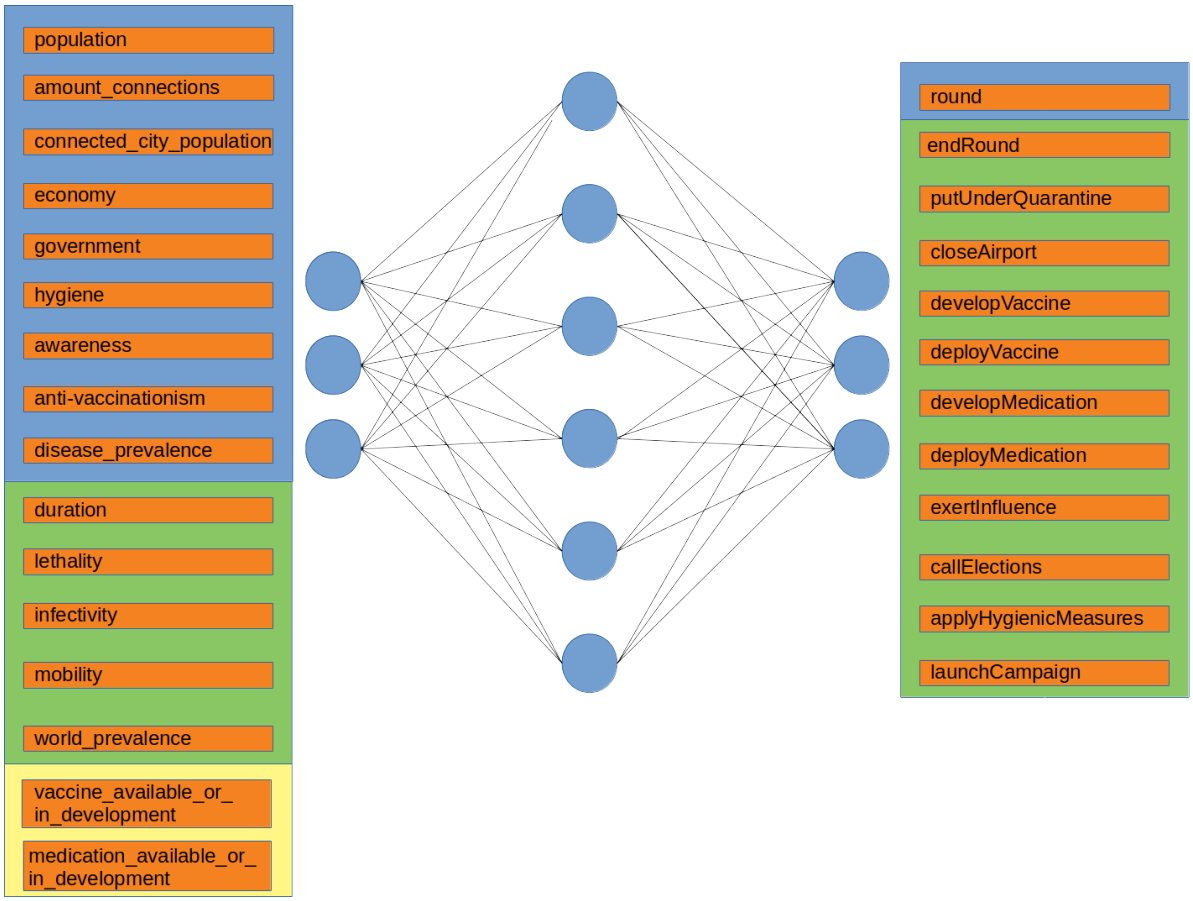

Abbildung 21: Gemeinsames Netz

Wenn die eigentliche Entscheidung ausgewählt wurde, wird diese an die Simulation gegeben, dabei werden gute Spielzüge mit einem höheren Reward als schlechte versehen. Mehrere Spielzüge werden gebuffered und dann nach einer gewissen Buffergröße dem Netz als Batch mit dem Reward zum Lernen gegeben.

5 Verwendete Technologien

In diesem Kapitel wird beschrieben, mit welchen Technologien die in Abschnitt 2 beschriebenen Konzepte umgesetzt wurden.

5.1 Containerisierung und dessen Orchestrierung

Docker[1] ist eine Containerisierungssoftware für Applikationen. D. h., dass es für jede Applikation eine eigene (Betriebssystem-)umgebung bereitstellt, die bei Bedarf für die Applikation angepasst werden kann, ohne dass sich mehrere Applikationen dabei in die Quere kommen. Die Beschreibung des Aufsetzens einer Applikation wird in einem sogenannten Dockerfile spezifiziert. Dieses wird cacheartig benutzt um verwendeten Code, Umgebungsveränderungen oder die Spezifikation des Startens von Diesten o. Ä. und das spezifizierte Base Image vorbereitend zusammen zu führen. Docker-Images widerrum können dann in einem Docker Container instanziiert werden, das dann die eigentliche Applikation mit der Umgebung, Code etc. ausführt. Ein großer Vorteil gegenüber herkömmlichen virtuellen Maschinen ist, dass Applikationen direkt auf dem Host System laufen und den Applikation nur die Umgebungsänderungen, nach der Spezifikation des Dockerfiles, von Docker untergeschoben werden. So wird der große Speicheroverhead von mehreren virtuellen Betriebssystemen und der Prozessausführungszeitoverhead auf virtuellen Umgebungen umgangen, da die Programme fast-nativ laufen.
Für die Verwaltung der Container auf den verschiedenen Maschinen, Netzwerkfreigaben, das mehrfache Instanziieren von Images in Containern, was jetzt teilweise von Hand oder mit bash Skripten umgesetzt wurde, könnte auch die Container-Orchestrierungssoftware Kubernetes oder Docker Swarm eingesetzt werden. Diese stellen, aus teilweise jahrelanger Erfahrung schöpfend, Lösungen für das Verwalten oder Skalieren von Systemen bereit, die auf Docker Containern aufbauen. Gerade dadurch, dass Docker Container oftmals keine Standalone Applikationen sind, sondern als Mircoservices mit anderen Applikationen über bspw. REST interagieren, müssen diese **gemeinsam** global verwaltet werden.

5.2 Versionsverwaltung

Für die Versionsverwaltung des Code haben wir zunächst das Gitlab[2] der unserer Uni verwendet. Wir haben einen eigenen Gitlab-Runner aufgesetzt und diesen als continous integration Teil einer CI/CD-Pipeline mit Gitlab verbunden. Zusätzlich haben wir das Wiki des Gitlabs benutzt, um Konzepte, Erkenntnisse etc. darzustellen. Allerdings gab es zwischenzeitlich technische Probleme, sodass wir auf GitHub[3] umgezogen sind. GitHub bietet unter Anderem den weiteren Vorteil, dass es direkt einen Runner mitbringt, der die Instruktionen aus ".github/workflows/*.yml" serverseitig bei einer spezifizierten Anweisung (bspw. einem push) ausführt. Mit einer automatischen Auslieferung auf Test- oder Produktivsysteme wäre das dann der continous delivery Teil einer CI/CD-Pipeline. Allerdings haben wir nur die Tests bei einem Push automatisiert durchlaufen lassen.

[1] `https://www.docker.com` - zuletzt Besucht 14.01.2020
[2] `https://about.gitlab.com` - zuletzt besucht 19.01.2020
[3] `https://github.com` - zuletzt besucht 10.01.2020

5.3 Programmiersprache mit Frameworks

Unser initiales Konzept war es, künstliche Intelligenz das Problem lösen zu lassen. So haben wir unsere Lösung in Python3 umgesetzt, da in dem Bereich Maschine Learning viele Lösungen in Python umgesetzt sind und daher die Annahme war, dass das Tooling und dessen Integration abgehangen, relativ gut zu bedienen und einigermaßen getestet ist. Desweiteren bot sich die Möglichkeit mit einer neuen Programmiersprache zu arbeiten. Als Entwicklungsumgebung haben wir PyCharm von JetBrains eingesetzt, da diese eine gute Integrationen mit bspw. git[4], aber auch numpy und scipy hat und einen hilfreichen Editor besitzt. Als Maschine Learning Framwork haben wir pyTorch eingesetzt, damit wir konzeptionell gleiche Operationen auf verwendeten Neuronalen Netzen nicht selber implementieren müssen. Flask wird benutzt um die Webserviceschnittstelle zu bedienen und eine bereitzustellen.

[4] `https://git-scm.com` - zuletzt besucht 10.01.2020

6 Deployment

In diesem Kapitel wird das genutzte Deployment vorgestellt. Dabei wird erklärt, wie die Verteilung für die Trainingsmodelle automatisiert wurde und wie die Container erstellt wurden. Durch die Abteilung Rechner- und Netzbetrieb Informatik[5] (ARBI) wurden uns kostenfrei Maschinen zur Verfügung gestellt, auf der unterschiedliche Modelle trainiert werden konnten. Die ARBI ist für die Organisation und Verwaltung des Rechnerbetriebs an der Carl von Ossietzky Universität Oldenburg zuständig. Dabei wurde eine virtuelle Maschine konfiguriert, die auf allen zur Verfügung stehenden Rechnern verteilt wurde.

6.1 Zugriff

Auf allen virtuellen Maschinen haben wir Zugriff per SSH bekommen. Mit bash Skripten haben wir automatisiert Änderungen auf den Maschinen vorgenommen, falls solche nach dem initialen Verteilen nötig waren. Dafür wurde auf jeder Maschine der Public Key hinterlegt und für jede Maschine eine Konfiguration in der SSH config eingetragen, damit einfach mit *ssh MASCHINEN_NAME* auf diese zugegriffen werden kann, ohne ein Passwort einzugeben.

6.2 Monitoring

Für das Überwachen der Rechner, der Speicherbelegung und der Prozesse bzw. Docker Container-Zustände haben wir ein bash Skript (siehe Quellcode 4) geschrieben, das durch einen cronjob automatisch nach einer gewissen Zeit aufgerufen wird. Dieses Skript schickt eine E-Mail mit der aktuellen Speicherbelegung und eine Prozesstabelle mit den ersten 15 Prozessen geordnet nach CPU-Verbrauch (siehe als Beispiel Quellcode 5).

```
#!/bin/bash
# Server und Email holen
server=$(cat /home/infocup/server)
emails=$(cat /home/infocup/config/emails | sed ':a;N;$!ba;s/\n/ /g')
# Speicher und Prozessbelegung holen
{ \
df -h ;\
ps aux k-pcpu | head -n15;\
}\
# Versenden mit einer E-Mail von GMail per sendEmail
| sendEmail -o tls=yes -f NAME@gmail.com -t $emails \
-s smtp.gmail.com:587 -xu NAME@gmail.com -xp PASSWORD \
-u "Status $server" > /dev/null
```

Quellcode 4: Monitoring Skript

```
Filesystem     Size  Used Avail Use% Mounted on
udev           7.8G     0  7.8G   0% /dev
tmpfs          1.6G  896K  1.6G   1% /run
```

[5] `https://uol.de/informatik/department/arbi` - zuletzt besucht 15.01.2020

```
/dev/sda2       30G   14G   15G  50% /
tmpfs          7.9G     0  7.9G   0% /dev/shm
tmpfs          5.0M     0  5.0M   0% /run/lock
tmpfs          7.9G     0  7.9G   0% /sys/fs/cgroup
/dev/loop0      55M   55M     0 100% /snap/lxd/12211
/dev/loop2      55M   55M     0 100% /snap/lxd/12631
/dev/loop1      90M   90M     0 100% /snap/core/8213
/dev/loop4      90M   90M     0 100% /snap/core/8268
tmpfs          1.6G     0  1.6G   0% /run/user/1000
/dev/loop3     121M  121M     0 100% /snap/docker/423
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      2515 86.5 1.1 1790840 187296 ?    Sl   15:27   2:59 python3
    ./src/TorchSlave.py -ip http://HOSTNAME:IP
root      6971 84.8 1.1 1744212 184448 ?    Sl   15:29   1:18 python3
    ./src/TorchSlave.py -ip http://HOSTNAME:IP
root      9098 76.0 1.1 1743700 183452 ?    Sl   15:30   0:24 python3
    ./src/TorchSlave.py -ip http://HOSTNAME:IP
root      8741 75.4 1.1 1741908 181552 ?    Sl   15:29   0:30 python3
    ./src/TorchSlave.py -ip http://HOSTNAME:IP
root     10314 11.0 0.0 775052 15768 ?     Sl   15:30   0:00 ./ic20_linux -o /dev/null
root     10211 10.5 0.1 849040 17344 ?     Sl   15:30   0:00 ./ic20_linux -o /dev/null
root     10198 10.0 0.0 775308 15884 ?     Sl   15:30   0:00 ./ic20_linux -o /dev/null
root     10222  9.2  0.0 849040 15520 ?     Sl   15:30   0:00 ./ic20_linux -o /dev/null
root      2177  6.8  0.6 1132284 104036 ?   Ssl  15:11   1:19 dockerd -G docker
    --exec-root=/var/snap/docker/423/run/docker
    --data-root=/var/snap/docker/common/var-lib-docker
    --pidfile=/var/snap/docker/423/run/docker.pid
    --config-file=/var/snap/docker/423/config/daemon.json
root       718  1.9  0.2 1367448 33100 ?    Ssl  15:02   0:32 /usr/lib/snapd/snapd
root      2265  0.4  0.2 767352 46444 ?     Ssl  15:11   0:04 containerd --config
    /var/snap/docker/423/run/docker/containerd/containerd.toml --log-level error
root      9054  0.4  0.0   2384   764 ?       Ss   15:30   0:00 /bin/sh -c ./start.sh
root      8697  0.3  0.0   2384   764 ?       Ss   15:29   0:00 /bin/sh -c ./start.sh
root         1  0.1  0.0 102524 11844 ?     Ss   15:02   0:02 /sbin/init
```

Quellcode 5: Monitoring Skript Output per E-Mail

6.3 Trainingsdeployment

Um die verwendeten Neuronalen Netze zu trainieren, haben wir ein Trainingsdeployment aufgesetzt. Damit das Trainieren der eigentlichen Netze schneller geht, wird das Training auf mehrere Netze verteilt und diese später wieder zusammen gemerged. Das Trainingsdeployment besteht daraus, dass wir auf einer virtuellen Maschine auf einem Server unserer Uni den Webserver im Trainingsmodus gestartet haben und den Webclient von der GI wiederholend ausgeführt haben. Dieses Setup wurde $\#_{Kerne}$-oft aufgesetzt und gestartet, in unserem Fall haben wir somit das Setup 48 mal aufgesetzt ($\#_{Kerne} = 4 \ \frac{Kerne}{Maschine} \ * \ 12 \ Maschinen = 48 \ Kerne$).
Das oben beschriebene Setup sind die **slaves**. Diese trainieren Neuronale Netze mit beliebig initialisierten Werten. Auch der ic20_Tool-Webclient hat durch den Zeitstempel als Seed nicht immer gleiche Weltzustände sowie Eventabfolgen. Diese Beliebigkeitsfaktoren führen dazu, dass die Neuronalen Netze unterschiedlich trainieren und somit später auch unterschiedlich sind. Die Netze auf den slaves werden nach einer gewissen Zeit zu einem

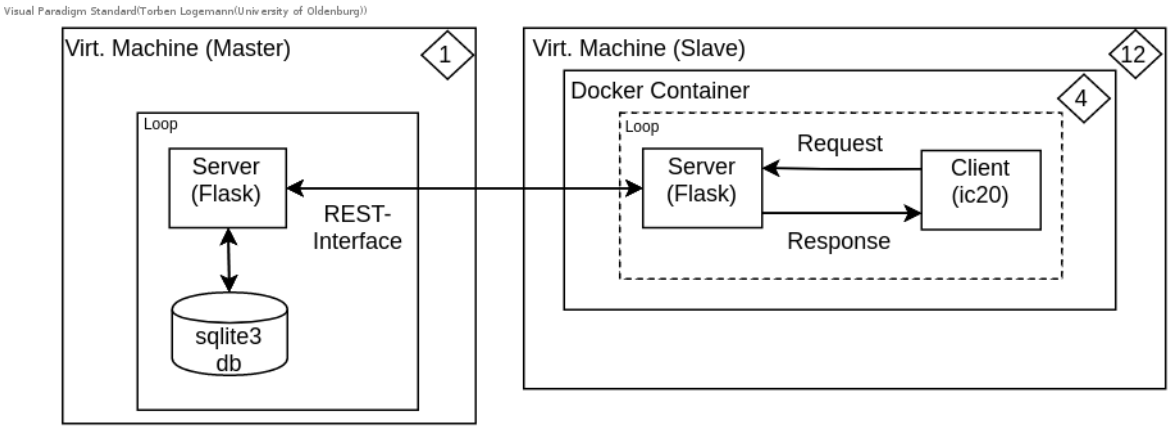

Abbildung 22: Block Diagramm zum Trainingsdeployment

master-Server geschickt, der diese merged, speichert und dann zurück auf die slaves verteilt.

6.4 Ausführungsdeployment

Um ein Netz nicht mehr trainieren zu lassen und das Programm als zustandslosen Webserver bereitzustellen, wird aus der sqlite3 Datenbank des Masters das PyTorch-Modell als PTH-Archive extrahiert und dem solutions-Projekt unter pytorch_models hinzugefügt. Das solutions-Projet stellt neben der Nicht-Trainingsfunktion noch die Visualisierung bereit.

6.5 Verwaltung der Software auf den Trainingsmaschinen

Für das eigentliche Verteilen der Software auf die Slaves, die dann Netze lernen lassen soll, die trainierten Netze dann zur Master-Maschine schicken soll, die diese dann merged, könnte die Software theoretisch per Hand auf alle Maschinen mit bspw. *scp* verteilt werden und gestartet werden, jedoch ist dies fehleranfällig, mühsam und dauert ggf. sehr lange. Außerdem müsste dafür nochmal extra Verwaltungs-Code geschrieben und verwaltet werden, die die Slaves bspw. bei einem Fehler neustartet.
Deshalb haben wir uns im ersten Ansatz dafür entschieden mit bash Skripten das git repository automatisch auf allen Traingsmaschinen clonen zu lassen und mittels einer *SystemD-Unit* die slaves starten zu lassen. Eine *SystemD-Unit* kann von Haus aus Daemons, also Services im Hintergrund, starten, die auch nach Anweisung bei einem fehlerhaften Abbruch ($errno \neq 0$) automatisch neugestartet werden können. Außerdem kann bei einem fehlerhaften Abbruch eines Daemons ebenfalls eine automatische Mail (siehe Unterabschnitt 6.2) verschickt werden. Diese Methode ist jedoch immer noch relativ fehleranfällig, da bspw. die git repositorys u. U. auf den Maschinen verwaltet werden müssen.
Um ein noch zuverlässigeres Deployment zu bekommen, haben wir uns dann dafür entschieden die Software der Subpackages *master*, *slave* (siehe Abbildung 23) und *solution* per Docker-Images über DockerHub auszuliefern (siehe dazu auch Unterabschnitt 5.1). So wird nach dem pushen von Code im git repository direkt ein Docker-Image gebaut, das automatisch zu DockerHub geuploaded. Dieses könnte dann durch ein kontinuierliches Ausliefern über eine CI/CD-Pipeline direkt auf die Maschinen verteilt werden, sodass auf

allen Maschinen immer der gleiche, neuste Softwarestand läuft. Den letzten Schritt des kontinuierlichen Auslieferns haben wir jedoch aus zeitlichen Gründen nicht mehr aufgesetzt. Allerdings können die neusten Docker-Images direkt von DockerHub mit dem “docker pull”-Befehl bezogen werden.

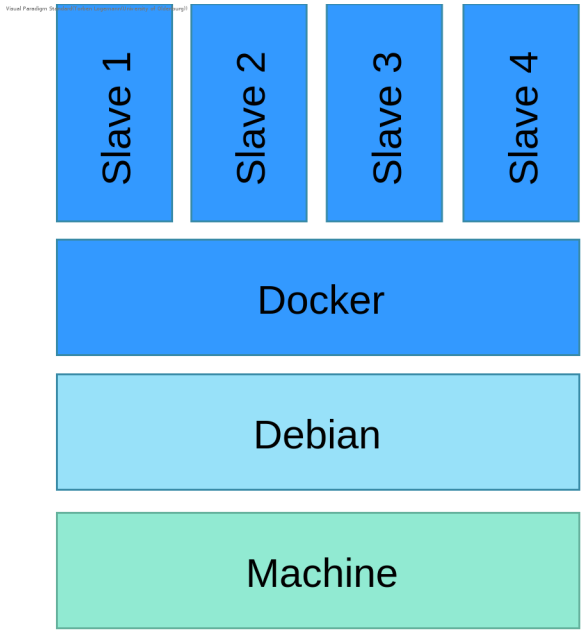

Abbildung 23: Aufbau unsers Docker Deployments (in Anlehnung an [Doc20])

7 Visualisierung

Um die Zustände der Spielrunden grafisch darzustellen, wurde eine webbasierte Visualisierung entwickelt. Die Stadtbevölkerungen werden proportional zueinander in der Größe eines Kreises um die Stadt auf einer Google Maps [6] Karte dargestellt (siehe dazu Abbildung 25). Wenn der Webclient den Weltzustand in einer Runde an den Webserver schickt, wird dieser als 'aktueller' Weltzustand gespeichert. Dieser Weltzustand wird dann als Response einer GET-Anfrage der Visualisierung an die Visualisierung geschickt. Aus den Response-Daten im JSON-Format werden dann zunächst die relevanten Daten extrahiert und über die Google Maps JavaScript API gerendert. Dadurch, dass die Anfrage über AJAX asynchron ist, kann nach dem Erhalten der Antwort zyklisch direkt die nächste gestellt werden, sodass das AJAX Long Polling-Verfahren realisiert ist. Die Asynchronität bietet das nicht blockieren der GUI, sodass die Google Maps Karte ganz normal benutzt werden kann (bspw. Zoomen). Dadurch, dass der Webserver solange wartet, bis ein neuer Weltzustand vom Webclient erhalten worden ist und die Visualierung erst nach Erhalt der Antwort eine neue Anfrage stellt, ist kein "normales", ressourcenverschwendenden Polling verwendet worden.

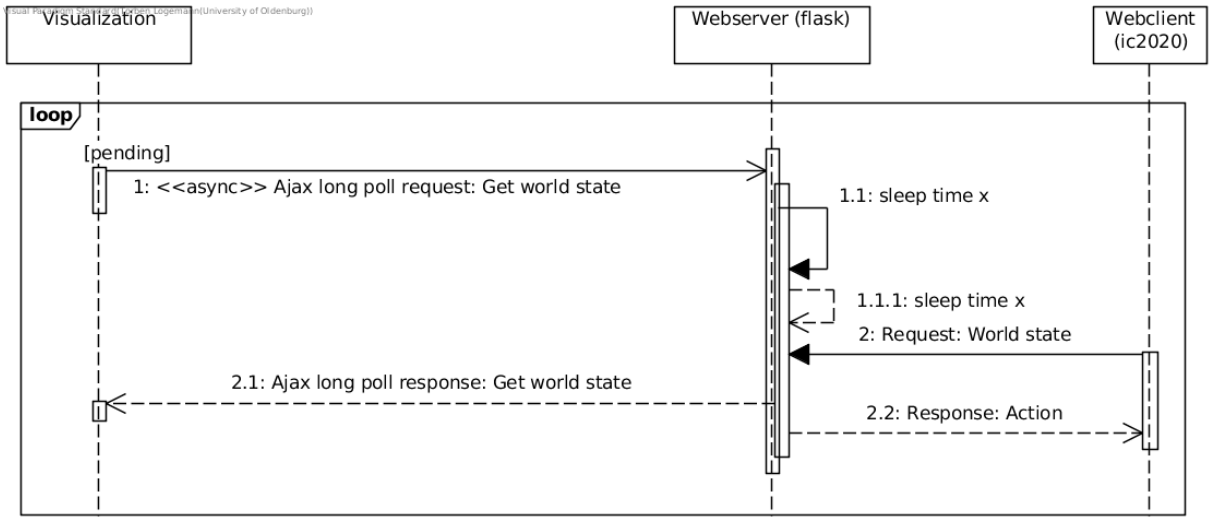

Abbildung 24: Sequenzdiagramm des Datenaustausches zwischen Visualisierung, Webserver und -client

[6] `https://developers.google.com/maps/documentation/javascript/tutorial` - zuletzt besucht 15.01.2020

In Abbildung 25 ist die Visualisierung im Browser zu sehen. Dabei wurde die Google Maps API verwendet. Auf der Map werden automatisch alle Cicle in Abhängigkeit der Einwohnerzahl gesetzt. Im Verlauf eines Spiels verändern sich die Kreise und werden kleiner, wenn die Einwohnerzahl einer Stadt sinkt.

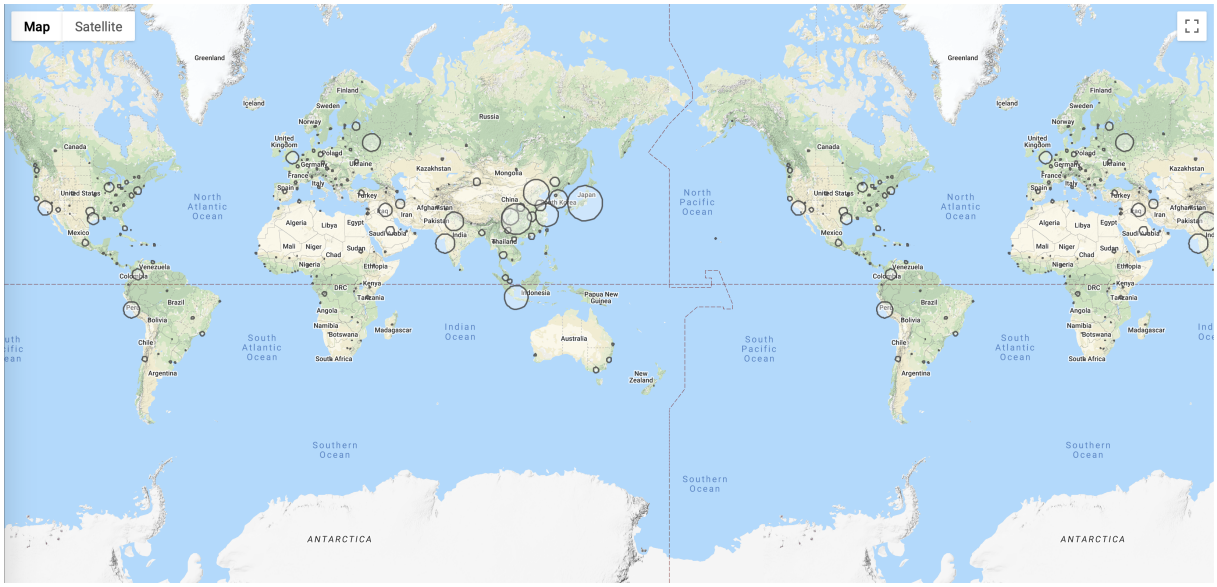

Abbildung 25: Karte von der Visualisierung

Beim Starten des Dockercontainers können wie in Quellcode 3 beschrieben Parameter übergeben werden, um die Visualisierung zu starten.

8 Testdokumentation

8.1 Aufbau

Der Aufbau der Teststruktur spiegelt die Struktur des Projektes wider. Dies ist daran erkennbar, dass es im Grundverzeichnis des Projektes, ein separater Ordner für die Tests vorhanden ist. In diesem Ordner befinden sich dann die selben Ordner, wie im eigentlichen Projekt. Ein kleiner Unterschied in der Namensnennung der Ordner, sowie der Dateien, ist ein kleingeschriebenes 'test_' vor dem eigentlichen Dateinamen.

$$d3t\_agent \rightarrow test\_d3t\_agent \tag{7}$$

$$TorchAgent.py \rightarrow test\_TorchAgent.py \tag{8}$$

Der Grund für die Konvention, ist auf die 'unittest' Bibliothek zurückzuführen, denn alle Tests müssen mit 'test_' beginnen. Daraus ist erkennbar, dass wir die Bibliothek 'unittest' verwenden und deren Konventionen für unser eigenes Projekt übernommen haben.
Des Weiteren lässt sich sagen, dass der Aufbau innerhalb der Tests äquivalent ist. Jede Testklasse besitzt eine kurze Beschreibung, sowie jeder Test eine Beschreibung enthält, was dieser Test testet.

8.2 Umsetzung der Tests

Innerhalb des Projektes wurden Integrationstests geschrieben. Diese Tests sollen dazu dienen, eine Methode auf deren Richtigkeit zu überprüfen. Der Input, sowie der erwartete Output ist bekannt und lässt sich somit einfach überprüfen.
In Python steht einem dazu, die im vorherigen Abschnitt erwähnte Bibliothek 'unittest'. Diese Bibliothek ermöglicht einem sein gewünschtes Ergebnis auf 'True' oder 'False' zu testen. Die dazu verwendeten Methoden sind

$$assertTrue(Wert) \tag{9}$$

$$assertFalse(Wert) \tag{10}$$

9 Arbeitsorganisation

In diesem Kapitel wird die Arbeitsorganisation des Teams vorgestellt. Bei der Arbeitsorganisation wurde bewusst auf eine agile Methode gesetzt. Dabei wurde nach dem Vorgehensmodell **Scrum** gearbeitet und entwickelt. Da nicht immer alle Studierenden zum selben Zeitpunkt Kapazitäten haben, um an dem Projekt zu arbeiten, wurden wöchentliche Treffen geplant um den Fortschritt der einzelnen Aufgaben zu besprechen. Bei den Treffen wurden dann zusammen neue Aufgaben definiert und verteilt. Ein Scrummaster wurde bewusst nicht festgelegt sondern die Aufgabe innerhalb des Teams geteilt.

9.1 Kommunikation

Bei der Kommunikation wurden verschiedene Medien eingesetzt. Um die Kommunikation zum Dozenten oder dem Kunden (der Gesellschaft für Informatik) abzuwickeln wurden E-Mails versendet, die alle anderen Gruppenmitglieder als Kopie empfangen haben. Interne Nachrichten wurden über das Medium Slack versendet, weil dieses Medium in vielen Unternehmen eingesetzt wird und einen großen Umfang an Funktionen bietet. Dabei ist z.B. vom Vorteil, dass Räume für unterschiedliche Themenbereiche eröffnet werden können. Einfachere Messenger hätten diese Funktionen nicht zur Verfügung gestellt und es wäre zu schnell sehr unübersichtlich geworden.

9.2 Planung

Es wurden anfangs vier Meilensteine definiert die zu festgelegten Zeitpunkten erreicht werden sollten:

1. Abschluss der Planungsphase
2. Abschluss des Prototypen
3. Abschluss des Programmes um Training beginnen zu können
4. Abgabe fertiges Programm

Die Meilensteine wurden innerhalb der Planungsphase festgelegt.
Am Anfang wurde ein grober Plan erstellt, zu welchen Zeitpunkten welche Funktionen zur Verfügung stehen müssen, um den Abgabetermin einhalten zu können. Dabei wurde beschlossen, dass der erste Meilenstein erledigt ist, wenn die Planungsphase abgeschlossen ist. In der Planungsphase wurde fokussiert, die Anforderungen an das Endprodukt zu definieren. Nachdem die Anforderungen definiert waren, wurden die ersten Überlegungen zur Umsetzung getätigt. Dabei fiel schnell der Entschluss, nicht mit Heuristiken zu arbeiten sondern etwas auszuprobieren, das bisher im Studium noch nicht thematisiert wurde, um einen großen persönlichen Lerneffekt zu erzielen und im Idealfall eine sehr gute Lösung zu erhalten. Nachdem die Technologien und der Lösungsansatz klar war, wurde im Meilensteinplan berücksichtigt, dass das Modell Zeit zum Trainieren benötigt.

9.2.1 Prototypen

In der zweiten Phase des Projektes wurde ein Prototyp entwickelt. Die zuvor festgelegten Anforderungen wurden dabei berücksichtigt. Der Prototyp wurde entwickelt um zu prüfen, ob die Anforderungen mit dem Lösungsansatz überhaupt lösbar ist und es möglich ist in kurzer Zeit ein Neuronales Netz zu entwickeln. Dabei wurde verifiziert, dass ausreichend Kenntnisse erworben werden konnten, um ein Netz zu implementieren.

9.2.2 Fertiges Programm

In der dritten Phase des Projektes wurde der Prototyp weiterentwickelt und optimiert. Währenddessen wurde geprüft, ob und wie viele Iterationen trainiert werden können. Dabei wurde festgestellt, dass die Iterationen nicht ausreichen werden. Es wurde entschlossen, dass mehrere Modelle parallel trainiert werden und die Modelle dann zusammengezogen werden. Dazu musste möglichst viel Rechenleistung angefordert werden, welche wir von der ARBI (siehe Abschnitt 6) zur Verfügung gestellt bekommen haben. Dazu musste die Verteilung der einzelnen Modelle automatisiert werden.

9.2.3 Abgabe

Für den letzten Meilenstein musste die Dokumentation komplett ausgearbeitet werden und die trainierten Modelle gesichert werden. Desweiteren wurde das Projekt aufgeräumt und es wurde auf Vollständigkeit geprüft. Der geschriebene Quellcode wurde um Tests erweitert (siehe Abschnitt 8). Es wurden die Anforderungen geprüft und sichergestellt, dass die Anforderungen umgesetzt wurden. Bei noch nicht umgesetzten Anforderungen wurde geprüft welche Anforderungen noch umsetzbar sind. Dabei war das Ziel nur kleinere Anforderungen, die eher unwichtig erscheinen unbearbeitet zu lassen.

9.3 GitLab issue board

Um einzelne Aufgaben festzulegen wurde bei GitLab das issue board genutzt. Hier konnten Aufgaben angelegt werden und wie in Scrum üblich als Backlog genutzt werden. Gruppenmitglieder, welche keine Aufgabe hatten konnten einen neue Aufgabe in die Bearbeitung transferieren. Das hilft der gesamten Gruppe dabei den Überblick darüber zu behalten welche Person welche Aufgabe hat. Einzelne Aufgaben konnten einem Meilenstein untergeordnet werden. Das hat dabei geholfen einen Fortschritt der Meilensteine beobachten zu können. Die einzelnen Aufgaben wurden im Backlog konkret ausformuliert und mit Informationen versehen. Jede Aufgabe wurde mit einem dateum versehen, an dem diese abgearbeitet sein muss. Projektmanagement in einer solchen Form hilft dabei die begrenzten Ressourcen effizient einzusetzen.

10 Auswertung

Die Besonderheit bei einem lernenden Agenten ist, dass er sich über die Zeit fortentwickelt. Um diese Entwicklung zu überwachen wurde zu Beginn ein Benchmark festgelegt. Als Benchmark wurde mit fest definierten Seeds und einer Explorationsrate von 0 (*greedy*-Verhalten) das Modell getestet um eine Reproduzierbarkeit zu gewährleisten. Mehr Seeds hätten eine besser Evaluation des Models ermöglicht. Gegen mehr Seeds hat die Dauer für den Evaluationsprozess gesprochen. Einen zeitlich akzeptablen Umfang für die Testdurchläufe hatten die Seeds 1-30.
Auf Basis dieses Benchmarkes haben wir regelmäßig das Referenzmodell aus der Datenbank geladen und evaluiert. Die Ergebnisse der Evaluation haben bis zum 18.1. jedoch regelmäßig gezeigt, dass das Modell nicht sinnvoll lernt. Bei der Überprüfung der Kantengewichte konnte erkannt werden, dass Veränderungen im Modell stattgefunden haben. Die Spielzüge änderten sich auch, so hat der Agent nach zwei Wochen ausschließlich den Zug *applyHygienicMeasures* durchgeführt bis er keine Punkte mehr zur Verfügung hatte. Mitte Januar hat sich das Verhalten des Agenten geändert, er führte regelmäßig *excertInfluence* aus. In den letzten zwei Tagen hat der Agent jedoch einige Runden mehr gewonnen und etwas langsamer verloren(siehe Abbildung 26. Der Abfall am Anfang kann darauf zurück zu führen sein, dass die trainierte Policy bei den verwendeten Benchmarkwerten nicht passend ist. Auch ist der Absolutbetrag der Differenz bei Betrachtung des Wertebereiches von -1 bis 1 gering. Die Policy sollte sich jedoch durch weiteres Training verbessern. Der kleine Anstieg der letzten Tage lässt darauf schließen, dass sich der Agent noch im Lernprozess befindet und der Prozess sehr langsam stattfindet. Ein divergierendes Verhalten ist jedoch nicht auszuschließen. Wir werden auch nach der Abgabe das Modell weitertrainieren

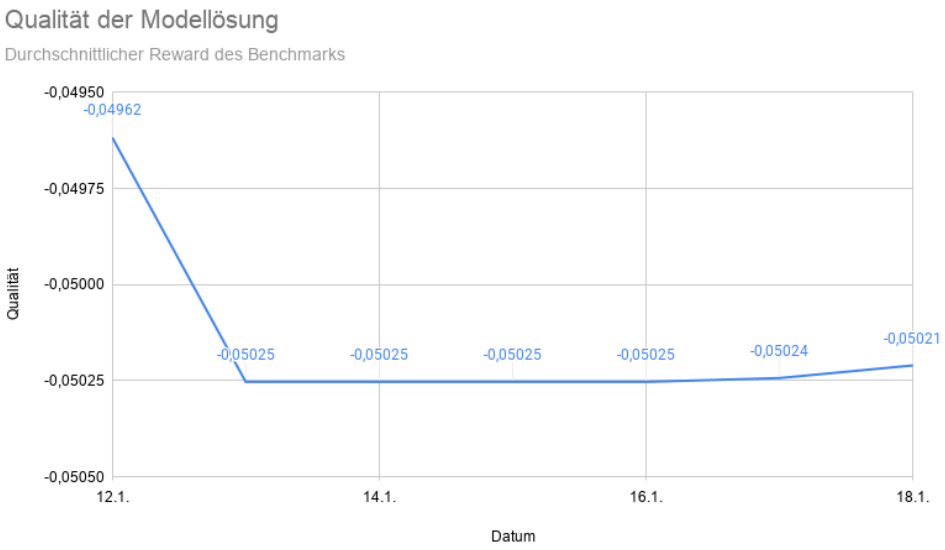

Abbildung 26: Entwicklung des Modells

um zu prüfen, ob der Lösungsansatz zu verwerfen ist, oder ob die Trainingszeiten zu gering waren. Ein weiterer Ansatz für die Optimierung des neuronalen Netzes ist das Hyperparametertuning.

11 Fazit und Ausblick

11.1 Fazit

Am Anfang des Projektes standen die in den Grundlagen aufgegriffenen Lösungsstrategien zur Auswahl. Nach ausgiebiger Betrachtung der verschiedenen Möglichkeiten fiel die Entscheidung darauf ein neuronales Netz zu implementieren Daraus folgte, dass wir uns mit dem Oberbegriff *Künstliche Intelligenz* auseinander setzen mussten, obwohl es nicht absehbar war, ob die Zeit ausreichen wird um ein gut performendes Modell zu trainieren. Im Laufe des Projekts haben wir uns mit verschiedenen Werkzeugen und Technologien auseinandergesetzt, von denen uns viele vorher nicht bekannt waren. Ein Großteil der, uns zur Verfügung stehenden, Zeit wurde auf die Einarbeitung in diese Technologien und Werkzeuge investiert, da die Inhalte in unserem Studium bisher nicht thematisiert wurden. Somit ist der persönliche Lernzuwachs im Bereich der neuronalen Netze immens und geht in vielen Bereichen über das hinaus, was Teil der Studieninhalte ist. Ein weiterer interessanter Punkt war die Umsetzung des Deployments, welches in Abschnitt 6 beschrieben ist. Dabei wurde anfangs viel mit Skripten gearbeitet und das Deployment immer weiter automatisiert. Die Ergebnisse des Trainings über die Weihnachtsferien haben keine Konvergenz des Modelles gezeigt. Aus diesem Grund wurden nach Weihnachten weitere Anpassungen an dem Code für die Modelle vorgenommen.
Auch aufgrund der Einarbeitung in die uns neuen Werkzeuge und Technologien, hat die Zeit nicht gereicht, um ein Modell zu trainieren, das gute Ergebnisse liefert. Eine Weiterarbeit an dem Training des Modells wäre für uns jedoch eine interessante Möglichkeit, uns weiterhin mit neuronalen Netzen auseinanderzusetzen.
Abschließend würden wir den Wettbewerb weiterempfehlen und auch selber noch einmal teilnehmen. Die Organisation war super und die Antworten egal auf welchem Kanal, ob GitHub oder E-Mail kamen sehr schnell.

11.2 Ausblick

Ein weiterer Punkt, der noch verbessert werden kann, ist die Visualisierung. Diese könnte interaktiv gestaltet werden. Damit wäre es möglich über die Visualisierung eine Aktion auszuwählen und somit das ic20 Tool händisch zu bedienen. Für eine bessere und zeiteffizientere Lösung der Aufgabe des Wettbewerbes wäre die Arbeit mit Heuristiken zielführender gewesen, da schon einfache Entscheidungen dazu führen, dass eine hohe Gewinnrate erzielt wird. Mit unserem Lösungsansatz konnten letztendlich aufgrund der zeitlichen Einschränkungen keine befriedigenden Ergebnisse erzielt werden, da das Trainieren des Modells einen erheblichen Zeitaufwand bedeutet. Aber mit den jetzt erlernten Technologien, ist es möglich, bei einem ähnlichem Projekt in Zukunft schneller anfangen zu können. Das Deployment könnte direkt am Anfang des Projektes automatisiert werden, was einen enormen Zeitvorteil bringen würde. Die längerfristigere Evaluation der trainierten Modelle würde unsere Modelle noch wasserdichter gegenüber potentiellen Bugs im Code und das Erlernen falscher Spielzugkonzepte machen.

12 Anhang

Der Anhang wird über folgenden QR-Code online zur Verfügung gestellt.

Literatur

[CF12] Schawel Christian and Billing Fabian. Entscheidungsbaum: (problemanalyse-tools). In *Top 100 Management Tools: Das wichtigste Buch eines Managers Von ABC-Analyse bis Zielvereinbarung*, pages 92–95. Gabler Verlag, Wiesbaden, 4., überarb. aufl. 2012. edition, 2012.

[Doc20] Inc. Docker. What is a container?, 2020. Accessed: 2020-01-19.

[Ert16] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz : eine praxisorientierte Einführung*. Computational Intelligence. Springer Vieweg, Wiesbaden, 4., überarbeitete auflage. edition, 2016.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *ArXiv*, abs/1802.09477, 2018.

[Han92] Hans-Jürgen Zimmermann. *Methoden und Modelle des Operations Research*, pages 258–259. Vieweg+Teubner Verlag, Wiesbaden, 1992.

[Har07] Steffen Harbich. Einführung genetischer algorithmen mit anwendungsbeispiel. *Popul.(English Ed.*, page 11, 2007.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[HZH+18] Tuomas Haarnoja, Aurick Zhou, Sehoon Ha, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *CoRR*, abs/1812.11103, 2018.

[JPS20] Whiyoung Jung, Giseung Park, and Youngchul Sung. Population-guided parallel policy search for reinforcement learning, 2020.

[Kaf17] T. Kaffka. *Neuronale Netze. Grundlagen: Mit Beispielprogrammen in Java*. mitp Professional. Mitp Verlag, 2017.

[Kat17] Katja Sandweg. Heuristik – die basis für schnelle entscheidungen in komplexen umwelten, 2017. Accessed: 2020-01-19.

[KHPK20] MyeongSeop Kim, Dong-Ki Han, Jae-Han Park, and Jung-Su Kim. Motion planning of robot manipulators for a smoother path using a twin delayed deep deterministic policy gradient with hindsight experience replay. *Applied Sciences*, 10(2):575, January 2020.

[Kri07] David Kriesel. *Ein kleiner Überblick über Neuronale Netze.* , 2007.

[Ork] Orklaert. Operations Research verständlich erklärt. Accessed: 2020-01-19.

[Ric19] Stefan Richter. *Statistisches und maschinelles Lernen*. Springer-Verlag GmbH Deutschland, Heidelberger Platz 3, 14197 Berlin, Germany, 2019.

[RN04] Stuart J. Russell and Peter Norvig. *Künstliche Intelligenz. Ein moderner Ansatz*. Pearson Studium, 2., überarb. a. edition, 2004.

[SB98] RS Sutton and AG Barto. Actor-critic methods. *Reinforcement Learning: An Introduction*, 6(2):151–153, 1998.

[Ste00] Stefanie Winter. Quantitative vs. qualitative methoden, 2000. Accessed: 2020-01-18.