Deep Reinforcement Learning für kooperatives Fußballspiel in einer Multi-Agenten Simulation

Masterarbeit

Jonas Kuball

7. Januar 2020

Fachbereich Mathematik und Informatik
Studiengang Informatik

1. Gutachter: Dr. Tim Laue
2. Gutachter: Prof. Dr. Tanja Schultz

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Bremen, 7. Januar 2020

_____
Jonas Kuball

„Konfusion vor Klarheit.“

INHALTSVERZEICHNIS

1 Einleitung **6**

1.1 Forschungsfrage und Voraussicht 7

1.2 Aufbau der Arbeit 8

2 Grundlagen **10**

2.1 Künstliche Neuronale Netzwerke 10

2.1.1 Das künstliche Neuron 11

2.1.2 Netzwerkschichten 12

2.1.3 Neuronale Netzwerke als azyklischer Graph 13

2.2 Das Training eines Netzes 14

2.2.1 Fehlerrückführung 14

2.2.2 Gradientenabstieg 17

2.3 Markow-Entscheidungsprobleme 17

2.4 Markow-Spiele 18

2.5 Verstärkendes Lernen 18

2.5.1 Zustands- und Beobachtungsräume 19

2.5.2 Aktionsräume 20

2.5.3 Belohnungsfunktionen 20

2.5.4 Spärliche Belohnungen 21

2.5.5 Exploration und Exploitation 21

2.5.6 Ertrag 22

2.6 Policies 23

2.7 Value Function 24
2.8 Advantage Function 24
2.8.1 GAE-Lambda 25
2.9 Stochastische Policies mit neuronalen Netzwerken 26
2.9.1 Anwendung 26
2.10 Lernalgorithmen 27
2.10.1 Modellfreies und -basiertes verstärkendes Lernen . . 27
2.10.2 Policy Optimization 28
2.10.3 Policy Gradient 28
2.10.4 Trust Region Policy Optimization 28
2.10.5 Proximal Policy Optimization 30

3 Verwandte Arbeiten **33**
3.1 AlphaZero (Silver, Hubert u. a., 2017) 33
3.2 OpenAI Five (OpenAI u. a., 2019) 34
3.3 Hide and Seek (Baker u. a., 2019) 37
3.4 Im Roboterfußball (Ocana u. a., 2019) 39

4 Die Fußballsimulation **42**
4.1 Die Gym-Schnittstelle 42
4.2 Umgang mit Multi-Agenten-Umgebungen 44
4.3 Umfang 45
4.3.1 Spielregeln 46
4.4 Architektur 46
4.4.1 Feldkonfiguration 46
4.4.2 Umgebungskomponenten 48
4.5 Koordinatensysteme 48
4.6 Aktionsräume 49
4.7 Beobachtungsräume 51
4.7.1 Erweiterung durch Polare Koordinaten 52
4.7.2 Normalisierung 54

5 Zwei gegen Zwei **55**
5.1 Umgebungsaufbau 55

5.2 Direktnachbau 56
5.3 Anreiz der Ballbewegung 60
5.4 Anreiz des Tores 62
5.5 Normieren und Negieren 63
5.6 Geteilte Schichten und Rekurrenz 69

6 Abschluss **74**
6.1 Fazit 74
6.2 Erkenntnisgewinn 75
6.2.1 Entwicklungsaufwand der Umgebung 75
6.2.2 Komplexität der Problemstellung 76
6.2.3 Netzwerkarchitekturen 76
6.2.4 Initialer Fokus auf Belohnungsfunktionen 77
6.3 Ausblick 77
6.3.1 Zwei-gegen-Zwei 77
6.3.2 Generelles Fußballspiel 78

EINLEITUNG

Bereits seit einigen Jahren durchläuft das Fachgebiet des „Maschinellen Lernens“ eine regelrechte Renaissance. Selbst Fachfremde können mit dem Begriff „Künstliches Neuronales Netzwerk“ etwas anfangen – auch thematisch passende Artikel in der Boulevardpresse sind keine Seltenheit. Dies liegt zum einen natürlich daran, dass die Künstliche Intelligenz (KI) eine starke ethische Komponente beinhaltet und somit ein diskussionswürdiges Thema für die Allgemeinheit ist; aber auch daran, dass sie inzwischen beinahe ubiquitär im Lebensstandard des modernen Menschen angekommen ist.

Diese Allgegenwärtigkeit ist in vielen Fällen eher unter einem Deckmantel unterwegs und wird von den Endnutzern meist gar nicht erst als KI wahrgenommen. Dies soll aber gar nicht die Thematik dieser Arbeit sein, sondern ein weiteres Phänomen, weshalb die KI in der Allgemeinheit angekommen ist: Die Wettbewerbe, die bereits seit vor der Jahrtausendwende stattfinden, bei denen die weltbesten menschlichen Spieler eines nicht genauer definierten Spieles gegen einen Computer antreten.

Der erste große bekannte Fall ist der Schachcomputer *Deep Blue*, welcher bereits 1996 den damals amtierenden Schachweltmeister Garri Kimowitsch Kasparow besiegen konnte (Campbell, Hoane Jr. und Hsu, 2002). Obgleich *Deep Blue* nach heutigen Standards keine KI darstellt, haben die sechs Partien gegen Kasparow große mediale Aufmerksamkeit genossen.

Zur selben Zeit wurde der *RoboCup* erdacht – ein Robotikwettbewerb, bei dem Teams aller Welt mit Robotern gegeneinander Fußball spielen. Der *RoboCup* spaltet sich in mehrere Ligen auf. Darunter die Standard Platform League (SPL), bei der jedes Team die selbe Hardware mit eigens entwickelter Software bespielt. Diese Einschränkungen machen aus der SPL somit einen reinen Softwarewettbewerb. Seit 1997 gibt es jährliche Wettbewerbe, die grundsätzlich das höhere Ziel verfolgen, im Jahre 2050 gegen den amtierenden (menschlichen) FIFA-Weltmeister anzutreten. Auch hier ist eine jährlich wachsende mediale Aufmerksamkeit zu beobachten.

Lange galt das ursprünglich traditionell chinesische Brettspiel Go als der nächste Schritt auf dieser Zeitlinie, da es die Komplexität von Schach auf Basis aller möglichen Spielfeldkonfigurationen um Längen überschreitet. Silver, Huang u. a. (2016) gelang es, einen mehrfachen Europameister im Go zu schlagen – ein paar Monate danach sogar einen der weltbesten Profispieler. Dort wurden bereits *künstliche neuronale Netzwerke* eingesetzt, welche als Grundlage für diese Masterarbeit gewählt sind. Auch das *verstärkende Lernen*, welches die zweite große Basis für diese Arbeit liefert, kam bereits zum Zug. Obgleich die sogetaufte Go-KI *AlphaGo* noch vom Expertenwissen der Menschen gelernt hat, ist dies ein großer Meilenstein. Mit Silver, Hubert u. a. (2017) ist die nächste Stufe in Entwicklung; *AlphaZero* – eine Go-KI, die einzig und allein mit den Spielregeln ausgestattet ist und somit komplett eigene Strategien entwickeln kann.

Die jüngsten Ereignisse dieser besonderen Wettbewerbe liefern die Inspiration dieser Arbeit: das Spielen von Spielen mittels künstlicher Intelligenz. Ganz besonders hervorzuheben ist ein Fall von 2017, nach dem die komplette (menschliche) E-Sports-Szene ihre Taktiken angepasst hat, da eine KI einen neuen Ansatz entdeckt hat, der weit bessere Ergebnisse erzielte als alle Menschen zuvor. Bei diesem Fall handelt es sich um ein Projekt der Firma OpenAI u. a. (2019), die mithilfe von neuronalen Netzwerken das hochdynamische Computerspiel Defense of the Ancients (DOTA 2) der US-Amerikanischen Videospielfirma Valve Corporation (2019) spielen.

1.1 Forschungsfrage und Voraussicht

Direkt aus den Ergebnissen der Arbeit von OpenAI u. a. (2019) folgt eine Forschungsfrage, die sich zunächst sehr simpel ausdrücken lässt: Wie gut lässt sich (Roboter-)Fußball mit neuronalen Netzwerken spielen? Im Spiel DOTA 2 hat eine KI neue Taktiken entdeckt, von denen menschliche Spieler profitiert haben – etwas, was im Roboterfußball sehr erstrebenswert wäre.

Diese Arbeit soll grundsätzlich untersuchen, wie sich moderne Ansätze des sogenannten verstärkenden Lernens (engl. *Reinforcement Learning*) auf das Fußballspiel übertragen lassen – genau genommen auf die simplifizierte Version des Fußballspiels, wie sie beim Roboterfußball gespielt wird. Dazu muss eine leicht zu modifizierende Simulationssoftware entwickelt werden, welche als konzeptioneller Beweis genutzt und als Grundstein für weitere Untersuchungen verwendet werden kann.

Die Entscheidung sich auf das (Roboter-)Fußballspiel zu beziehen, fiel aufgrund eigener intrinsischer Motivation, das Bremer Team *B-Human* voranzubringen, für das ich mich seit 2015 einsetze. Weiterhin ist Fußball in seiner Dynamik weit weniger komplex als DOTA 2, da es insgesamt weniger verschiedene Spielzustandskonfigurationen gibt. Das könnte einen wichtigen Punkt darstellen, da *B-Human* nicht annähernd die selbe Hardwarestärke zur Verfügung stellen kann, wie sie bei OpenAI u. a. möglich ist (Vgl. Kapitel 3).

Daraus folgen auch zu erwartende Probleme: zur Verfügung steht ein einzelner Computer mit einer einzelnen CPU und einer einzelnen GPU. Die genauen Spezifikationen sind die Folgenden:

- **Intel® Core™ i5-7500** Prozessor, Sockel LGA 1151, **4-core @ 3,40 GHz**, 6 MB SmartCache
- **32 GB (2x 16 GB) DDR4 / PC2400 UDIMM** Arbeitsspeicher
- **NVIDIA® TITAN V** Grafikkarte
  - Architektur: Volta
  - CUDA-Kerne: 5120 / Tensor-Kerne 640
  - Core Clock: 1200 MHz / Boost Clock: 1455 MHz
  - VRAM: 12 GB HBM2 / L2 Cache: 4,5 MB

In dem Blogpost, in dem OpenAI u. a. ihren Ansatz beschreiben, fällt ebenfalls die Aussage, dass sie besonders wegen des stark skalierten Lernalgorithmus' über tausende CPUs und des wochenlangen Trainings gute Ergebnisse erzielen.

Das Hauptziel dieser Arbeit ist, unter Einbezug dieser Information, insbesondere die Entwicklung eines vielversprechenden Ansatzes mit *Deep Reinforcement Learning* für eine simplifizierte Version des Fußballspiels (Kapitel 4).

1.2 Aufbau der Arbeit

Nach dieser Einleitung folgen in Kapitel 2 die technischen Grundlagen der künstlichen Intelligenz, künstlichen neuronalen Netzen und ganz besonders die Grundlagen des verstärkenden Lernens – alle Kernkonzepte, die für das Verständnis dieser Arbeit nötig sind.

In Kapitel 3 folgt die Analyse verwandter Publikationen, welche ebenfalls

mithilfe künstlicher Intelligenz kompetitive (Multi-Agenten-)Spiele spielen. Teil und Hauptziel dieser Analyse ist die Extraktion der Ideen und Einsichten, von denen die Entwicklung eines eigenen Ansatzes profitieren könnte.

Um vollste Kontrolle über den Lernprozess der künstlichen Intelligenz (d.h. der neuronalen Netze) zu bekommen, wurde im Zuge dieser Arbeit eine Fußballsimulationssoftware entwickelt, welche in Kapitel 4 kurz beschrieben wird, einschließlich der Vor- und Nachteile der Eigenentwicklung.

Für Kapitel 5 wird ein Szenario definiert, auf dem mehrere Experimentreihen ausgeführt werden, welche sich damit befassen, wie das Problem des Fußballspielens unter Zuhilfenahme der eingeführten Methodiken zu lösen wäre.

Das letzte Kapitel 6 zieht das Fazit der Arbeit und soll einen Ausblick darüber geben, welche Ansätze weiter verfolgt werden könnten, um in Zukunft das Fußballspiel als Ganzes mit modernen Methoden der künstlichen Intelligenz anzugehen.

GRUNDLAGEN

Das folgende Kapitel leitet den Hauptteil der Arbeit mit den relevanten mathematischen Grundlagen ein. Diese Grundlagen umfassen neuronale Netzwerke und das Konzept des verstärkenden Lernens. Detailliertere Begriffe und Konzepte werden bei Erstbenutzung im Hauptteil eingeführt.

2.1 Künstliche Neuronale Netzwerke

Neuronale Netzwerke sind ein Konzept der künstlichen Intelligenz, welche als universelle Funktionsapproximatoren eingesetzt werden und gerade in der jüngsten Zeit große Aufmerksamkeit sowohl in der Wissenschaft als auch in den Medien erlangt haben. Die ersten Steine wurden schon sehr früh von McCulloch und Pitts (1943) gelegt, die eine mathematische Definition einer abstrakten Version eines Neurons, wie es im menschlichen Gehirn vorkommt, aufstellten. Die sogenannten *McCulloch-Pitts-Zellen* setzten sich nicht durch, hauptsächlich verschuldet dadurch, dass es keine Methoden gab, diese Zellen zu trainieren – die Parametrisierung einzelner Zellen war den Entwicklern vorbehalten. Von Hebb (1949) wurde erst am Ende des Jahrzehnts eine Methodik eingeführt, welche die rein auf binären Daten basierenden Zellen automatisch anlernen konnte.

Rosenblatt (1958) führte das *Perzeptron* ein, welches die Grundlage für moderne künstliche neuronale Netzwerke bildet. Dort wurden künstliche Neuronen bereits komplexer als noch von McCulloch und Pitts (1943) formuliert. Die Publikation wird heute noch als Pionierarbeit betrachtet.

Der in der modernen Zeit benutzte Lernalgorithmus der Fehlerrückführung (engl. *Backpropagation*) wurde über die Jahre von verschiedenen Quellen entdeckt, lässt sich aber auf die Doktorarbeit von Werbos (1994) zurückführen.

In den späten 80er Jahren begann die Wissenschaft, neuronale Netzwerke als universale Funktionsapproximatoren zu verwenden, was das Interesse

an diesen weiter entfachte – die währenddessen steigende Kraft an physikalischen Komponenten ermöglichte gleichzeitig immer komplexere Herangehensweisen. Zwischen 2009 und 2012 erreichten Anwendungen mit neuronalen Netzwerken in stark spezialisierten Aufgaben wie der Klassifikation von Verkehrsschildern oder der Erkennung von Handschriften Leistungen, die mit der von Menschen vergleichbar sind (Jürgen Schmidhuber, 2014).

Moderne neuronale Netzwerke erreichen teilweise übermenschliche Leistungen in verschiedensten Kategorien, von Bildverarbeitung, Spracherkennung und -synthese über das Spielen von Computerspielen – bis hin zum frühzeitigen Erkennen von Tumoren in medizinischen Daten.

2.1.1 Das künstliche Neuron

Ein modernes künstliches Neuron ist als Abstraktion zu den organischen Neuronen im menschlichen Gehirn zu betrachten. Ein Neuron besitzt m Eingänge x, denen jeweils m Gewichte w zugeordnet sind. Zusätzlich gibt es einen Bias-Wert b. Jeder Eingang x_i wird mit dem dazugehörigen Gewicht w_i multipliziert und aufsummiert. Um den Ausgang des Neurons zu berechnen, wird das Bias b des Neurons auf diese Summe addiert und das Ergebnis durch eine sogenannte Aktivierungsfunktion $\varphi : \mathbb{R} \to \mathbb{R}$ verrechnet.

$$u = \varphi\left(\sum_{i=0}^{m} x_i w_i + b\right) \tag{2.1}$$

Als Aktivierungsfunktion φ kann jede mögliche Funktion benutzt werden, sofern die Ableitung dieser bekannt ist. In dieser Arbeit wird, sofern nicht anders vermerkt, eine gleichrichtende Funktion benutzt – ein sogenannter *Rectifier*.

Rectifier

Als Rectifier wird die gleichrichtende Funktion genannt, welche erstmals von Hahnloser u. a. (2000) eingeführt wurde und bis heute die beliebteste Aktivierungsfunktion im Einsatz von künstlichen neuronalen Netzwerken ist. Der Rectifier ist dabei als Positivteil der Eingabe definiert:

$$f(x) = \max(0, x) \tag{2.2}$$

Neuronen, welche einen solchen Gleichrichter benutzen, werden als Rectified Linear Unit (RelU) bezeichnet.
Die Ableitung ist durch eine simple Fallunterscheidung beschrieben.

$$f'(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases} \tag{2.3}$$

Es liegt theoretisch eine mathematische Unstetigkeit bei $x = 0$ vor, beinahe alle Implementierungen definieren die Ableitung an der Stelle dennoch als 0.

2.1.2 Netzwerkschichten

Moderne künstliche neuronale Netzwerke bestehen nicht aus arbiträr miteinander verbundenen einzelnen, sondern aus Schichten von vielen Neuronen (Stanford University u. a., 2015). Eine solche Schicht (engl. *Layer*) beschreibt üblicherweise eine Sammlung von k Neuronen, denen allen die gleich Aktivierungsfunktion φ zugeordnet ist.

Jedes Neuron der Schicht ist mit jedem Eingang verbunden – meist mit den Ausgängen der Neuronen einer anderen Schicht. Der Ausgang des k-ten Neurons einer Schicht lässt sich analog zu Gleichung (2.1) errechnen, wobei jedes Neuron seine eigenen Gewichte w_k und sein Bias b_k zugeordnet hat.

$$u_k = \varphi\left(\sum_{i=0}^{m} x_i w_{ki} + b_{ki}\right) \tag{2.4}$$

Die Modellierung eines künstlichen neuronalen Netzwerks durch einzelne Schichten ist besonders hilfreich bei der Implementierung. Die Parametrisierung eines Layers kann als Matrix W und Vektor b und die Ein- und

Ausgänge als Vektoren X und U dargestellt werden. Das hat den Vorteil, dass die Ausgänge U eines Layers durch lineare Algebra gestützt wie folgt definiert berechnet werden können:

$$U = \varphi(W \cdot X + b) \tag{2.5}$$

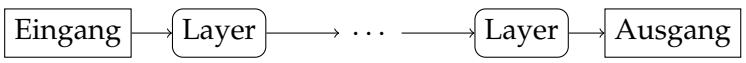
Dies erlaubt Implementierungskniffe wie parallele Ausführung und andere Optimierungen. Ganz besonders zu erwähnen ist dies, da Grafikkarten (GPUs) besonders gut mit Matrixmultiplikationen umgehen können – das üblicherweise zeitaufwendige Training von neuronalen Netzwerken kann allein durch die Nutzung einer GPU um ein Vielfaches beschleunigt werden.

Operationen

Dadurch, das einzelne Schichten auf Vektoren und Matrizen arbeiten, kann eine solche Schicht auch komplett ohne Neuronen definiert werden und einfache (oder komplexe) mathematische Operationen kapseln. Beispiele dafür sind das häufig eingesetzte Pooling, welches die Ausgänge einzelner Neuronen zusammenfasst, oder eine einfache tanh()-Operation um die Eingänge in den Wertebereich $[-1, 1]$ zu bringen.

2.1.3 Neuronale Netzwerke als azyklischer Graph

Ein (tiefes) neuronales Netzwerk ist als azyklischer gerichteter Graph aus einzelnen Schichten zu betrachten. Die am häufigsten auftretende Netzwerkstruktur (auch: Architektur) ist die des einfachen *Feed-Forward* Netzwerkes, welches sich durch die Aneinanderreihung von einzelnen Layern und genau einem Ein- und Ausgang auszeichnet:



Eingangs- und Ausgangsgrößen sind durch die genaue Anzahl der Neuronen der Layer definiert. Diese Art der Architektur zeigt gut, dass ein neuronales Netz als generische Funktion $f(x) = y$ betrachtet werden kann. Die Wertebereiche von x und y hängen von der gewählten Struktur ab.

Netzwerkarchitekturen sind aber in keinster Weise beschränkt, solange die azyklische, gerichtete Graphstruktur beibehalten wird. Ein modernes neu-

ronales Netzwerk kann beispielsweise aus mehreren Ein- und Ausgangsschichten bestehen und ist daher eher als generische Funktion mit einer variablen Anzahl an n Ein- und m Ausgängen $f(x_0, \ldots, x_n) = (y_0, \ldots y_m)$ zu betrachten.

Diese Arbeit nutzt aus, dass ein neuronales Netzwerk mehrere Ausgangsschichten besitzen kann.

2.2 Das Training eines Netzes

Der Ausgang eines künstlichen neuronalen Netzwerkes ist sowohl durch die Eingänge als auch durch die Gewichte bestimmt. Diese Gewichte sind als Parametrisierung des Netzes zu betrachten.

Anders als bei üblicher Parametrisierung von Algorithmen der Informatik werden diese Gewichte nicht vom Entwickler bestimmt, sondern durch einen iterativen Prozess – durch einen Lernalgorithmus. Dafür wird das Prinzip der Fehlerrückführung verwendet.

2.2.1 Fehlerrückführung

Der Algorithmus der Fehlerrückführung (engl. *Backpropagation*) wurde, wie in der Einleitung dieses Kapitels bereits angeschnitten, über die Jahre in vielen Arbeiten eingeführt. Sehr gut beschrieben wurde der Algorithmus von Rumelhart, Hinton und Williams (1988), deren Herleitungen im folgenden zusammengefasst werden.

Das Konzept der Fehlerrückführung basiert auf Trainingsdaten, welche eine Menge von Eingabedaten umfassen, für die bereits die gewünschten Ausgaben bekannt sind. Ziel ist es, Gewichte zu finden, mit welchen das neuronale Netz für jedes Ein- und Ausgabetupel der Trainingsdaten ausreichend genaue Ergebnisse liefert.

Der totale Fehler E des Netzwerkes mit den aktuellen Gewichten W über die gegebene Trainingsmenge D lässt sich wie folgt definieren, wobei c die Ein- und Ausgabetupel und j die Ausgabeneuronen indiziert. Das u bezeichnet die tatsächlichen und d die gewünschten Ausgaben des Netzwerkes.

$$E = \frac{1}{2} \sum_{c} \sum_{j} \left(u_{jc} - d_{jc} \right)^2 \tag{2.6}$$

E soll minimiert werden. Dafür ist es nötig, die partielle Ableitung von E in Hinsicht auf die einzelnen Gewichte des Netzwerkes zu errechnen – was laut den Autoren mit der Summe der partiellen Ableitungen über jedes Ein- und Ausgabetupel aus D gleichzusetzen sei.

Diese partiellen Ableitungen lassen sich in zwei Durchläufen bilden – einem Vorwärts- und einem Rückwärtsschritt. Im Vorwärtsschritt durchlaufen die Eingabedaten aus D nach den bereits definierten Methoden aus Gleichungen (2.4) und (2.5) das neuronale Netzwerk. Die partiellen Ableitungen dieser Funktionen müssen bekannt sein.

Der Rückwärtsschritt beginnt damit, $\partial E/\partial y$ für jedes Ausgangsneuron zu bestimmen. Die Differenzierung der Fehlerfunktion aus Gleichung (2.6) für einen speziellen Fall c führt zu

$$\partial E/\partial y_j = y_{cj} - d_{cj}. \tag{2.7}$$

Im Folgenden wird der Index c ausgelassen. Unter Anwendung der Kettenregel kann nun $\partial E/\partial x_j$ errechnet werden.

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j \tag{2.8}$$

Die Differenzierung von Gleichung (2.6) erlaubt es, dy_j/dx_j zu berechnen. Das Einsetzen der Ergebnisse in Gleichung (2.8) führt zu folgender Gleichung:

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j\left(1 - y_j\right) \tag{2.9}$$

Gleichung (2.9) liefert die Information, wie sehr sich die Änderung eines speziellen Eingangs x auf den Fehler E auswirkt. Dieser Eingang x ist aber als lineare Funktion aller Ausgaben der vorhergehenden Verbindungen und der Gewichte ebenjener zu betrachten. Dadurch ist es einfach zu berechnen, wie sehr sich der Fehler E verändern würde, würden wir diese Ausgaben und Gewichte anpassen.

Für ein Gewicht vom Neuron i zum Neuron j ist die Ableitung als

$$\partial E/\partial w_{ji} = \partial E/\partial x_j \cdot y_i \tag{2.10}$$

zu berechnen, was wiederum zum Fehlerbeitrag der Ausgabe des Neurons i führt:

$$\partial E/\partial x_j \cdot \partial x_j/\partial y_i = \partial E/\partial x_j \cdot w_{ji} \tag{2.11}$$

Unter Berücksichtigung aller ausgehenden Verbindungen des Neurons i, lässt sich mit folgender partieller Ableitung schließen:

$$\partial E/\partial y_i = \sum_j \partial E/\partial x_j \cdot w_{ji} \tag{2.12}$$

Gegeben der Gleichung (2.12) ist gezeigt, dass der Fehlerbeitrag eines jeden Neurons errechnet werden kann, wenn die Fehlerbeiträge aller ausgehenden Neuronen vorliegen. Da sich ebenjene Fehlerbeiträge der letzten Schicht des Netzwerkes nach Gleichung (2.7) direkt berechnen lassen können, kann nun rekursiv rückwärts durch die Netzwerkarchitektur gerechnet werden, während ebenfalls die Fehlerbeiträge jedes einzelnen Gewichts errechnet werden können.

Mithilfe der Fehlerbeiträge eines jeden Gewichtes $\partial E/\partial w_{ji}$ können dieselben nun angepasst werden, um den Fehler E zu minimieren. Dies geschieht üblicherweise durch Gradientenabstieg.

Fehlerfunktionen

Bei moderner Anwendung von neuronalen Netzwerken ist es üblich, eine eigene Fehlerfunktion E aufzustellen, die auf das zu lösende Problem zugeschneidert ist und somit besondere Annahmen treffen und Spezialistenwissen ausnutzen kann. Solange die Ableitung ebenjener Fehlerfunktion vorliegt, lassen sich die Fehlerbeiträge der einzelnen Gewichte $\partial E/\partial w_{ij}$ analog zu den Gleichungen (2.7) bis (2.12) berechnen – üblicherweise durch die verwendete Software automatisiert.

2.2.2 Gradientenabstieg

Rumelhart, Hinton und Williams (1988) empfehlen noch, die Fehlerbeiträge der einzelnen Gewichte über den kompletten Trainingsdatensatz zu akkumulieren, die Anpassung der Gewichte Δw daraus zu berechnen und einen kleinen Teil dieser mit jedem Trainingsschritt anzuwenden.

Heutzutage gibt es viele verschiedene Ansätze, welche sich üblicherweise aus der Stochastik bedienen. Ein solcher Algorithmus zum stochastischen Gradientenabstieg ist der von Kingma und Ba (2014) eingeführte *Adam* (kurz für *Adaptive Moment Estimation*), welcher auch in dieser Arbeit zum Einsatz kommt.

Adam ist dabei mit drei Werten parametrisierbar; der Lernrate α, welche die Stärke definiert, mit der die Parameter w eines neuronalen Netzwerkes in jedem Lernschritt angepasst werden und zwei sogenannte Verfallraten β_1, β_2.

2.3 Markow-Entscheidungsprobleme

Als Markow-Entscheidungsproblem (engl. *Markov Decision Problem*, MDP) bezeichnet man die mathematische Grundlage des Problems des verstärkenden Lernens. Ein MDP ist ein 5-Tupel $\langle S, A, T, R, \rho_0 \rangle$, wobei jeweils gilt:

- S ist eine Menge von Zuständen s.
- A ist eine Menge von Aktionen a.
- T ist eine Wahrscheinlichkeitsverteilung $T : S \times A \times S \to [0, 1]$, die beschreibt, wie wahrscheinlich eine gewisse *Transition* ist.
- R ist eine Bewertungsfunktion $R : S \times A \to \mathbb{R}$, die einer Transition eine unmittelbar folgende Bewertung zuweist.
- ρ_0 ist die initiale Zustandsverteilung.

Transition

Als Transition bezeichnet man einen Übergang von einem Zustand $s \in S$ mit einer gewählten Aktion $a \in A$ in einen folgenden Zustand $s' \in S$.

Das Ziel eines MDP ist es, eine Vorgehensweise $\pi : S \to A$ zu finden, welche zu jedem Zustand eine passende Aktion auswählt, um die gegebene Belohnung über die Zeit zu maximieren. Alles weitere über MDPs ist in Howard (1960) nachzulesen.

2.4 Markow-Spiele

Das Markow-Spiel (engl. *Markov Game*, MG) ist als direkte Multi-Agenten-Erweiterung des MDP zu sehen. Anstatt eines einzelnen Akteurs gibt es mehrere, die alle gleichzeitig in der selben Umgebung agieren. Jeder Agent kann dabei eine eigene Belohnungsfunktion zugewiesen haben. Ein MG ist wieder ein 5-Tupel $\langle S, \mathcal{A}, T, \mathcal{R}, \rho_0 \rangle$, wobei dieses sich von den MDP folgendermaßen unterscheidet:

- $\mathcal{A}$ ist die Sammlung aller Aktionsmengen der einzelnen Agenten $A^1, \ldots, A^k$.
- $\mathcal{R}$ ist die Belohnungsfunktion $\mathcal{R}^i : S \times A^1 \times \ldots \times A^k \to \mathbb{R}$, die für jeden Agenten eine eigenen Wert berechnen kann, aber nicht muss.
- S, T und ρ_0, wie bei MDP.

Markow-Eigenschaft

Sowohl MDPs als auch MGs tragen ihren Namen, da ihnen die Markow-Eigenschaft zugrundeliegt: Transitionen hängen nur vom aktuellen Zustand und den aktuellen Aktionen ab und nicht von ihrer Vergangenheit.

2.5 Verstärkendes Lernen

Verstärkendes Lernen (auch: Bestärkendes Lernen, engl. *Reinforcement Learning*), ist der Oberbegriff aller Methoden des maschinellen Lernens, die sich damit beschäftigen, dass ein Agent selbstständig lernt, eine Belohnung über die Zeit zu maximieren. Dementsprechend behandelt verstärkendes Lernen das Finden einer Lösung für ein gegebenes MDP.

Die beiden großen Hauptteile des verstärkenden Lernens sind ebenjener Agent und die Umgebung, mit der er interagiert. Die Umgebung (engl. *Environment*) beliefert den Agenten mit dem momentanen Zustand s_t der Welt. Der Agent nutzt die Vorgehensweise $\pi(s_t) \rightarrow a_t$ um eine Aktion a_t zu wählen, die der Umgebung mitgeteilt wird. Der Zustand der Umgebung verändert sich basierend auf der gewählten Aktion, kann sich aber auch aus anderen Gründen verändern. Es resultiert ein neuer Zustand s_{t+1}, mit welchem der Kreislauf wieder von vorn beginnt. Die Umgebung ist außerdem dafür zuständig, dem Agenten direkt mitzuteilen, wie „gut“ die gewählte Aktion war, indem sie zusätzlich einen Belohnungswert r_{t+1} errechnet. Das Ziel des Agenten ist eine Maximierung der kumulativen Belohnung über die Zeit.

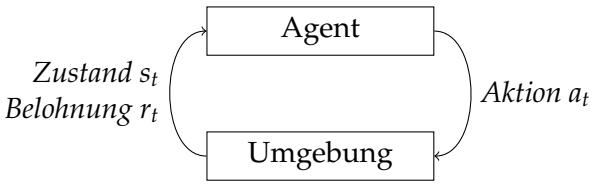


Abbildung 2.1: Die sogenannte Agenten-Umgebungs-Interaktionsschleife (engl. *Agent-Environment-Interaction Loop*).

2.5.1 Zustands- und Beobachtungsräume

Ein Zustand s_t ist eine komplette Beschreibung der Welt, welche keine Information auslässt. Nicht immer können wir einen perfekten Zustand zur Verfügung stellen, besonders dann, wenn die Umgebung mit der realen Welt interagiert um beispielsweise einen Roboterarm zu steuern. Eine nur partielle Beschreibung des Zustandes der Umgebung wird Beobachtung o_t (engl. *Observation*) genannt. In der Literatur werden diese beiden Begriffe häufig synonym eingesetzt, es sollte aus dem Kontext aber immer klar hervorgehen, welche der beiden Bezeichnungen korrekt ist.

Ein Zustand oder eine Beobachtung kann dabei jeglicher Repräsentation von Daten entsprechen, üblich sind allerdings Vektoren, Matrizen oder höherdimensionale Tensoren – beispielsweise die RGB-Kanäle eines Kamerabildes. Das liegt hauptsächlich daran, dass dies die bevorzugten Daten sind, mit denen ein neuronales Netz umgehen kann.

Tensoren

Ein Tensor ist ein mathematisches Konstrukt und beschreibt eine lineare Abbildung von algebraischen Objekten auf andere. Die Anzahl der indizierenden Objekte bestimmt den Rang des Tensors. Ein Tensor des 0. Ranges ist als Skalar, ein Tensor des 1. Ranges als Vektor und des 2. Ranges als Matrix zu betrachten. (Kolecki, 2002)

2.5.2 Aktionsräume

Die Möglichkeit, wie ein Agent mit einer Welt interagieren kann ist über die Wahl seiner Aktion $a \in A$ bestimmt. Rein theoretisch kann auch hier jede mögliche Art der Darstellung gewählt werden, üblich sind aber genau zwei Möglichkeiten:

- **Diskret**: Der Agent wählt aus einer endlichen Menge von diskreten Aktionen.
- **Stetig**: Der Agent gibt einen Vektor an stetigen, reellen Zahlen aus, die von der Umgebung interpretiert und umgesetzt werden, um den Zustand zu ändern. In diesem Fall hat der Agent natürlich eine Möglichkeit, Aktionen filigran auswählen zu können.

2.5.3 Belohnungsfunktionen

Die Belohnungs- oder Bewertungsfunktion R weist einer Transition ein unmittelbar folgendes Skalar zu. Das Ziel des Agenten ist es, die kumulative Summe aller Belohnungen über die Zeit aller *Trajektorien* zu maximieren.

Trajektorie

Eine Trajektorie τ ist die Bezeichnung für eine Sequenz von aufeinander folgenden Zuständen und Aktionen.

$$\tau = (s_0,\ a_0,\ \ldots,\ s_n,\ a_n,\ s_{n+1}) \tag{2.13}$$

Je nach Quelle werden Trajektorien auch als sogenannte Episoden bezeichnet. Die Worte sind dabei komplett synonym.

Es gibt keine Einschränkung des Wertebereichs einer Belohnungsfunktion. Eine bestimmte Belohnung r zu einem bestimmten Zeitpunkt t kann

den Wert jeder rellen Zahl annehmen, also $r_t \in \mathbb{R}$. Dennoch profitieren die meisten Lernalgorithmen davon, dass sich Belohnungen verschiedener Zeitpunkte nicht zu weit voneinander unterscheiden – üblich ist es, normalisierte Zahlen zwischen -1 und 1 zu benutzen.

Die Wahl der Belohnungsfunktion hat starken Einfluss auf das gelernte Verhalten des Agenten. So wird ein Agent immer versuchen, die Episode schnellstmöglich zu beenden, wenn die Bewertungsfunktion ihn regelmäßig mit negativen Zahlen belohnt. Andersherum wird sich ein Agent, der nur positive Belohnungen erfährt, hauptsächlich in Bereichen des Zustandsraums aufhalten, an dem diese Zahlen maximal hoch sind – der Fokus liegt nicht auf dem Beenden der Episode.

2.5.4 Spärliche Belohnungen

Ein Agent kann nicht besser werden als seine Belohnungsfunktion es zulässt. Moderne Methoden des verstärkenden Lernens setzen dabei auf sogenannte spärliche Belohnungsfunktionen (engl. *Sparse Rewards*).

Das bedeutet namensgerecht, dass eine Belohnung nur zu gewissen Zeitpunkten existiert und ansonsten marginal oder gleich null ist. Ein Beispiel dafür wäre die folgende Funktion für das Fußballspiel:

$$R_t = \begin{cases} 1 & \text{wenn Tor fällt} \\ -1 & \text{wenn Eigentor fällt} \\ 0 & \text{sonst} \end{cases}$$

Der große Vorteil daran ist, dass der Agent die Taktik, mit der er das Ziel erreicht, komplett selbst wählen kann. Dies ist aber nur möglich, wenn der Agent den Zustandsraum der Umgebung entdeckt, bei dem er die Belohnung überhaupt bekommt.

Damit der Agent eine solche spärliche Belohnung entdeckt, wird zum Zeitpunkt des Trainings eine sogenannte Strategie zur *Exploration* benötigt.

2.5.5 Exploration und Exploitation

Während des Trainings muss ein Agent entscheiden, welche Aktion er auf die Umwelt auswirkt. Dabei muss entschieden werden, ob der Agent sich

auf das bereits Gelernte verlässt (*Exploitation*) oder ob er eine andere Aktion auswählt, die das bisher Gelernte nicht widerspiegelt (*Exploration*).

Es gibt mehrere moderne Ansätze wie beispielsweise der Einsatz von Neugierde (engl. *Curiosity*), der in Burda u. a. (2018) umrissen wird, aber nicht Teil dieser Arbeit ist.

2.5.6 Ertrag

Im verstärkenden Lernen wird schlussendlich der kumulative Ertrag (engl. *Return*) der Aktionen eines Agenten maximiert. Der Ertrag einer Trajektorie τ wird als $R(\tau)$ notiert und ist zumindest theoretisch als einfache Summe aller Belohnungen derselben Trajektorie definiert:

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.14}$$

Praktisch wird ein sogenannter Diskontierungsfaktor (engl. *Discount Factor*) $\gamma \in [0, 1]$ hinzugezogen, mit dem es möglich ist, nicht nur den Ertrag einer Trajektorie zu berechnen, sondern den Ertrag aller Aktionen eines Agenten über die komplette Zeitachse zu berechnen.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.15}$$

Diese endlose Summe konvergiert wegen des γ-Faktors und kann daher auch maximiert werden.

Erwarteter Ertrag

Bei einer stochastischen Umgebung und Vorgehensweise ist die Wahrscheinlichkeit einer Trajektorie wie folgt beschrieben:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\, \pi(a_t|s_t) \tag{2.16}$$

Die erwartete Ertragsfunktion – basierend auf der Wahrscheinlichkeit einer Trajektorie – notiert sich als:

$$J(\pi) = \int_{\tau} P(\tau|\pi)\, R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] \tag{2.17}$$

Diese Funktion $J(\pi)$ ist im verstärkenden Lernen das, was maximiert wird.

2.6 Policies

Die sogenannte *Policy* (zu Deutsch „Vorgehensweise“ oder „Strategie“) ist das Herzstück eines jeden Agenten. Eine Policy entscheidet auf der Basis eines Zustandes eines bestimmten Zeitpunktes s_t welche Aktion a_t der Agent wählt. Sie kann dabei deterministisch oder stochastisch sein. Deterministische Policies werden üblicherweise mit μ notiert, Stochastische mit π. Ist eine Policy parametrisierbar (beispielsweise ein künstliches neuronales Netzwerk), werden die Parameter mit θ bezeichnet und an die Notation angehängt.

Policy-Notationen

Für die Berechnung einer Aktion aus einer diskreten Policy schreibt man einfach

$$a_t = \mu_\theta(s_t)\,, \tag{2.18}$$

für das Samplen aus einer stochastischen Policy schreibt man

$$a_t \sim \pi_\theta(\cdot|s_t)\,. \tag{2.19}$$

Eine Policy kann prinzipiell jedmögliche Funktion sein, die Aktionen aus dem gegebenen Aktionsraum A diskretisieren kann. Im Falle des „tiefen“ verstärkenden Lernens ist eine Policy meist ein neuronales Netzwerk π mit den Gewichten θ.

Das Grundkonzept des verstärkenden Lernens lässt sich nun folgendermaßen darstellen, wobei π^* die theoretisch optimale Policy darstellt.

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.20}$$

2.7 Value Function

Beinahe jeder Algorithmus im verstärkenden Lernen verwaltet neben der Policy π noch eine Bewertungsfunktion (engl. *Value Function*) $V^\pi(s)$, welche den erwarteten Ertrag aller Trajektorien τ eines Zustandes errechnet, wenn beginnend mit dem Zustand s folgend strikt nach der Policy π vorgegangen wird:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)\,|\,s_0 = s] \tag{2.21}$$

Eine Value Function wird üblicherweise ebenfalls als neuronales Netz dargestellt.

Manchmal wird ebenfalls eine *Action-Value Function* $Q^\pi(s, a)$ benötigt, welche den erwarteten Ertrag explizit für einen Zustand s und eine arbiträre Aktion a approximiert, wenn damit begonnen und dann strikt nach der Policy π vorgegangen wird.

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)\,|\,s_0 = s, a_o = a] \tag{2.22}$$

Genau wie die theoretisch optimale Policy als π^* bezeichnet wird, gibt es auch die theoretisch optimalen Value Functions V^* und Q^*.

$$V^\pi(s) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)\,|\,s_0 = s] \tag{2.23}$$

$$Q^\pi(s, a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)\,|\,s_0 = s, a_o = a] \tag{2.24}$$

2.8 Advantage Function

Es ist nicht immer nötig genau zu bestimmen, wie gut eine Aktion auf einer absoluten Skala ist. Stattdessen reicht es meist zu berechnen, wie viel besser eine Aktion a in Relation zu anderen Aktionen ist. Dieses relative Maß wird Vorteil (engl. *Advantage*) genannt und von einer Vorteilsfunktion $A^\pi(s, a)$ bestimmt.

Mathematisch ist die Vorteilsfunktion durch die Differenz von V und Q definiert: wie viel besser es ist, die gewählte Aktion a zu nehmen anstatt direkt nach der Policy π zu handeln.

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s) \tag{2.25}$$

Es gibt viele Methoden zur Approximierung einer Vorteilsfunktion, in dieser Arbeit kommt das sogenannte GAE(λ) zum Einsatz.

2.8.1 GAE-Lambda

Generalized Advantage Estimation (GAE) ist ein von Schulman, Moritz u. a. (2015) eingeführter Algorithmus zur Schätzung von Advantages. In der einführenden Publikation wird dieser Algorithmus nur GAE benannt, in anderen Publikationen wird er gemeinhin als GAE(λ) bezeichnet – geschuldet durch den erforderlichen Hyperparameter $\lambda \in [0, 1]$. Zusätzlich wird noch ein Diskontierungsfaktor $\gamma \in [0, 1]$ benötigt, welcher durch den Ertrag aber bereits eingeführt wurde und meist wiederverwendet wird.

Policy Gradient-Algorithmen (Vgl. Abschnitt 2.10.3) maximieren den erwarteten Ertrag, indem sie wiederholt den Gradienten g der Policy berechnen. Für diesen wird ein Ertragsschätzwert $\hat{A}_t$ benötigt, welcher von GAE(λ) bestimmt wird. Schulman, Moritz u. a. (2015) nehmen einen Gradienten der folgenden Form an, wobei N eine Menge von Episoden beschreibt.

$$\hat{g} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{\infty} \hat{A}_t^n \nabla_{\theta} \log \pi_{\theta}\left(a_t^n | s_t^n\right) \tag{2.26}$$

Mit V als angenäherter Value Function definieren sie

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)\,, \tag{2.27}$$

auf Basis dessen sie den folgenden Ertragsschätzungswert herleiten.

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}^V \tag{2.28}$$

In dieser Arbeit ist λ grundsätzlich auf 0,95 gesetzt.

2.9 Stochastische Policies mit neuronalen Netzwerken

Eine indirekte Methode zur Lösung des *Exploration vs. Exploitation*-Problems (Abschnitt 2.5.5) ist es, die per Definition deterministischen neuronalen Netzwerke auf eine stochastische Version zu erweitern. Anstatt das neuronale Netzwerk direkt den Aktionsvektor ausgeben zu lassen, werden die Parameter einer diagonalen multivariaten Gaussverteilung ausgegeben.

Diagonale multivariate Gaussverteilung

Eine multivariate Gaussverteilung ist eine Wahrscheinlichkeitsverteilung, die durch einen Mittelwertvektor μ und einer Kovarianzmatrix Σ dargestellt wird. Die diagonale multivariate Gaussverteilung beschreibt den Spezialfall, dass die Kovarianzmatrix nur Einträge auf der Diagonalen besitzt; als Folge daraus lässt sich diese als Vektor darstellen. Das Ziehen eines Elementes x aus einer Gaussverteilung $\mathcal{N}$ mit den Parametern μ und Σ wird als $x \sim \mathcal{N}(\mu, \sigma)$ notiert.

Dadurch, dass die Kovarianzen der Gaussverteilung initial sehr ungenau definiert (da zufällig belegt) sind, wählt der Agent indirekt immer Aktionen zur Exploration der Umgebung aus. Je länger der Trainingsvorgang läuft, desto sicherer wird der Agent in seiner Entscheidung – die Rate der Exploitation steigt stetig. Da es sich um einen stochastischen Vorgang handelt, kann die Explorationsrate niemals exakt Null erreichen.

2.9.1 Anwendung

Das Berechnen einer Aktion aus einer sogenannten diagonalen Gaussschen Policy ist wie folgt definiert, wobei $\mu_\theta(s)$ den gegebenen Mittelwertvektor und $\sigma_\theta(s)$ die gegebene Standardabweichung der Policy beschreiben:

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z \tag{2.29}$$

Das zusätzliches Rauschen z ist gezogen aus der multivariaten Standardverteilung $\mathcal{N}(0, I)$. Die Operation $\odot$ bezeichnet das elementweise Produkt zweier Vektoren.

Um dem neuronalen Netz das Lernen zu vereinfachen, soll dieses nie tatsächlich die Standardabweichungen ausgeben, sondern die logarithmi-

schen Standardabweichungen $\log \sigma_\theta(s)$, aus denen die Abweichung $\sigma_\theta(s)$ berechnet werden kann. Dies hat den Hintergrund, dass die logarithmischen Standardabweichungen den kompletten Bereich der reellen Zahlen abdecken können – einfache Standardabweichungen müssen dagegen per Definition positiv sein.

2.10 Lernalgorithmen

Das verstärkende Lernen ist ein riesiger wissenschaftlicher Zweig mit einer großen Menge an verschiedenen Lernalgorithmen, welche sich zumindest grob in verschiedene Kategorien einordnen lassen (Vgl. Abbildung 2.2).

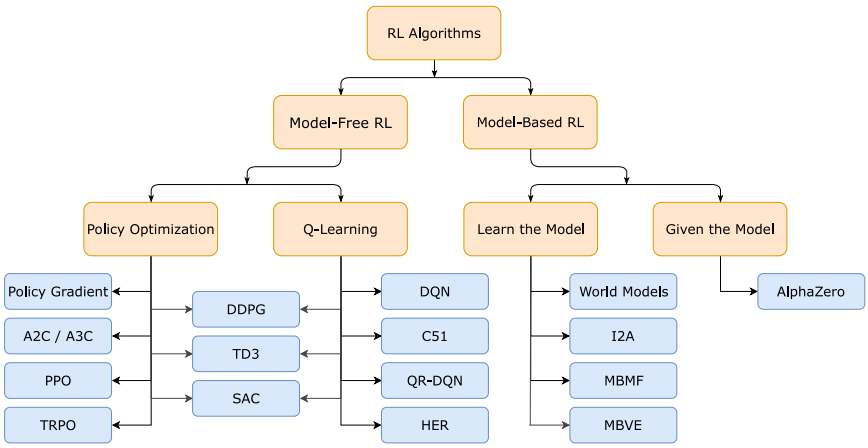


Abbildung 2.2: „A Taxonomy of RL Algorithms“ – eine nichtausschöpfende Liste von Algorithmen des verstärkenden Lernens. Abbildung unverändert übernommen von OpenAI (2019b).

2.10.1 Modellfreies und -basiertes verstärkendes Lernen

Die Trainingsalgorithmen des verstärkenden Lernens lassen sich in zwei Oberkategorien einordnen, *modellfrei* oder *modellbasiert*. Diese Einordnung lässt sich darauf zurückführen, ob der Agent Zugriff auf ein Modell der Umgebung hat oder nicht. Dieses Modell kann entweder programmiert (wie bei einem simplen Spiel wie Schach, wie es schon bei Vinyals u. a. (2019) der Fall war) oder vom Agenten selbst gelernt werden.

Die Popularität modellbasierten Lernens ist allerdings vergleichsweise gering, was darauf zurückzuführen ist, dass es bei hochkomplexen Umge-

bungen äußerst schwer ist, überhaupt ein Modell bereitzustellen.

Diese Arbeit fokussiert sich daher hauptsächlich auf Algorithmen des modellfreien Lernens.

2.10.2 Policy Optimization

Modellfreie Lernalgorithmen lassen sich ebenfalls in zwei Kategorien einteilen – die des *Q-Learnings* (für diese Arbeit nicht relevant) und die der *Policy Optimization*, auf denen hier der Fokus liegt.

Algorithmen der *Policy Optimization* repräsentieren die Vorgehensweise explizit als $\pi_\theta(a|s)$ und optimieren die Parameter θ direkt durch Gradientenaufstieg auf der erwarteten Ertragsfunktion $J(\pi_\theta)$.

2.10.3 Policy Gradient

Der simpelste Algorithmus zur *Policy Optimization* und gleichzeitig die Basis für Folgende ist Policy Gradient (PG), welcher von Sutton u. a. (2000) eingeführt wurde. Im Kern wird der Gradient der erwarteten Ertragsfunktion J der Policy π mit den Parametern θ berechnet, wobei A^{π_θ} die Vorteilsfunktion bezeichnet und τ eine beliebige Trajektorie.

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A^{\pi_\theta}(s_t,\ a_t)\right] \tag{2.30}$$

Damit können in einem Lernschritt die Parameter θ der Policy durch stochastischen Gradientenaufstieg optimiert werden.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta) \tag{2.31}$$

Die Wahl der Advantage Estimation ist dabei nicht festgelegt, benötigt aber üblicherweise eine Value Function.

2.10.4 Trust Region Policy Optimization

PG stellt sicher, dass sich eine „neue“ Policy im Parameterraum nicht weit von ihrer „alten“ Version unterscheidet. Gerade bei neuronalen Netzwer-

ken können aber bereits sehr kleine Veränderungen zu komplett anderem Verhalten führen, sodass die Policy sehr schnell zusammenbrechen kann.

Trust Region Policy Optimization (TRPO), eingeführt von Schulman, Levine u. a. (2015) fungiert als direkte Erweiterung des PG und adressiert dieses Problem mit einer sogenannten Ersatz-Advantage-Funktion (engl. *Surrogate Advantage*) $\mathcal{L}(\theta_k, \theta)$, welche sicherstellt, dass sich die Performanz der neuen Parameter nicht weit von den alten entfernt.

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta), \text{ sodass } \bar{D}_{KL}(\theta||\theta_k) \leq \delta \tag{2.32}$$

Die Parameteranpassung macht den größtmöglichen Schritt im Parameterraum ohne, dass sie sich aus einem sicheren Bereich hinausbewegt; der sogenannten *Trust Region*. Dieser sichere Bereich definiert sich aus der ebengenannten *Surrogate Advantage*-Funktion und einem Hyperparameter δ, der eine maximal erlaubte Abweichung beider Policies (alt und neu) voneinander festlegt.

Die *Surrogate Advantage*-Funktion beschreibt dabei einen Messwert, wie die Policy π_θ in Relation zu ihrer alten Version π_{θ_k} abschneidet, gegeben gesammelter Zustände der alten Policy.

$$L(\theta_k, \theta) = \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right] \tag{2.33}$$

Die Distanz $\bar{D}_{KL}(\theta||\theta_k)$ ist definiert als die durchschnittliche *Kullback-Leibler Divergenz* beider Policies über die Zustände, die die alte Policy bereits besucht hat.

$$\bar{D}_{KL}(\theta||\theta_k) = \mathop{\mathbb{E}}_{s\sim\pi_{\theta_k}} \Big[D_{KL}\Big(\pi_\theta(\cdot|s) \,||\, \pi_{\theta_k}(\cdot|s)\Big)\Big] \tag{2.34}$$

Kullback-Leibler Divergenz

Die Kullback-Leibler-Divergenz (auch: *relative Entropie*) ist ein Distanzmaß für zwei gegebene Wahrscheinlichkeitsverteilungen P und Q.

$$D_{KL}(P||Q) = \sum_{x\in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right) \tag{2.35}$$

Trust Region Policy Optimization kommt in dieser Masterarbeit nie zum Einsatz, ist aber fundamental zum Verständnis des folgenden Abschnitts.

2.10.5 Proximal Policy Optimization

TRPO ist mathematisch sehr klar definiert, allerdings schwer zu implementieren und von einigen Annäherungen abhängig. Stand der Technik im verstärkenden Lernen ist de facto der sogenannte Algorithmus Proximal Policy Optimization (PPO), welcher direkt an TRPO anschließt. Der Algorithmus ist einfach zu implementieren und liefert mindestens vergleichbare Ergebnisse wie TRPO, wenn nicht sogar bessere (OpenAI, 2019b). Eingeführt wurde der Algorithmus von Schulman, Wolski u. a. (2017).

Der Lernschritt zum Optimieren der Parameter basiert darauf, den Erwartungswert der Zielfunktion L (engl. *Objective*) zu maximieren, welche genau wie bei TRPO dafür sorgt, dass sich die Performanz der neuen Policy nicht zu weit von den alten entfernt.

$$\theta_{t+1} = \arg\max_{\theta} \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} [L(s, a, \theta_k, \theta)] \tag{2.36}$$

Die Zielfunktion L wird in der Ursprungspublikation folgendermaßen konstruiert, wobei der Trimmfaktor ϵ einen üblicherweise kleinen Hyperparameter beschreibt, der grundsätzlich die erlaubte Distanz der neuen Policy zur alten einschränkt.

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\mathcal{A}, \operatorname{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)\mathcal{A}\right) \tag{2.37}$$

mit

$$\mathcal{A} = A^{\pi_{\theta_k}}(s, a) \tag{2.38}$$

$$\operatorname{clip}(x, l, h) = \begin{cases} l & \text{wenn } x < l \\ h & \text{wenn } x > h \\ x & \text{sonst} \end{cases} \tag{2.39}$$

Achiam (2018) führt eine simplifizierte Version ebenjener Funktion ein, welche äquivalent zur Ursprungsfunktion ist:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right) \tag{2.40}$$

$$\text{mit } g(\epsilon, A) = \begin{cases} (1+\epsilon)\, A & \text{wenn } A \geq 0 \\ (1-\epsilon)\, A & \text{sonst} \end{cases} \tag{2.41}$$

Die Optimierung des Erwartungswertes dieser Funktion führt dazu, dass sich jedes einzelne Zustands-Aktions-Paar (s, a) basierend auf dem dazugehörigen Vorteil (*Advantage*) auf die Parameter auswirkt. Bei positivem Vorteil erhöht sich die Wahrscheinlichkeit, dass die Aktion mit den neuen Parametern wiedergewählt wird, bis $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$ gilt. Ist der Vorteil negativ, wird die Wahrscheinlichkeit der Aktion analog verringert, bis die untere ϵ-Grenze erreicht wird. Der Algorithmus ist als Pseudocode in Abbildung 2.3 abgedruckt.

Input: Initialbelegte Parameter der Policy θ_0 und der Value Function ϕ_0.

1: **for** k = 0, 1, ... **do**

2: Sammle eine Menge von Trajektorien $\mathcal{D}_k = \{\tau_i\}$, in dem die aktuelle Policy π_k mit der Umgebung interagiert.

3: Berechne den Ertrag für jede Transition $\hat{R}_t$.

4: Schätze die Advantage für jede Transition $\hat{A}_t$ basierend auf der aktuellen Value Function V_{ϕ_k}.

5: Nutze einen Algorithmus zum stochastischen Gradientenaufstieg um die Zielfunktion $L(s, a, \theta_k, \theta)$ zu maximieren:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \mathcal{A}, g(\epsilon, \mathcal{A}) \right) \tag{2.42}$$

$$\text{mit } \mathcal{A} = A^{\pi_{\theta_k}}(s_t, a_t) \tag{2.43}$$

6: Nutze einen Algorithmus zum Gradientenabstieg um die Value Function durch Regression über den Mean Squared Error (MSE) anzupassen.

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

7: **end for**

Abbildung 2.3: PPO-Clip als Pseudocode, welcher übersetzt von OpenAI (2019b) übernommen wurde.

VERWANDTE ARBEITEN

Das folgende Kapitel beinhaltet eine Auswahl anderer wissenschaftlicher Arbeiten, die sich bereits mit verstärkendem Lernen im Kontext von dynamischen und meist kompetitiven Multi-Agenten-Systemen beschäftigt haben. Aufgrund der sehr modernen Problemstellung finden sich nicht viele Publikationen, die älter als ein paar Jahre sind.

3.1 AlphaZero (Silver, Hubert u. a., 2017)

Wie in der Einleitung bereits angeschnitten ist mit Silver, Hubert u. a. (2017) der Algorithmus *AlphaZero* in Entwicklung, der eine generelle Erweiterung der erfolgreichen Künstliche Intelligenz (KI) *AlphaGo* darstellt. Wobei letztere sich noch auf Expertenwissen stützen musste, fungiert *AlphaZero* als genereller KI-Algorithmus, der ein Spiel von Grund auf und nur auf Basis der Regeln lernen kann.

Dies geschieht durch den Einsatz eines neuronalen Netzwerkes, welches das Spielbrett als 2×2 Matrix betrachtet und mögliche Züge sowie die davon zu erwartende Wertigkeit berechnet. Durch diese Generalisierung kann AlphaZero auf jedmögliches Brettspiel angewandt werden, was neben Go auch mit Schach und dem hierzulande weniger bekanntem Shogi getan wurde.

Ihre Ergebnisse sind bereits 2017 äußerst vielversprechend gewesen, und schon nach wenigen Stunden Trainingszeit schlug die künstliche Intelligenz herkömmliche Go-, Shogi- und Schach-KIs. Auch gegen die vorherigen Ansätze, die noch Expertenwissen ausnutzen, erreicht AlphaZero gute Gewinnraten.

Der Trainingsvorgang nutzt dabei eine allgemeingültige Monte-Carlo Suche über den mathematisch aufstellbaren Spielbaum; durch die kontinuierliche Natur des Fußballspiels es ist leider unmöglich, diesen aufzuspannen. Daher wird hier auf eine detailliertere Erklärung des Algorithmus verzichtet.

Interessant zu betrachten ist aber, dass AlphaZero während des Trainings niemals gegen einen anderen Spieler als sich selbst antritt. Diese moderne Vorgehensweise nennt sich *Self Play* und es wurde seitdem mehrfach gezeigt, dass dadurch Taktiken gefunden werden können, die weit über die Komplexität der Belohnungsfunktion der gegebenen Umgebung hinausgehen (Bansal u. a., 2017; Baker u. a., 2019; OpenAI u. a., 2019).

3.2 OpenAI Five (OpenAI u. a., 2019)

Inspiriert durch den großen Erfolg von *AlphaGo* begannen OpenAI u. a. (2019) mit der Entwicklung eines KI-Systems, welches mit hochdynamischen und weitaus weniger vorhersehbaren Umgebungen agiert. Bei der Auswahl der Problemstellung fiel die Entscheidung auf das Videospiel Defense of the Ancients (DOTA 2), da viele Videospiele die Komplexität und kontinuierliche Natur der echten Welt weit besser einfangen als Schach oder Go.

DOTA 2 ist ein erstmals 2013 von Valve Corporation (2019) veröffentlichtes Echtzeitstrategie Mehrspieler-Computerspiel, in dem üblicherweise zwei Teams von Fünf Spielern gegeneinander antreten. Das Spiel wird in der E-Sports-Szene äußerst aktiv auf professionellem Niveau gespielt; das insgesamt ausgeschüttete Preisgeld der internationalen DOTA 2-Meisterschaften 2019 überstieg mit über 35 Millionen Dollar alle anderen bis zu dem Zeitpunkt ausgetragenen Meisterschaften aller Videospiele.

Laut den Autoren eignet sich DOTA 2 durch die hohe Dimensionalität des Beobachtungs- und Aktionsraumes besonders als Herausforderung für das verstärkende Lernen. Der Beobachtungsraum ist zudem beschränkt: Beide Teams können nur einen Teil des Spielfeldzustandes einsehen und müssen selbst Rückschlüsse aus unvollständigen Daten ziehen. Dazu kommt ein Satz von komplexen Spielregeln und die Tatsache, dass eine Runde im Schnitt 45 Minuten anhält – wobei taktische Entscheidungen zu Beginn starke Auswirkungen auf das Ende des Spiels haben. Valve Corporation stellt weiterhin eine Softwareschnittstelle (API) für die Entwicklung eigener „Bots“ und Kompatibilität mit Linux bereit, wodurch die Spielsoftware äußerst gut als *Environment* eingesetzt werden kann.

Die tatsächliche Komplexität des Spielproblems lässt sich in Zahlen fassen und ist gut mit den herkömmlichen Problemen vergleichbar. Mit durchschnittlichen 45-Minuten-Runden bei 30 hz können theoretisch 81.000 Ent-

scheidungen pro Runde getroffen werden. Schach endet meist vor 40 Zügen und Go vor 150.

Ähnlich wie diese Masterarbeit es später tun wird, haben die Autoren sich zunächst auf eine Teilmenge des Problems fixiert. Anstatt direkt den üblichen Fünf-gegen-Fünf-Spielmodus anzugehen, fixierten sie sich auf das weniger beliebte (aber dennoch in Wettkämpfen ausgetragene) Eins-gegen-Eins-Szenario.

Die größte Inspirationsquelle dieser Arbeit stellen die Ergebnisse des für das Eins-gegen-Eins-Szenario entwickelten KI-Systems dar, wie weiter oben bereits eingeleitet wurde: bei der Internationalen DOTA 2-Meisterschaft 2017 trat eine frühe Version der *1v1-KI* auf der Bühne vor Publikum gegen den mehrfachen Turniergewinner Danil „Dendi“ Ishutin an – und gewann ein „Best-Of-Three“-Match. Außerhalb der Bühne konnten viele weitere Profis inklusive des amtierenden Eins-gegen-Eins-Weltmeisters geschlagen werden. Besonders hervorzuheben ist die spätere Einschätzung des dort anwesenden Livekommentators und ebenfalls professionellen Spielers William „Blitz“ Lee, der Bot habe die Art, wie professionelle Spieler über das 1v1-Spiel denken, verändert. Es sei wohl so, dass sich die allgemein meistbenutzte Taktik aller menschlichen Spieler angepasst habe – die KI hat eine bisher unbekannte, viel aggressivere Vorgehensweise bevorzugt.

Mit dem Erkenntnisgewinn aus der Entwicklung des 1v1-Bots konnte sich das Team auf das 5v5-Szenario einlassen, bei dem sie schlussendlich im Jahr 2019 mit fünf Instanzen neuronaler Netzwerke das Weltbeste DOTA 2-Team *Team OG* schlagen konnten. Durch die noch im selben Jahr veröffentlichte Publikation geben sie Einblick in ihre komplette Trainingsarchitektur.

Eine simplifizierte Darstellung der benutzten neuronalen Netzwerke ist in Abbildung 3.1 abgedruckt.

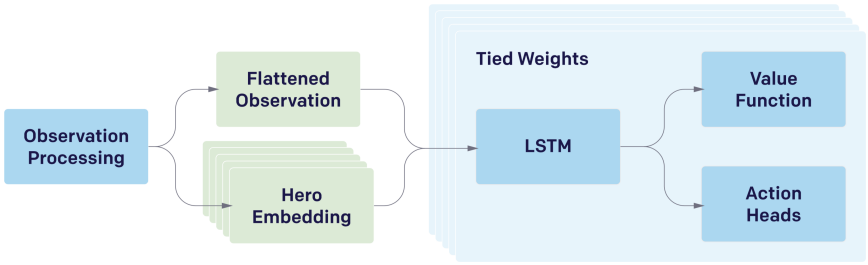


Abbildung 3.1: Eine stark simplifizierte Übersicht der eingesetzten Netzwerkarchitektur. Die Grafik ist unverändert übernommen aus OpenAI u. a. (2019).

Alle fünf Spielfiguren werden jeweils von einer Instanz desselben neuronalen Netzwerks gesteuert, welches im Kern aus einer 4096-Einheiten Long-Short-Term Memory-Schicht (LSTM) besteht, welche sich dadurch auszeichnet, sich Informationen über verschiedene Zeitschritte speichern zu können. Die Beobachtung des aktuellen Zeitschrittes wird durch ein sogenanntes *Embedding* vorverarbeitet – eine häufig eingesetzte Schicht, welche lernt, eine Sammlung von diskreten Variablen in einen n-dimensionalen Vektor abzubilden, wobei die einzelnen diskreten Messpunkte im Vektorraum maximal voneinander entfernt sind. In der Netzwerkarchitektur soll diese Schicht dafür sorgen, dass die einzelnen Instanzen der LSTM-Schichten verschiedene Eingänge bekommen.

Der Beobachtungsraum umfasst einen kontinuierlichen Vektor an 20.000 Zahlen, der verschiedenste Informationen über die Spielumgebung beinhaltet. Die tatsächliche Zusammensetzung des Raumes ist an dieser Stelle nicht genauer ausgeführt, da sie äußerst Domänenspezifisch für DOTA 2 entwickelt wurde. Auch der Aktionsraum übersteigt die Komplexität der für das Fußballspiel benötigten Aktionen um große Mengen. Insgesamt geben die einzelnen Agenten jeweils acht kontinuierliche Zahlen aus, welche je nach Kontext anders interpretiert werden können. So sind zwei dieser Aktionszahlen als Zielselektor eines Angriffes zu verstehen und wirken sich nur auf den Weltzustand aus, wenn zusätzlich auch eine Aktion ausgewählt wurde, welche die Selektion eines Ziels benötigt.

Ihr Trainingsvorgang besteht genau wie der von Silver, Hubert u. a. (2017) aus *Self-Play*. Das bedeutet, die Agenten spielen während des Trainings grundsätzlich gegen sich selbst. Um mehr Robustheit zu erlangen, entscheiden sich die Autoren dafür, die neuronalen Netzwerke in 20% der Spiele gegen ältere Versionen ihrer selbst antreten zu lassen.

Die eingesetzte Belohnungsfunktion setzt sich als Linearkombination von diversen Signalen des Spiels zusammen. Aus der aktuellen Spielsituation (Aktuelle Erfahrungspunkte, Lebenspunkte, ...) setzt sich eine Zahl zusammen, die den aktuellen Spielzustand mit einer Punktzahl bewertet. Der Agent wird nun mit der Punktzahldifferenz zwischen zwei Zeitschritten belohnt. Um Fokus auf den kompetitiven und kooperativen Teil des Spiels zu legen, werden die Belohnungen der einzelnen Agenten noch weiterverarbeitet, indem sie beispielsweise gewichtet auf das ganze Team aufgeteilt werden; dieses Vorgehen soll das Miteinanderspiel anreizen.

Sie führen ihren großen Erfolg besonders auf ihre massive Skalierung des Algorithmus Proximal Policy Optimization (PPO) zurück, den sie parallel

auf 128.000 CPUs und 256 GPUs durchführen, welche an einem Echtzeittag etwa 180 Jahre des Spieles simulieren – was, aufgeteilt auf fünf neuronale Netzwerke, Daten von über 900 Jahren Spielzeit pro Echtzeittag entspricht.

Weiterhin sind ihre neuronalen Netzwerke insgesamt über 10 Echtzeitmonate trainiert worden, wobei sie allerdings nach eigenen Aussagen etwa alle zwei Wochen Anpassungen an der Trainingsstruktur vorgenommen haben. Dabei haben sie sehr früh den Fokus darauf gelegt, möglichst viele Teile des bis dahin trainierten Netzwerkes übernehmen zu können, damit nicht nach jeder Anpassung von Null auf angefangen werden muss. Diese Vorgehensweise wird *surgery* genannt.

3.3 Hide and Seek (Baker u. a., 2019)

Baker u. a. (2019) haben jüngst gezeigt, wie effektiv auch die simpelsten Belohnungsfunktionen sein können, wenn *Self-Play* eingesetzt wird. Die Autoren lassen dabei zwei variabel große Teams von Agenten gegeneinander in einer Partie „Hide and Seek“ antreten; ein beliebtes Kinderspiel, bei dem ein suchendes Team (die *Sucher*) ein sich zu versteckendes Team (die *Verstecker*) nach einer kurzen Versteckphase suchen müssen.

Die Sucher und die Verstecker werden zusammen in eine mit Mauern versehene Arena gesetzt. In dieser Arena befinden sich Kisten und Rampen, die die Agenten bewegen können, um sich eigene Verstecke zu schaffen oder Mauern zu erklimmen.

Jeder Agent nimmt seine Umgebung durch die Positionen der Objekte in seinem eigenen Sichtkegel wahr, zudem ist er ringsherum mit Distanzmessern ausgestattet, welche den Abstand zur nächsten Wand oder dem nächsten Objekt messen. Mit der Welt interagieren sie, indem sie physikalische Kräfte auf ihre Körper auswirken. Außerdem können sie Kisten und Rampen ziehen und schieben und an einer Stelle fixieren, sodass nur das eigene Team diese Objekte wieder lösen kann.

Die bereits erwähnte äußerst simple Belohnungsfunktion ist im Kern binärer Natur. Während der initialen Versteckphase werden sowohl die Sucher als auch die Verstecker mit 0 belohnt. Das komplette Verstecker-Team wird während des laufenden Spiels mit 1 belohnt, wenn kein versteckter Agent direkten Blickkontakt mit einem Sucher hat – falls doch, mit -1. Bei den Suchern ist es umgekehrt, das Team wird mit 1 entlohnt, wenn auch

nur ein Sucher direkten Blickkontakt mit einem Verstecker aufbauen kann, ansonsten gibt es Punktabzug von −1.

Die Natur dieser Belohnungsfunktion ist, dass immer nur eins der beiden Teams eine 1 bekommt und das andere −1. Wollen beide Teams nun ihren erwarteten Ertrag maximieren, müssen regelmäßig neue Vorgehensweisen entwickelt werden um das andere Team zu übertrumpfen.

Genau diese Erwartungen legen die Autoren in ihren Ergebnissen dar. Abbildung 3.2 zeigt eine Übersicht über ihren Trainingsverlauf inklusive der dabei entstandenen Taktiken. Beginnend jagen die Sucher den Versteckern einfach nach, bis die Verstecker lernen, sich in der initialen Versteckphase ein Fort aus gegebenen Weltobjekten zu bauen. Die Sucher lernen im weiteren Trainingsverlauf, diese Mauern der Forts mithilfe der Rampen zu überspringen. Als direkte Antwort darauf beginnen die Verstecker diese Rampen an Ort und Stelle festzusetzen, bevor sie sich in ihr Fort zurückziehen.

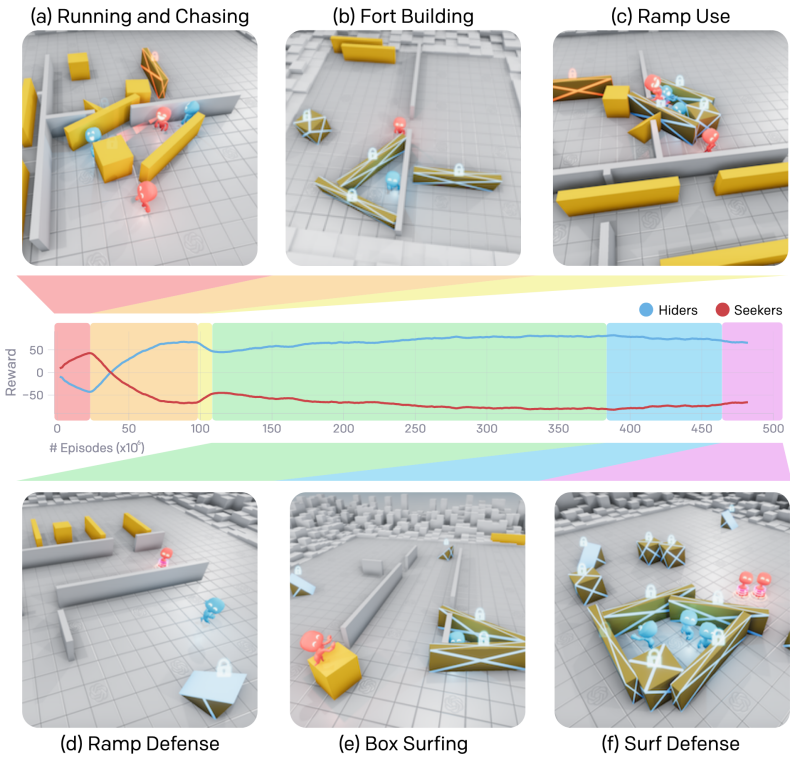


Abbildung 3.2: Eine Übersicht über die im Trainingsverlauf gebildeten Vorgehensweisen. Die Grafik ist unverändert aus Baker u. a. (2019) übernommen.

Zu erwarten ist, dass die Sucher nun keine Chance mehr haben, die Verstecker zu finden; dennoch wird ein Weg gefunden. Durch Ausnutzung der Physiksimulation entdecken die Sucher, dass sie sich mit den festgesetzten Rampen auf die Box bewegen können, die sie schieben wollen. Durch einen Programmierfehler in der Umgebung konnten sie die Rampen aber weiterhin schieben, während sie darauf standen (die Autoren tauften dieses Verhalten *Box Surfing*). Im Abschluss lernten die Verstecker, vor dem Bau des Forts alle bewegbaren Objekte der Arena zu fixieren.

Durch die Verringerung der Dauer der initialen Versteckphase konnten bei den Versteckern Verhaltensweisen ausgemacht werden, die bewusster Zusammenarbeit entsprachen. So wurde beobachtet, wie die Agenten sich gegenseitig Objekte zurechtlegen, um das enge Zeitfenster maximal effizient ausnutzen zu können.

Die benutzte Trainingsarchitektur ist sehr ähnlich zu der von OpenAI u. a. (2019) aufgebaut. So besteht das Netzwerk im Kern aus einer großen LSTM-Schicht und der gewählte Trainingsalgorithmus ist PPO.

3.4 Im Roboterfußball (Ocana u. a., 2019)

Direkt im Kontext des Roboterfußballs ist eine aktuelle Arbeit von Ocana u. a. (2019) entstanden, welche sich als großer Ankerpunkt für diese Arbeit herausstellte. Die Autoren haben dort eine Zwei-gegen-Zwei-Situation in einer 3D-Fußballsimulation analysiert und diverse neuronale Netzwerke mithilfe des Algorithmus Deep Deterministic Policy Gradients (DDPG) angelernt.

Besonders hervorzuheben ist hier ihre Trennung in zwei verschiedene Ansätze, die sie auf dem gleichen Experiment evaluieren. Zum einen lassen Sie das angelernte neuronale Netzwerk je einen Roboter steuern, zum anderen lassen sie ein neuronales Netzwerk alle Agenten eines Teams gleichzeitig steuern. Sie nennen diese Ansätze dabei Independent Learners (IL) und Joint-Action Learners (JAL).

Die Agenten im IL-Ansatz haben keinen zusätzlichen internen Kommunikationskanal, sodass der Weltzustand nicht einzig und allein von dem beobachtenden Agenten abhängig ist, sondern von versteckten, zufälligen und über die Zeit verändernden Variablen (den *Policies* der anderen Agenten). Dies bedeutet, die Welt ist als sogenannter nicht-stationärer Prozess

zu verstehen. Wenn jeder Agent den gleichen Beobachtungs- und Aktionsraum verwendet, kann aber dieselbe Policy über alle Agenten gelernt werden. Laut Ocana u. a. (2019) sei bereits durch andere Publikationen wie Kalyanakrishnan und Stone (2010) bekannt, dass dieses Vorgehen zu hinreichend guten Ergebnissen führen kann. Durch den Ansatz von OpenAI u. a. (2019) ist aber auch gezeigt, dass das Problem mit verschiedenen Policies pro Agent mehr als zufriedenstellend gelöst werden kann.

Die Joint-Action Learners tauschen einen großen Teil der Probleme einer nicht-stationären Welt durch schlechtere Skalierbarkeit aus. Dadurch, dass ein Netzwerk nun n Agenten steuern muss, wachsen Beobachtungs- und Aktionsraum linear mit der Anzahl der Agenten. Dementsprechend muss die Komplexität der benutzten Policy im Vergleich zur IL-Policy erhöht werden – dafür haben die Agenten nun einen impliziten internen Kommunikationskanal, was zu theoretisch perfektem Teamspiel führen kann.

Beide Ansätze haben sie mit dem gleichen Experiment evaluiert. Anstatt eines kompletten Fußballspiels simuliert ihre Umgebung eine sogetaufte „*2 vs. 2 offensive free-kick* "-Aufgabe. Wie der Name vermuten lässt, spielen zwei angreifende Roboter gegen zwei verteidigende Roboter auf einer einzelnen Spielfeldhälfte.

Die verteidigenden Spieler sind dabei Teil der Umgebung und folgen einer handprogrammierten Vorgehensweise: bis der Ball sich nicht bewegt, bewegen sie sich selbst auch nicht. Sobald dies der Fall ist, versuchen sie ihn so schnell wie möglich zu erreichen – der Torwart der beiden verlässt dabei nie seine Torlinie. Die angreifenden Spieler werden durch eine zusammengesetzte Belohnungsfunktion bewertet, welche sich aus Annäherungen von Robotern zum Ball, Annäherung des Balles zum Tor und einem großen Bonus im Falle eines Tores zusammensetzt. Das eigentliche Ziel der Angreifer ist es, den Ball in das Tor zu befördern.

Die Roboter nehmen ihre Welt durch einen sehr kleinen Vektor von 18 Zahlen wahr (Vgl. Abbildung 3.3), welcher die komplette Information der Welt beinhaltet, inklusive eines Zeitstempels, welcher die Markow-Eigenschaft der Umgebung sicherstellt.

Sie interagieren durch einen fünfzahligen Vektor mit der Welt, mit dem sie die Entscheidung treffen können, ob sie jetzt treten oder laufen – und wenn sie laufen, wohin.

Die Architektur des gewählten neuronalen Netzwerkes fällt dabei mit drei vergleichsweise kleinen Kernschichten nicht tief aus – eine Entscheidung,

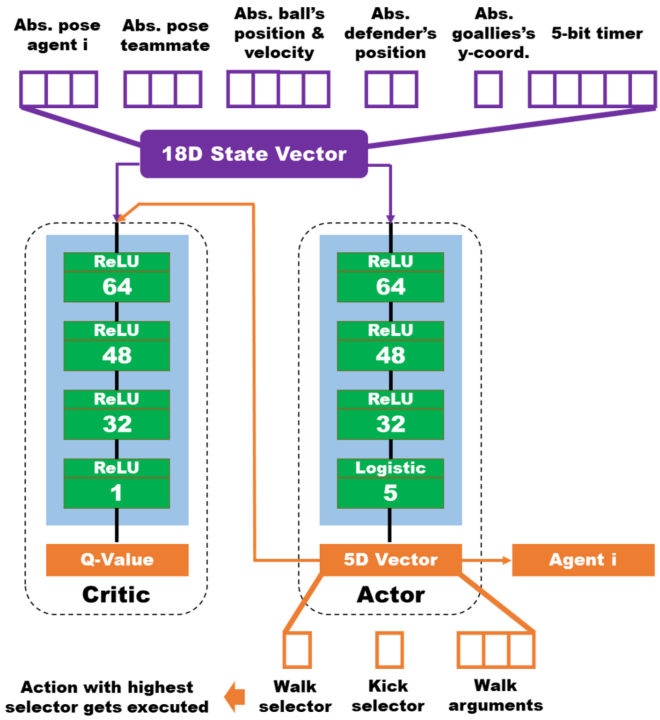


Abbildung 3.3: Die Netzwerkarchitektur des IL-Ansatzes. Die Netzwerkarchitektur des JAL-Ansatzes ist die gleiche, mit konkateniertem Ein- und Ausgang – die einzelnen Rectified Linear Unit (RelU)-Schichten wurden nicht vergrößert.

die aber bewusst gefallen sei, da es sich ebenfalls um einen sehr kleinen Zustandsraum handle.

In Konklusion sehen die Autoren in ihren Ergebnissen noch Entwicklungspotential. Besonders der IL-Ansatz ist häufig in lokalen Minima stecken geblieben und hat weniger gute Ergebnisse geliefert als der JAL-Ansatz. Häufig auftretende lokalen Minima führen sie auf die handprogrammierte Belohnungsfunktion zurück, die teilweise zu ungewolltem Verhalten anreizen, wie dribbeln statt schießen. Sie erwähnen das üblicherweise auftretende Problem des verstärkenden Lernens, dass eine Anwendung nur so gut ist, wie die spezialangefertigte Belohnungsfunktion erlaubt und, dass sie in folgenden Arbeiten vermutlich komplett auf dieses Konzept verzichten wollen.

DIE FUSSBALLSIMULATION

Der erste Schritt zur Lösung eines Problems des verstärkenden Lernens ist die Wahl und der Aufbau der Umgebung – des *Environments*. Im Falle des Fußballspiels gibt es bereits einige vorgefertigte, teilweise quelloffene Lösungen anderer Forschungsarbeiten (Kurach und Bachem, 2019; The Learning Agents Research Group, 2019).

Um vollständige Kontrolle über die Umgebung zu bekommen, sodass die verschiedensten Ansätze in Beobachtungsräumen, Aktionsräumen und Belohnungsfunktionen ausprobiert werden können, fiel am Ende die Entscheidung, die Fußballspielsimulation von Grund auf selbst zu entwicklen.

4.1 Die Gym-Schnittstelle

OpenAI (2019a) haben einen allgemeinen Standard einer Softwareschnittstelle (API) auf Basis des Markow-Entscheidungsproblems ausformuliert, den eine Umgebung im Kontext des verstärkenden Lernens einhalten kann und sollte, um Kompatibilität mit verschiedensten Lernalgorithmen sicherzustellen. Durch diese API ist es möglich, die Entwicklung von Problemstellung und Lernalgorithmus komplett zu trennen.

Die sogenannte *Gym*-Schnittstelle setzt damit folgende Attribute und Funktionen voraus:

- `observation_space: gym.Space`
  `action_space: gym.Space`

  Diese beiden Attribute beschreiben den Beobachtungs- und Aktionsraum der Umgebung, damit interagierende Algorithmen bereits im Voraus benötigten Speicher allozieren und die korrekten Aktionen bereitstellen können. (Sub-)klassen von `gym.Space` kodieren, ob Daten in diskreter Form, als Vektor, Matrix oder Tensor vorliegen.

- `reward_range: Integer, Integer`

Dieses Intervall beschreibt die untere und obere Grenze der möglichen Belohnungen pro Zeitschritt.

- `reset(): Observation`

  Diese Funktion (re-)initialisiert die Umgebung und gibt die erste Beobachtung o_0 zurück.

- `step(action: Action): Observation, Reward, Done, Info`

  Diese Funktion ist das Kernstück einer jeden Umgebung und darf erst nach einem initialen Aufruf von `reset()` benutzt werden. Sie nimmt eine mit dem Aktionsraum konforme Datenrepräsentation einer Aktion für den aktuellen Zeitschritt a_t an und verändert daraufhin den internen Zustand der Welt auf dieser Basis; sie führt die gewählte Aktion aus.

  Zurückgegeben wird ein 4-Tupel aus wichtigen Informationen:

  - `Observation`: Die neue Beobachtung o_{t+1}
  - `Reward`: Die Belohnung r_t, die die direkte Auswirkung von a_t auf die Welt quantifiziert
  - `Done`: Ein Wahrheitswert, der beschreibt, ob die Umgebung mit dem Zeitschritt t beendet wird
  - `Info`: Nicht genauer spezifizierte zusätzliche Information, die möglicherweise außerhalb des Lernalgorithmus' gebraucht wird

- `render(mode: String = "human")`

  Um die Geschwindigkeit des Lernvorgangs zu optimieren, ist das tatsächliche grafische Darstellen der Umgebung von der Logik abgespaltet. Der Lernalgorithmus kann also problemlos eine große Menge an Umgebungen instanziieren und diese ressourcensparend verwalten.

  `render()` ist die Methode, die aufgerufen werden muss, um eine grafische Darstellung des aktuellen Zeitschrittes zu erzeugen. Der Parameter `mode` ist dabei ein String und soll die Art der Darstellung wählen: üblich ist `"human"`, manchmal wird auch `"ascii"` benutzt. Eine tatsächlich standardisierte Bedeutung haben die Zeichenketten dabei nicht.

- `close()`

Diese Methode kann dazu genutzt werden, Ressourcen freizugeben.

- `seed(seed: Seed)`

  Nicht jede Umgebung verhält sich komplett deterministisch. Wenn ein *Environment* Pseudo-Zufallsgeneratoren benutzt, sollen diese mit dem hier übergebenen *Seed* initialisiert werden.

4.2 Umgang mit Multi-Agenten-Umgebungen

Die Gym-API sieht nicht vor, dass eine Umgebung mehr als einen Agenten beinhaltet. Für die Implementierung der eigenen Fußballsimulation ist dieser Standard allerdings leicht erweiterbar: anzupassen sind `reset()` und `step()` so, dass Aktionen, Beobachtungen und Belohnungen als Mengen vorliegen, die jedem Roboter jeweils einen Wert zuordnen. Diese Anpassungen sind ausreichend, da die Annahme getroffen werden kann, dass sich Beobachtungs- und Aktionsraum eines jeden Agenten deckungsgleich verhalten.

Der in dieser Arbeit präferierte Lernalgorithmus Proximal Policy Optimization (PPO) profitiert per Definition davon, möglichst viele *Trajektorien* von möglichst vielen Agenten aus üblicherweise möglichst vielen parallel simulierten Umgebungen zu sammeln. Auch, wenn die Gym-API diesen Umstand nicht einplant und daher die in der Arbeit genutzte Fremdimplementierung des Lernalgorithmus' von Hill u. a. (2019) ebenfalls nicht auf Multi-Agenten-Umgebungen ausgelegt ist, können diese Anpassungen leicht integriert werden – die Hauptsache ist die Sammlung der einzelnen Trajektorien.

Wie sich im Laufe der Arbeit abgezeichnet hat, kann es aber zu Problemen führen, wenn Trajektorien von zwei miteinander konkurrierenden Agenten zur Anpassung der Gewichte des selben neuronalen Netzwerkes benutzt werden – zumindest, wenn die zusammenaddierten Belohnungen beider Seiten eine Nullsumme bilden.

Infolgedessen fiel die Entscheidung, vorerst das Problem dadurch zu umgehen, dass sich der Vorgehensweise von Ocana u. a. (2019) bedient wird, indem Beobachtungs- und Aktionsräume konkateniert werden und Belohnungsfunktionen das ganze Team betreffen. Das entspricht einer Transformation eines Multi-Agenten-Problems in ein normales Problem des verstärkenden Lernens.

4.3 Umfang

Gleich zu Beginn ist festzulegen, in welchem Detailreichtum sich die Simulation bewegen soll; es läge theoretisch die ganze Welt der Simulationssoftware offen. Die Entscheidung fällt aber darauf, die Umgebung weniger komplex als tatsächlich möglich zu implementieren. Der Hauptgrund dafür ist ein weiteres Mal die Limitation der verfügbaren Hardware (Vgl. Abschnitt 1.1), aber auch, dass eine vollwertige Fußballsimulationssofware im dreidimensionalen Raum genug Arbeitsaufwand für ein weiteres umfangreiches Projekt beinhaltet.

Unter Zuhilfenahme der Physiksimulationssoftware *Box2D* entstand die *„MiniSoccer“* getaufte zweidimensionale Fußballsimulation (Catto, 2019). Abbildung 4.1 stellt einen Screenshot der grafischen Darstellung der entwickelten Umgebung dar.

Roboter werden als Rechtecke umgesetzt, der Ball als Kreis; die Kollisionserkennung und andere relevante Funktionen werden an die Physiksimulation ausgelagert. Durch die Simplifizierung der Roboterdarstellung gibt es keinen Hardwareverschleiß, wie er im echten Roboterfußball auftritt; in der Simulation sind alle Roboter baugleich.

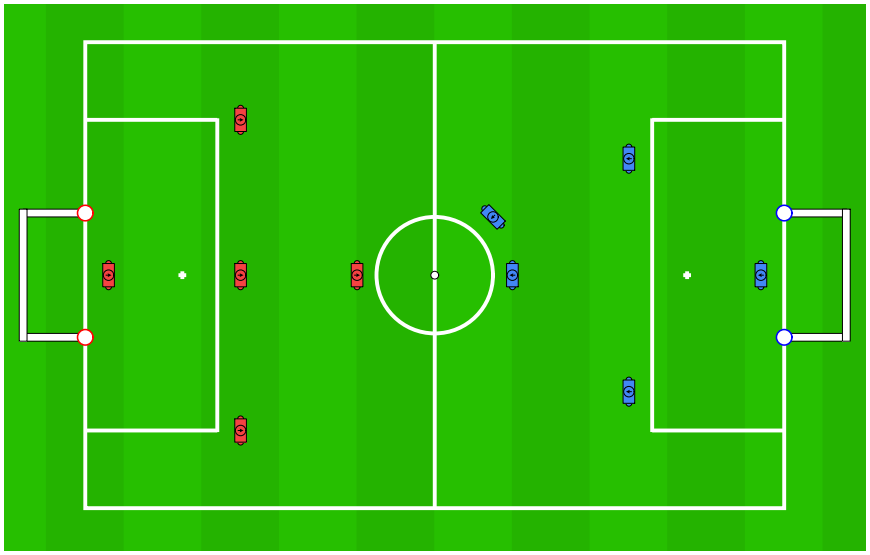


Abbildung 4.1: Eine beispielhafte Feldkonfiguration, bei der sich zwei Teams mit je fünf Robotern bei einer Anstoßsituation gegenüber stehen. Die dargestellten Hände der Roboter sind rein visueller Natur und haben keinerlei Auswirkung auf jegliche Kollision der Physiksimulation.

4.3.1 Spielregeln

Die weiter unten beschriebene Konfigurierbarkeit der Umgebung erlaubt es schnell, Spielregeln wie beispielsweise Anstöße oder Auszeiten einzelner Roboter einzubauen. Je nach Problemstellung können diese ein- oder ausgeschaltet – oder sogar durch neue Logik erweitert werden.

4.4 Architektur

Besonders wichtig bei der Entwicklung ist, dass sich mit wenig Aufwand verschiedene Experimente formulieren lassen; dem einher geht ein hohes Maß an möglicher Konfiguration. Wie die Architekturübersicht in Abbildung 4.2 zeigt, besteht eine Instanz der *MiniSoccer*-Umgebung aus einer Parametrisierung, mit der sich ein Experiment komplett beschreiben lässt.

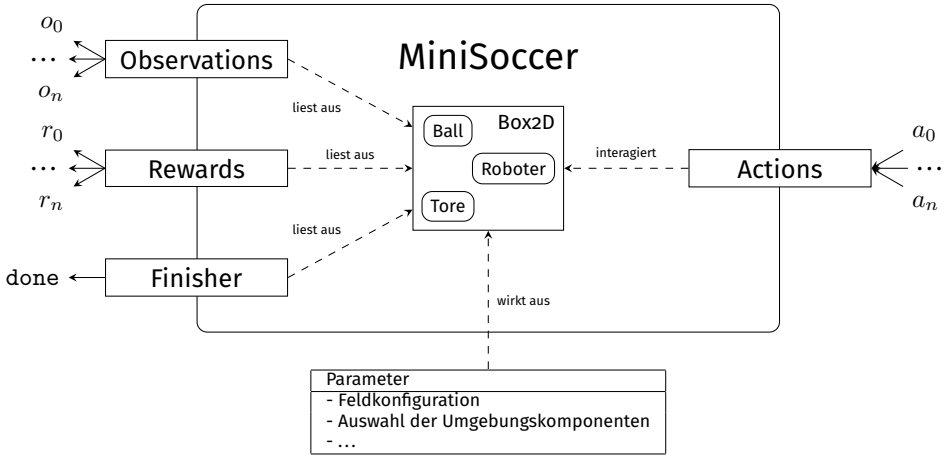


Abbildung 4.2: Beispielhafte Darstellung der Architektur der *MiniSoccer*-Umgebung, die konform mit der *Gym*-API ist.

Diese Parametrisierung besteht aus zwei Hauptteilen; der Feldkonfiguration und der Auswahl der einzelnen Umgebungskomponenten.

4.4.1 Feldkonfiguration

Die Feldkonfiguration umfasst hauptsächlich Informationen, die die physikalischen Gegebenheiten des Experiments beschreiben:

- Die Konstanten, welche den Aufbau der Welt und Eigenschaften der Objekte beschreiben:
  - Länge und Breite des Feldes
  - Größe der Tore
  - Größe, Masse und Reibungseigenschaften des Balles
  - Größe, Masse und Reibungseigenschaften der Roboter
  - Anzahl der Roboter pro Team
  - Maximale Bewegungsgeschwindigkeiten (Vor-, rück-, seitwärts, Drehung um die eigene Achse)
- Die initialen Positionen:
  - Position des Balles
  - Posen der Agenten von Team 1
  - Posen der Agenten von Team 2

Pose

Als *Pose* wird die räumliche Lage als Komposit von Position und Ausrichtung eines Objektes bezeichnet. Im zweidimensionalen Raum dieser Arbeit ist eine Pose als Tupel von Koordinaten und der Rotation des Objektes definiert.

$$\text{Pose2D} := \langle x, y, \alpha \rangle$$

Die Weltkonstanten können, müssen aber nicht für jedes Experiment definiert werden. In letzterem Fall fallen sie auf ihre normalen Belegungen zurück: gewählte Parameter, um den Originalbedingungen in der Standard Platform League (SPL) maximal nahe zu kommen. Das bedeutet eine Feldgröße von 6×9 m, eine Torbreite von 1,6 m und ein Ballradius von 5 cm (RoboCup Technical Committee, 2019). Dazu kommen $15 \times 7{,}5$ cm-Roboter, welche sich mit 22 mm/s vorwärts, seit- und rückwärts aber nur mit 11 mm/s bewegen können. Rotationen sind mit einer Maximalgeschwindigkeit von 5° möglich.

Positionen können sowohl als Konstanten als auch als Zufallsverteilungen angegeben werden, aus denen bei jeder Neuinitialisierung der Umgebung gezogen wird, zum Beispiel bei einem Aufruf von `reset()`.

4.4.2 Umgebungskomponenten

Die Wahl der Umgebungskomponenten definiert zusammen mit der Parametrisierung ein Experiment. Diese sind schlussendlich die Teile der Umgebung, welche als Schnittstelle zwischen Fussballsimulation und Lernalgorithmus fungieren. Auf Code-Ebene sind die Komponenten zu implementierende *Interfaces*, welche alle zusammen eine valide `Gym`-Anbindung formen:

- **Aktionsanwender** (`Actions`)

  Die Komponente des Aktionsanwenders stellt das Attribut des Aktionsraumes der Umgebung. Weiterhin ist sie dazu da, die von den Agenten gegebenen Aktionsvektoren zu interpretieren und in der physikalischen Welt umzusetzen.

- **Beobachtungsfunktion** (`Observations`)

  Die Beobachtungsfunktion stellt das Attribut des Beobachtungsraumes der Umgebung und berechnet für jeden Agenten eine Repräsentation dessen, wie ebenjener die Welt wahrnimmt.

- **Belohnungsfunktion** (`Rewards`)

  Die Belohnungsfunktion stellt das Intervall, welches den Zahlenbereich der Belohnungsfunktion beschreibt und berechnet für jeden Agenten die Belohnung, die er für den aktuellen Zustand der physikalischen Welt bekommen würde.

- **Beendungsfunktion** (`Finisher`)

  Die Beendungsfunktion entscheidet, ob eine Episode beendet wird oder nicht. Dies gilt für alle Agenten einer Simulation gleichzeitig.

4.5 Koordinatensysteme

Die in der Simulation benutzten Koordinatensysteme sind beinahe analog zu den in der Software von *B-Human* (Röfer u. a., 2019) integrierten Koordinaten definiert. Der Ursprung des Feldkoordinatensystems eines jeden Roboters liegt auf dem Feldzentrum; die x-Achse zeigt – teamabhängig – vorwärts auf das gegnerische Tor, die y-Achse zeigt – anders als bei Röfer u. a. (2019) – nach rechts (Vgl. Abbildung 4.3). Eine z-Achse gibt es nicht, da sich die Simulation in reinen zwei Dimensionen bewegt. Rotationen im

Feld bewegen sich gegen den Uhrzeigersinn, wobei die x-Achse nach 0° und die y-Achse nach 90° zeigt.

Transformation zwischen den Koordinatensystemen ist durch einfache Verschiebung um die Ursprünge und wenn nötig durch eine Spiegelung möglich.

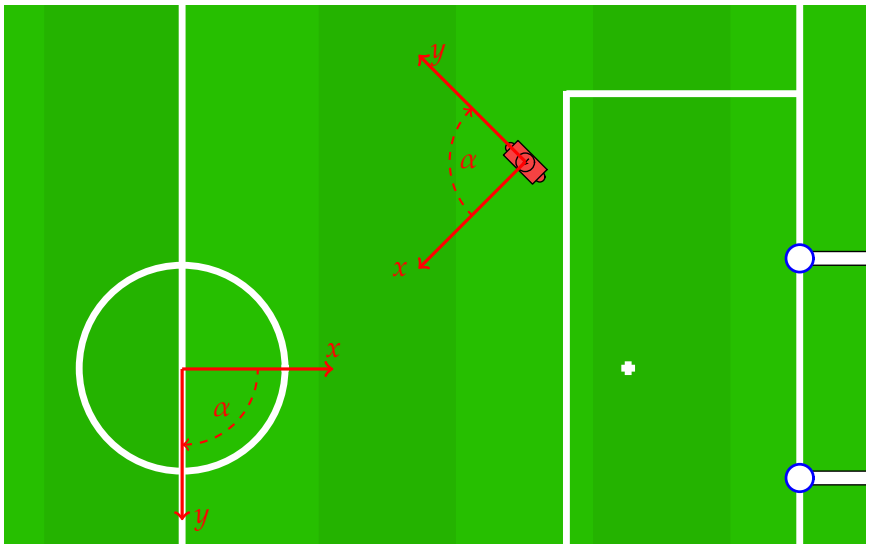


Abbildung 4.3: Die Darstellung von Teamfeld- und Roboterkoordinaten.

4.6 Aktionsräume

Für die Gestaltung des Aktionsraums eines jeden Roboter gibt es nicht viel Spielraum: Er soll sich in seiner Umgebung bewegen und den Ball treten können. Der in dieser Arbeit genutzte Aktionsraum ist ein kontinuierlicher Vektor von vier Zahlen.

Auf Versuche mit einem diskreten Aktionsraum wurde aus Zeitgründen verzichtet, darauf wird in Kapitel 6 noch eingegangen.

Frei nach Baker u. a. (2019) und da die Roboter keine tatsächlichen Beine haben, ist die Bewegung des Roboters nur über drei Zahlen definiert, mit denen ein Agent jeden Spielschritt seine Geschwindigkeit setzen kann.

$$A_{\text{move}} = \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{\alpha} \end{pmatrix} \text{ mit } \bar{x}, \bar{y}, \bar{\alpha} \in [-1, 1] \tag{4.1}$$

Die Steuerungszahlen sind alle auf das Intervall $[-1, 1]$ skaliert und beschnitten, da neuronale Netze erfahrungsbedingt die besten Ergebnisse mit normalisierten Daten erzielen. Die neue Rotationsgeschwindigkeit des Roboters $\vec{\alpha}_t$ errechnet sich durch eine einfache Multiplikation mit der in der Umgebung konfigurierten Konstante der Maximalrotationsgeschwindigkeit α_{max}.

$$\vec{\alpha}_t = \bar{\alpha}_t \cdot \alpha_{\text{max}} \tag{4.2}$$

Die neue Bewegungsgeschwindigkeit des Roboters $\vec{v}_{\text{robot}} := (\vec{x}\ \vec{y})^T$ ist aufgrund zweier Dinge nicht mit einer einfachen Multiplikation gelöst; die Vorwärts-Maximalgeschwindigkeit eines Roboters ist nicht die gleiche wie die Seit- bzw. Rückwärtsgeschwindigkeit – und $\bar{x}$ und $\bar{y}$ dürfen nicht unabhängig voneinander berechnet werden, da der Roboter sonst die maximale Geschwindigkeit erreichen würde, wenn er schräg läuft ($\bar{x} = \bar{y} = 1$).

Die implementierte Transformation von $\bar{x}, \bar{y}$ nach $\vec{v}_{\text{robot}}$ wird in Abbildung 4.4 dargestellt.

Der vom Roboter gewählte Bewegungsvektor $\bar{v}_{\text{robot}} := (\bar{x}\ \bar{y})^T$ wird auf eine Kreisform beschnitten, sodass $0 \leq |\bar{v}_{\text{robot}}| \leq 1$. Der beschnittene Vektor wird abschließend unter Berücksichtigung der erlaubten Maximalgeschwindigkeiten auf die komplette Bewegungsreichweite skaliert.

Damit der Roboter den Ball nicht nur schieben, sondern auch treten kann, wird der Aktionsraum noch um einen weiteren Faktor $\bar{k} \in [0, 1]$ erweitert. Ist der Ball nicht in der Nähe des Roboters, hat dieser Aktionswert keine Auswirkung auf die Umgebung. Liegt der Ball vor dem Roboter, wird eine Kraft $\dot{k} = \bar{k} \cdot k_{\text{max}}$ ausgerechnet, welche auf den Ball in Sichtrichtung des Roboters wirkt.

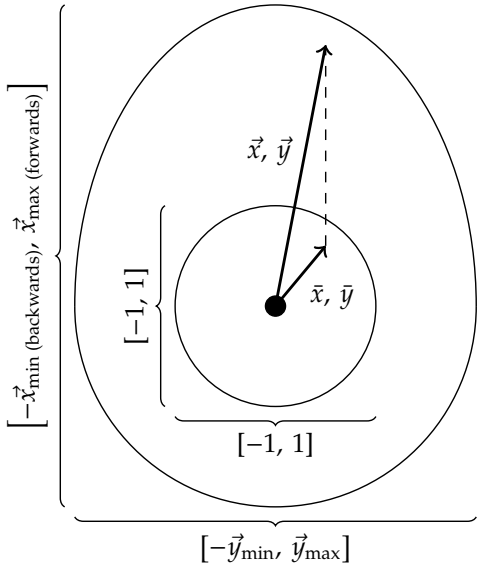


Abbildung 4.4: Die Visualisierung der Transformation von $(\bar{x}, \bar{y})$ zu $(\vec{x}, \vec{y})$ aus der Vogelperspektive. Im Ursprung ist sich ein Roboter vorzustellen, welcher nach oben ausgerichtet ist.

Der finale, in dieser Arbeit eingesetzte Aktionsraum lässt sich folgendermaßen notieren:

$$A = \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{\alpha} \\ \bar{k} \end{pmatrix} \text{ mit } \bar{x}, \bar{y}, \bar{\alpha} \in [-1, 1] \text{ und } \bar{k} \in [0, 1] \tag{4.3}$$

4.7 Beobachtungsräume

Für den Beobachtungsraum eines Agenten gibt es viele verschiedene mögliche Ansätze. Der simpelste Ansatz ist der, den Ocana u. a. (2019) verfolgen: eine einfache Aneinanderreihung der Weltinformationen.

Die Welt kann dabei mit einer vergleichsweise kleinen Menge an Zahlen beschrieben werden, die Position in kartesischen Feldkoordinaten des Balles, der Torpfosten und die Posen aller Roboter. Roboter und der Ball besitzen zusätzlich noch eine momentane Geschwindigkeit, mit der sie sich über das Feld bewegen.

Es lässt sich damit folgende stark abstrahierte Notation aufstellen:

$$O_{\text{naïve}} = \begin{pmatrix} p(\text{ball}) \\ p(\text{roboter}) \\ p\big(\text{teammitglied}_1\big) \\ \dots \\ p\big(\text{teammitglied}_n\big) \\ p\big(\text{gegner}_1\big) \\ \dots \\ p\big(\text{gegner}_n\big) \\ p\big(\text{zieltor pfosten}_1\big) \\ p\big(\text{zieltor pfosten}_2\big) \\ p\big(\text{eigentor pfosten}_1\big) \\ p\big(\text{eigentor pfosten}_2\big) \end{pmatrix} \quad \text{mit } p(\kappa) = \begin{pmatrix} \kappa_x \\ \kappa_y \\ \kappa_{\dot{x}} \text{ wenn } \exists \kappa_{\dot{x}} \\ \kappa_{\dot{y}} \text{ wenn } \exists \kappa_{\dot{y}} \\ \kappa_{\alpha} \text{ wenn } \exists \kappa_{\alpha} \end{pmatrix} \tag{4.4}$$

In der reinen Theorie sind dies alle nötigen Informationen, um Fußball spielen zu können. Indirekte Feldkonfigurationen wie der Ballradius, Roboter- und Torgrößen könnten durch Interaktion mit der Welt mitgelernt werden.

In der Praxis birgt dieser Ansatz aber einige Probleme. Wie bereits von Ocana u. a. (2019) beschrieben, kann bei dieser Darstellung die Fußballfeldsymmetrie nicht genutzt werden – ganz allgemein können neuronale Netzwerke mit dieser Art Information nicht gut generalisieren. Das liegt daran, dass die Distanz zweier Punkte im Zustandsraum nur im extremen Nahbereich aussagen kann, wie verschieden zwei Zustände tatsächlich sind. Um gute Ergebnisse mit diesem Beobachtungsraum zu erzielen, werden vermutlich Netzwerke mit Tausenden von Eingangsneuronen benötigt, um gezielt auf einzelne Konfigurationen des Feldes reagieren zu können.

Weiterhin ist ein auf diesem Beobachtungsraum trainiertes Netz nicht auf eine neue Feldkonfiguration übertragbar. Die Anzahl der einzelnen Roboter ist daher komplett festgelegt und nicht variabel.

4.7.1 Erweiterung durch Polare Koordinaten

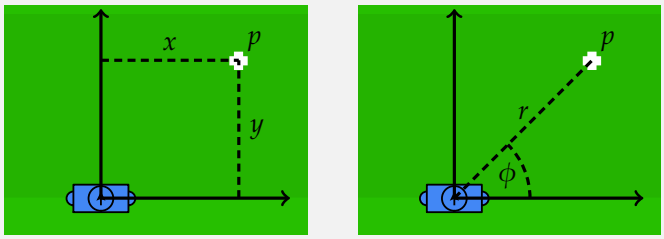
Kartesische Koordinaten lassen sich aus dem gleichen Grund nicht gut mit Neuronen verarbeiten; die einzelnen Komponenten eines Vektors im kartesischen Raum sagen sehr wenig über die tatsächliche Position aus. Neurona-

le Netzwerke funktionieren am besten, wenn sie auf gewisse Schwellwerte reagieren können, das ist wegen dieser Eigenschaft nicht einfach, es werden viele Linearkombinationen von Schwellwerten benötigt, die sich einzelne Neuronen teilen müssen.

Ein möglicher Lösungsansatz dazu wäre es, die einzelnen Koordinaten ins polare Koordinatensystem des Roboters zu transformieren.

Polare Koordinaten

Anders als kartesische Koordinaten (links) stellen polare Koordinaten (rechts) einen Punkt p nicht als zwei Abstände x und y vom Ursprung dar, sondern durch einen Radius r (welcher die Distanz des Punktes zum Ursprung bezeichnet) und den Winkel ϕ dorthin.



Umrechnungen der Koordinatensysteme sind folgendermaßen möglich.

$$x = r \cos \phi \tag{4.5}$$

$$y = r \sin \phi \tag{4.6}$$

$$r = \sqrt{x^2 + y^2} \tag{4.7}$$

$$\phi = \begin{cases} \arctan \frac{y}{x} & \text{wenn } x > 0 \\ \left(\arctan \frac{y}{x}\right) + \pi & \text{wenn } x < 0,\ y \geq 0 \\ \left(\arctan \frac{y}{x}\right) - \pi & \text{wenn } x < 0,\ y < 0 \\ +\pi \cdot \frac{1}{2} & \text{wenn } x = 0,\ y > 0 \\ -\pi \cdot \frac{1}{2} & \text{wenn } x = 0,\ y < 0 \end{cases} \tag{4.8}$$

Die Fallunterscheidungen in Gleichung (4.8) sind in Mathematikbibliotheken üblicherweise als `atan2(x, y)` implementiert.

Die einzelnen Komponenten eines Vektors im polaren Koordinatensystem sind aussagekräftiger als die des kartesischen. Im Falle eines neuronalen Netzes können so einzelne Neuronen die Distanzen der Objekte für taktische Entscheidungen beobachten und andere Neuronen entscheiden, wohin man sich drehen müsste, um zum Beispiel zum Ball zu blicken.

$$O_{\text{polar}} = O_{\text{naïve}} \text{ mit } p(\kappa) = \begin{pmatrix} \kappa_r \\ \kappa_\varphi \\ \kappa_{\dot{x}} \text{ wenn } \exists \kappa_{\dot{x}} \\ \kappa_{\dot{y}} \text{ wenn } \exists \kappa_{\dot{y}} \\ \kappa_\alpha \text{ wenn } \exists \kappa_\alpha \end{pmatrix} \tag{4.9}$$

Da die Koordinaten ins Roboterkoordinatensystem transformiert werden, kann die Feldsymmetrie ausgenutzt werden. In dieser Darstellung liegen lokale Feldkonfigurationen der gleichen Spielsituation immer gleich vor, sodass die Neuronen auch situationsbedingt reagieren können.

Im Roboterkoordinatensystem sind die Koordinaten des beobachtenden Roboters natürlich immer $(0, 0)$ – auch im kartesischen System. Diese können infolgedessen aus der Beobachtung weggelassen werden. Die Selbstlokalisierung des Roboters ist zumindest theoretisch dadurch sichergestellt, dass die Positionen der einzelnen Torpfosten im Beobachtungsraum enthalten sind.

Damit eine Korrelation zwischen Beobachtungs- und Aktionsraum existiert, müssen die in Abschnitt 4.6 beschriebenen Aktionsräume so angepasst werden, dass sie ebenfalls in Roboterkoordinaten agieren.

4.7.2 Normalisierung

Eine der wichtigsten Faustregeln beim Arbeiten mit neuronalen Netzwerken ist, dass Eingangsdaten normalisiert sein und einen Durchschnitt nahe Null halten sollten. Sowohl bei kartesischen als auch bei polaren Koordinaten ist dies relativ einfach möglich, da mini- und maximal mögliche Werte errechnet werden können – auf dem Feld gibt es eine maximal mögliche Distanz der Felddiagonale (nach Pythagoras), Rotationen bewegen sich im Bereich von einer kompletten Rotation $[0, 360)^\circ$.

$$O_{|\text{polar}|} = O_{\text{naïve}} \text{ mit } p(\kappa) = \begin{pmatrix} \kappa_r / r_{\max} \\ \kappa_\varphi / \varphi_{\max} \\ \kappa_{\dot{x}} \text{ wenn } \exists \kappa_{\dot{x}} \\ \kappa_{\dot{y}} \text{ wenn } \exists \kappa_{\dot{y}} \\ \kappa_\alpha \text{ wenn } \exists \kappa_\alpha \end{pmatrix} \tag{4.10}$$

Sofern nicht anders verzeichnet, ist das der in dieser Arbeit verwendete Beobachtungsraum.

ZWEI GEGEN ZWEI

Als Auswertungsszenario wurde ein Zwei-gegen-Zwei beinahe analog zu Ocana u. a. (2019) durchgeführt. Wie ursprünglich in Abschnitt 3.4 beschrieben, spielen zwei angreifende Roboter gegen zwei verteidigende Roboter. Das Ziel ist es, dass das angreifende Team den Ball in das Tor der Verteidiger bewegt.

5.1 Umgebungsaufbau

Die initialen Positionen aller Objekte der Umgebung sind in Abbildung 5.1 dargestellt. Der Ball liegt fest auf dem Punkt (1,0; 2,0), ein Angreifer wird nah davor auf (0,5; 2,5) platziert – mit einer zufälligen Rotation zwischen 0 und $\pi/4$ Radiant. Der zweite Angreifer steht in Passnähe, mit einer festen Position von $-0{,}5$ auf der y-Achse. Seine Position auf der x-Achse wird für jede Umgebungsinstanz uniform aus dem Intervall [0, 1,5] gezogen. Beide Verteidiger werden auf den festen Positionen (2,25; $-1{,}5$) und (4,25; 0,0) mit einer festen Rotation von π gesetzt.

Die Verteidiger sind Teil der Umgebung und folgen einer handprogrammierten Vorgehensweise. Beide halten still, bis die Angreifer den Ball bewegt haben. Sobald dies eintritt, versucht der vordere Verteidiger den Ball so schnell wie möglich zu erreichen. Der Torwart bewegt sich grundsätzlich auf den Schnittpunkt von der Balllaufbahn und seiner Torlinie zu, dabei kann er letztere nicht verlassen.

Eine Episode endet, sobald eins der folgenden vier Geschehnisse eintrifft:

i.) Es fällt ein Tor.

ii.) Es vergehen 600 Zeitschritte. Bei einer Aktionsrate von 30 hz entspricht das 20 Sekunden Spielzeit.

iii.) Ein verteidigender Spieler berührt den Ball.

iv.) Der Ball verlässt das Feld.

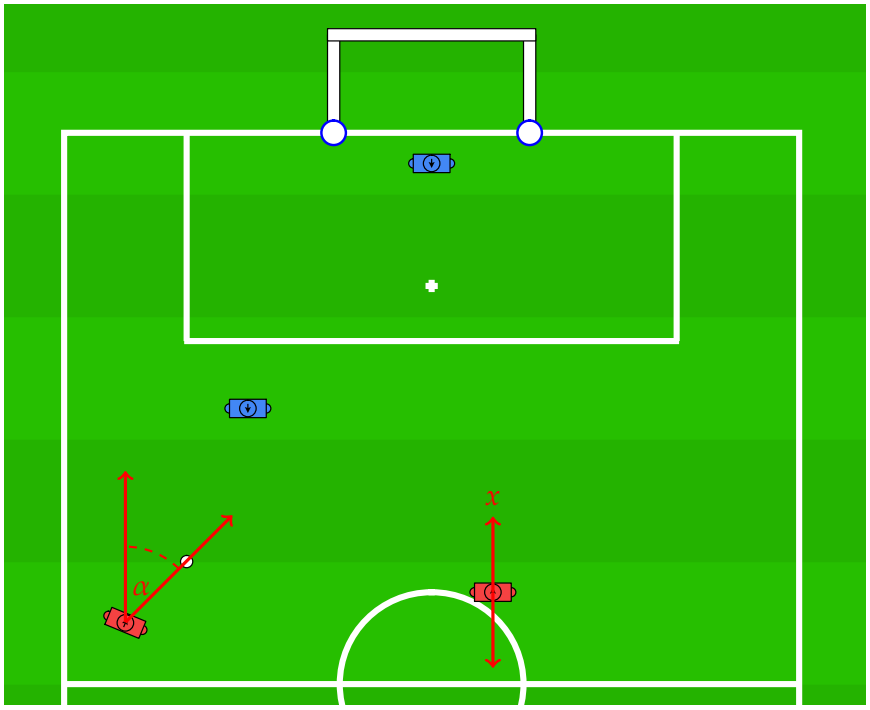


Abbildung 5.1: Eine Initialbelegung des beschriebenen Zwei-Gegen-Zwei-Szenarios. Eingezeichnet sind die uniform verteilten Stellschrauben, welche bei jeder Instanziierung neu gezogen werden.

5.2 Direktnachbau

Als initialen Lösungsversuch des definierten Szenarios ist es sinnvoll, die Herangehensweise von Ocana u. a. (2019) so gut wie möglich nachzubilden, damit direkte Vergleichswerte vorliegen. Da die besseren Ergebnisse durch die Nutzung ihres Joint-Action Learners (JAL)-Ansatzes erzielt wurden, bei dem alle Roboter vom selben neuronalen Netzwerk gesteuert werden, wird dieser auch hier angewandt. Dementsprechend werden die Beobachtungs- und Aktionsräume beider Angreifer konkateniert.

Die Belohnungsfunktion, welche den Spielzustand des ganzen angreifenden Teams bewertet, setzt sich als Linearkombination von mehreren Faktoren zusammen. In den folgenden Formeln beschreibt **p** jeweils eine zweidimensionale Position mit x- und y-Komponente. Weiterführend bezeichnen die Buchstaben A, B, L und R die **A**genten, den **B**all und den **L**inken sowie **R**echten Torpfosten. x^p bezeichnet die x-Position der Torlinie. Alle Notationen sind leicht verändert von Ocana u. a. (2019) übernommen.

$$R = \sum_i D_{A_i}^B + \max\left(\left\{D_B^{A_i},\ \forall i\right\},\ D_B^G\right) + G \tag{5.1}$$

mit

$$G = \begin{cases} 20 & \text{wenn ein Tor gefallen ist} \\ 0 & \text{sonst} \end{cases} \tag{5.2}$$

$$D_A^B = d\left(\mathbf{p}_t^B,\ \mathbf{p}_{t-1}^A\right) - d\left(\mathbf{p}_t^B,\ \mathbf{p}_t^A\right) \tag{5.3}$$

$$D_B^A = d\left(\mathbf{p}_{t-1}^B,\ \mathbf{p}_t^{A_i}\right) - d\left(\mathbf{p}_t^B,\ \mathbf{p}_t^{A_i}\right) \tag{5.4}$$

$$D_B^G = d_{\text{goal}}\left(\mathbf{p}_{t-1}^B\right) - d_{\text{goal}}\left(\mathbf{p}_t^B\right) \tag{5.5}$$

$$d(p_1,\ p_2) = \left\|p_2 - p_1\right\|_2 = \sqrt{\left(x^{p_1} - x^{p_2}\right)^2 + \left(y^{p_1} - y^{p_2}\right)^2} \tag{5.6}$$

$$d_{\text{goal}}\left(\mathbf{p}^B\right) = \begin{cases} d\left(\mathbf{p}^L,\ \mathbf{p}^B\right) & \text{wenn } y^B \geq y^L \\ d\left(\mathbf{p}^R,\ \mathbf{p}^B\right) & \text{wenn } y^B \leq y^R \\ \left|x^B - x^P\right| & \text{sonst} \end{cases} \tag{5.7}$$

Die einzelnen Faktoren sollen verschiedene Verhaltensweisen anreizen. Die Summe über die Distanzänderung beider Agenten zum Ball soll dafür sorgen, dass beide Roboter sich dem Ball nähern; besonders wenn letzterer nicht in Bewegung ist. G sorgt dafür, dass der letzte Zeitschritt einer Episode einen starken Ausschlag im Belohnungswert aufweist und somit wegen des Diskontierungsfaktors γ den Ertrag aller Zeitschritte kurz vor einem möglichen Tor erhöht.

Der zentrale Faktor der Belohnungsfunktion betrachtet die Bewegung des Balles auf dem Feld und belohnt die Agenten entweder mit der Bewegung des Balles auf einen der beiden Agenten oder das Tor zu, jeweils mit dem bestmöglichen Wert. Damit soll nicht nur die Annäherung des Balles in Richtung des Tores angereizt werden, sondern auch ein Pass.

Die genutzte Netzwerkarchitektur soll die gleiche sein, wie sie bereits von Ocana u. a. (2019) aufgestellt wurde. Für ihr JAL-Experiment haben sie die Architektur ihres Independent Learners-Ansatzes (Abbildung 3.3) wiederverwendet: Sowohl das *Value*- als auch das *Policy*-Netzwerk bestehen aus drei Kernschichten mit je 64, 48 und 32 Rectified Linear Unit-Einheiten.

Die benutzten Hyperparameter sind nicht einfach von Ocana u. a. (2019)

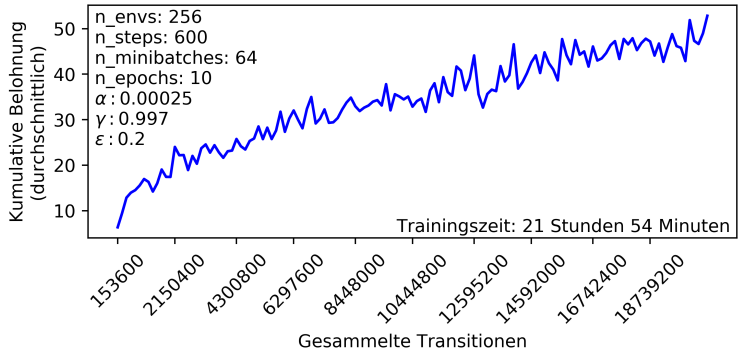
zu übernehmen, da dort ein anderer Trainingsvorgang verfolgt wird, für den andere Parameter nötig sind. Aus 256 parallel simulierten Zwei-gegen-Zwei-Umgebungen wurden jeweils *Trajektorien* der Länge 600 gesammelt – was der maximalen Länge einer möglichen Episode entspricht. Die Gewichte des neuronalen Netzwerkes werden somit alle $256 \times 600 = 153.600$ gesammelten *Transitionen* mit einer Lernrate α von 0,00025 angepasst. Ein Diskontierungsfaktor $\gamma = 0{,}997$ mit einer Halbwertszeit von $\ln 0{,}5 / \ln \gamma \doteq 230$ soll dafür sorgen, dass die Agenten auf ein Tor hinarbeiten, sollten sie es durch Exploration finden. Als Trimmfaktor ϵ werden vorerst die üblichen 0,2 genommen.

Auswertung

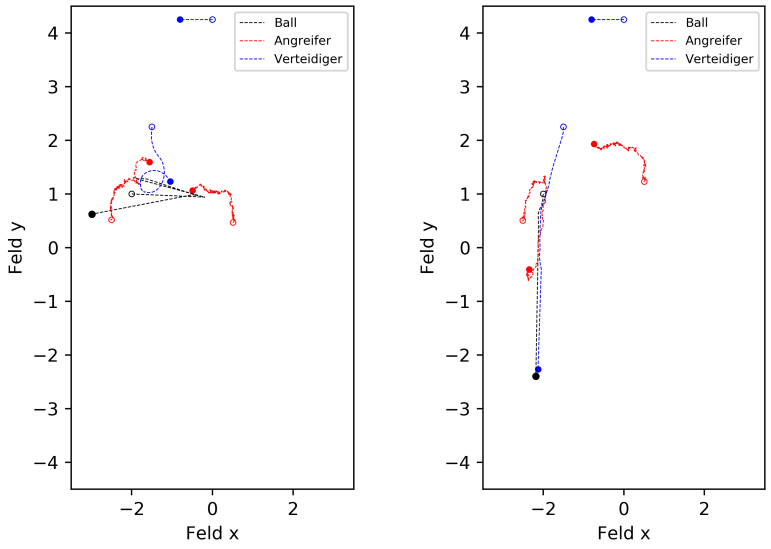
Nach zwei Stunden Training sehen einzelne Trajektorien der Policy bereits vielversprechend aus. Bei den Trajektorien, die nicht früh durch das Berühren des Balles durch einen Verteidiger beendet werden, zeichnen sich bereits nach zwei Stunden Training zwei Taktiken ab, wobei beide das gleiche Ziel verfolgen: die Zeit zu maximieren, die der Ball in Bewegung bleibt.

Zwei handausgewählte Trajektorien ebenjener Taktiken sowie eine Übersicht über den Trainingsverlauf sind in Abbildung 5.2 dargestellt. Obgleich sich bereits Taktiken entwickeln, ist ein Problem ersichtlich: der Ball scheint nicht in Richtung des Tores bewegt zu werden. Nach beinahe 22 Stunden Training ist zwar noch keine Konvergenz zu erkennen (Vgl. Abbildung 5.2a), aber die allgemeine Taktik des neuronalen Netzes fixiert sich auf die in Abbildung 5.2c abgebildete Strategie. In einem Großteil aller gesammelten Trajektorien wurde der Ball auf die eigene Spielfeldhälfte gebracht. Dieses Verhalten lässt sich unweigerlich sowohl auf die Belohnungsfunktion als auch auf die Explorationsmethode des Agenten zurückführen: Die Belohnungsfunktion aus Gleichung (5.1) bewertet die Bewegung des Balles zum Tor genauso gut wie die Bewegung des Balles zu einem Agenten der Angreifer. Dadurch wird ein Torschuss nur sehr schwach und indirekt angereizt; die Angreifer finden den Torschussfaktor G nicht und suchen sich andere Wege, möglichst gut bewertet zu werden.

Die Tatsache, dass hier nicht einfach die Erfolgsergebnisse von Ocana u. a. (2019) nachgebildet werden können, ist der höheren Komplexität von *Mini-Soccer* zu verschulden. Anders als die Agenten dort muss hier jeder Angreifer zu jedem Zeitschritt eine neue Entscheidung treffen. Dies erlaubt zwar filigranere Bewegungen, erhöht aber die Schwierigkeit der Umgebung.



(a) Das erste Training mit starker Nähe zu dem Ansatz von Ocana u. a. (2019). Es wurde frühzeitig beendet, noch bevor 24 Stunden Trainingszeit vergangen sind. Zu sehen ist der Median der zu einem Trainingszeitpunkt vorliegenden gesammelten Transitionen für den Lernalgorithmus. Benutzte Hyperparameter werden ebenfalls dargestellt.



(b) Taktik 1: Ausspielen des Verteidigers

(c) Taktik 2: Aktives Blocken des Verteidigers

Abbildung 5.2: Eine Trainingsübersicht und zwei ausgewählte Trajektorien, die das neuronale Netzwerk aus Abschnitt 5.2 nach etwa zwei Stunden Training produziert hat. Die Grafiken zeigen das Feld in der Vogelperspektive, das Zieltor ist oben. Links (5.2b) zu sehen sind mehrfache Pässe zwischen den Angreifern um den Verteidiger herum; rechts (5.2c) der Versuch eines Angreifers, den Ball möglichst lange Zeit von den Verteidigern fernzuhalten, indem der Ball in die andere Spielfeldhälfte bewegt wird.

5.3 Anreiz der Ballbewegung

Um die angesprochenen Probleme des Lösungsversuches aus Abschnitt 5.2 zu lösen, können drei Faktoren angepasst werden:

Das Explorationsverhalten der Agenten kann indirekt durch mehr parallel simulierte Umgebungen erhöht werden, da für jeden Trainingsschritt eine höhere Diversität an verschiedenen Trajektorien vorliegt. Im Zuge dessen werden mit 512 Fußballsimulationen doppelt so viele wie beim vorigen Experiment benutzt.

Der vorherig benutzte Diskontierungsfaktor γ von 0,997 hat mit seiner vergleichsweise hohen Halbwertszeit von $\sim$7 Sekunden möglicherweise dafür gesorgt, dass die Angreifer möglichst lange Episoden bevorzugt haben, anstatt diese durch einen Torschuss zu beenden. Ein kleiner Faktor von 0,99 mit einer Halbwertszeit von $\sim$2 Sekunden soll dies verhindern.

Die Belohnungsfunktion aus Gleichung (5.1) belohnt die Bewegung des Balles zum Tor nicht stark genug. Durch einen einfachen Multiplikator β, welcher vorerst auf 2 gesetzt wird, wird die Ballbewegung in Richtung des Tores den Pässen vorgezogen. Letztere werden dennoch belohnt, aber nicht präferiert.

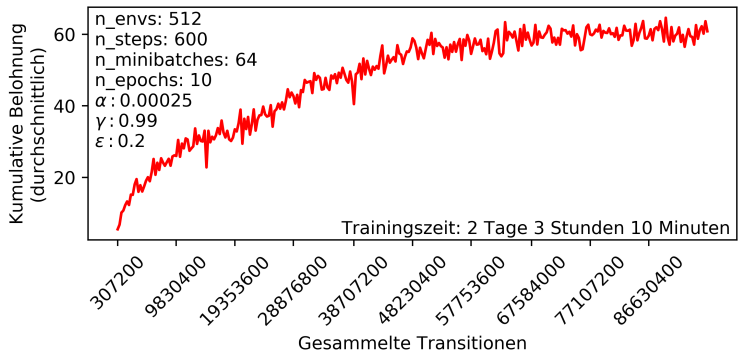
$$R_{\text{mod}} = \sum_i D^B_{A_i} + \max\left(\left\{D^{A_i}_B, \forall i\right\}, \beta D^G_B\right) + G \tag{5.8}$$

Die beschriebenen Anpassungen haben die gewünschten Auswirkungen. Bereits nach einer Stunde des Trainings entstehen Trajektorien, die durch eine Ballberührung des verteidigenden Torwartes beendet werden. Eine solche Episode ist in Abbildung 5.3b abgebildet. Trotz dessen, dass diese Art der Trajektorie keine Seltenheit ist und häufig auftritt, fällt kein tatsächliches Tor. Dies geschieht erstmalig eine Stunde später in etwa 0,2% aller Episoden, 77% enden durch Ballberührung der Verteidiger, der Rest durch den Ablauf der Zeit.

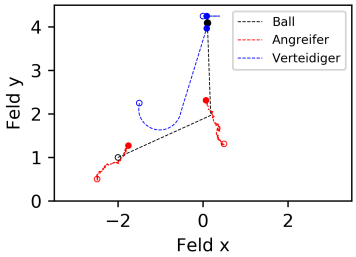
Auswertung

Nach vermeintlicher Konvergenz und zwei Trainingstagen lässt sich zwar eine Tendenz zur Lösung der Zwei-gegen-Zwei-Problemstellung erkennen, dennoch liegt der Fokus nicht auf Beendung durch tatsächlichen Torschuss.

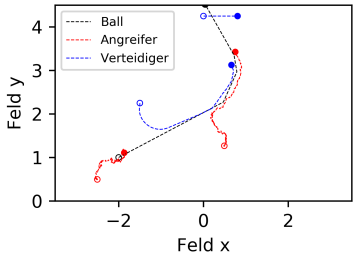
Aus 2048 simulierten Episoden endeten nur Sechs durch ein Tor, was einer Lösungsrate von gerade ein mal 0,29% entspricht.



(a) Eine Darstellung des Trainingsvorgangs. Konvergenz trat nach über 90 Millionen gesammelten Transitionen in zwei Tagen Trainingszeit ein.



(b) Eine Episode, wie sie bereits nach einer Stunde Training vorkam. Das angreifende Team beginnt schnell zu verstehen, dass der Ball in Richtung des Tores bewegt werden muss.



(c) Ein Torschuss nach zwei Trainingsstunden. Zu sehen ist eine ähnliche Vorgehensweise wie sie bereits in Abbildung 5.2c zum Einsatz kam, bei der sich der Angreifer direkt zwischen Verteidiger und Ball drängt, um Kontakt zu verhindern.

Abbildung 5.3: Eine Übersicht des in Abschnitt 5.3 beschriebenen Ansatzes.

5.4 Anreiz des Tores

Die Anpassungen aus Abschnitt 5.3 sind ein Schritt in die richtige Richtung, priorisieren aber den Torschuss selbst nicht. Dies ist ein allgemeines Problem des verstärkenden Lernens. Dadurch, dass das Problem durch Maximierung der kumulativen Belohnung über die Zeit betrachtet wird, ist eine frühe Beendigung der Episode nicht erstrebenswert. Im Szenario des Fußballspiels ist aber genau dies das Ziel, die Belohnungsmaximierung ist nur Mittel zum Zweck – die Hauptsache ist, dass der Ball vor Ablauf der Zeit in das Tor bewegt wird.

Durch weitere Anpassungen der Trainingsparameter kann dieses Ziel dennoch erreicht werden. Es muss dafür gesorgt werden, dass die höchstmöglich erreichbare kumulative Punktzahl die tatsächlich Höchstmögliche ist. Betrachten wir die Trajektorien der beiden vorgehenden Versuche, können zwei Beobachtungen getroffen werden:

i.) Fällt ein Tor, fällt es bereits vor Ablauf der ersten 300 Zeitschritte.

ii.) Die Angreifer präferieren es, den Ball aus der Reichweite der Verteidiger zu bewegen, wie es bereits in den Abbildungen 5.2b und 5.2c geschehen ist.

Punkt ii.) ist als direkte Folge von Punkt i.) zu betrachten. Durch die bisher gewählte maximale Episodenlänge von 20 Sekunden und der Tatsache, dass ein Tor bereits nach 10 Sekunden erzielt werden kann, ist die logische Konsequenz, dass ein schneller Torschuss nicht lukrativ ist – in den verpassten zehn Sekunden können mehr Belohnungen gesammelt werden. Der theoretisch beste Fall wäre auf Basis dieser Annahmen ein Torschuss im letztmöglichen Zeitschritt.

Um diesen Problemen entgegen zu wirken, können drei weitere zu den in Abschnitt 5.3 eingeführten Änderungen hinzugefügt werden:

- Das Zeitlimit wird um die Hälfte verringert, sodass die Agenten nur noch 10 Sekunden pro Episode zur Verfügung haben. Der Trainingsalgorithmus Proximal Policy Optimization (PPO) wird ebenfalls so angepasst, dass er Trajektorien der Länge 300 sammelt.
- Der Bonus, der bei einem Torschuss durch Gleichung (5.2) in die Belohnungsfunktion einfliesst, wird auf 100 erhöht.
- Der Diskontierungsfaktor γ wird erneut auf 0,997 mit einer Halbwertszeit von 230 Zeitschritten gesetzt. Dies hat die Folge, dass der

letztmögliche Zeitschritt noch zu $e^{300 \cdot \ln 0{,}997} \cong 40\%$ in die Entscheidungen der Angreifer einfließt.

Auswertung

Bereits früh im Trainingsvorgang kommt es zu Torschüssen. Nach acht Stunden ist eine Abschlussrate von knapp einem Prozent erreicht – das ist schon drei mal so hoch wie die vorhergehenden Lösungsversuche möglich machten. Doch auch der neue Trainingsansatz priorisiert keine Torschüsse. Die Parameter der Policy bevorzugen eine eigene Taktik nach Konvergenz des Trainings: Der Ball wird so nah wie möglich vor das Tor geschossen, ohne ein tatsächliches Tor zu erzielen – knapp 65% aller Episoden enden mit abgelaufener Zeit, der Rest damit, dass die Verteidiger den Ball berührt haben. Bei genauer Betrachtung geschieht letzteres in den meisten Fällen direkt vor dem Tor. Eine solcher Trajektorien ist in Abbildung 5.4 zu sehen.

Im Fazit führt auch diese Belohnungsfunktion nicht zum gewünschten Erfolg. Dennoch lässt sich eindeutig behaupten, dass die gesammelten Belohnungen das Explorationsverhalten der Agenten positiv beeinflusst, sodass sie verlässlich auf „das Schießen eines Tores“ stoßen.

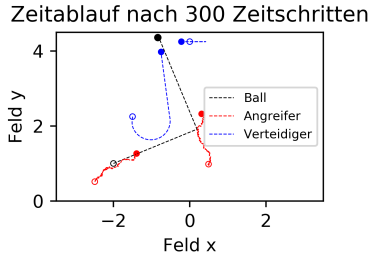


Abbildung 5.4: Eine Trajektorie, die ohne Torschuss endet, sondern durch Zeitablauf, kurz bevor der Verteidiger den Ball erreicht.

5.5 Normieren und Negieren

Die vorhergehenden Lösungsversuche basieren allesamt auf fundierten Ideen, laufen aber auf die gleichen Ergebnisse hinaus: Der Torschuss ist nicht im Fokus, sondern einzig und allein die Maximierung der Belohnungen. Mit weiterem arbiträren Anpassen der einzelnen eingeführten Boni der

Belohnungsfunktion ließe sich ein Zustand herstellen, der sich für das vorgestellte Szenario eignet, aber keinen tatsächlichen Mehrwert für andere bietet. Es muss also ein neuer Ansatz gefunden werden.

Das durch die bisher hergeleitete Belohnungsfunktion R angeregte Explorationsverhalten ist dennoch wünschenswert. Interessant wäre es, eine generelle Funktion $f(R)$ aufzustellen, welche die einzelnen Belohnungen so transformiert, dass folgende zwei Eigenschaften eingehalten werden:

i.) Jeweils zwei diskrete Belohnungen aus $f(R)$ sollen den gleichen relativen Abstand zueinander haben wie das analoge Belohnungspaar aus R. Mit dieser Eigenschaft reizt die Transformationsfunktion f das gleiche Explorationsverhalten wie die Belohnungsfunktion R an.

ii.) Trajektorien, die das definierte Ziels der Umgebung (in diesem Fall den Torschuss) schnell erreichen, müssen eine größere kumulative Belohnungssumme behalten als die, die mehr Zeitschritte benötigen.

Punkt ii.) ist dadurch erreichbar, dass der Wertebereich der Belohnungsfunktion komplett unter Null bleibt. Gibt es für jeden Zeitschritt eine konstant negative Belohnung, sind die kumulativen Belohnungssummen kürzerer Trajektorien grundsätzlich bevorzugt – der Agent wird versuchen, die Episoden schnellstmöglich zu beenden. Punkt i.) ist sichergestellt, solange sich die gefundene Transformation linear verhält.

Unter diesen Annahmen lässt sich f wie folgt aufstellen, wobei ϵ ein sehr kleiner Wert ist, der sicherstellt, dass die obere Grenze unter 0 liegt.

$$f(R) = R - R_{\max} - \epsilon \tag{5.9}$$

Die aufgestellte Transformationsfunktion führt auf einfachen, üblichen Experimenten im verstärkenden Lernen dazu, dass Agenten schnellstmöglich die Episoden beenden wollen.

Bevor die Funktion f aus Gleichung (5.9) im Roboterfußball eingesetzt wird, lässt sie sich noch optimieren. Unter der allgemeinen Annahme, dass PPO bessere Ergebnisse hervorbringt, wenn die Belohnungswerte sich um einen kleinen Bereich normiert aufhalten, kann sich einer Methodik aus der Statistik bedient werden.

Anstatt eine statische Funktion zu definieren, kann der sogenannte *gleitende Mittelwert* gebildet werden, inklusive der Standardabweichung. Das heißt, dass die „echten“ Belohnungen, die der Agent während des Trai-

nings verdient, gesammelt und verarbeitet werden: aus ihnen kann der aktuelle Mittelwert und die aktuelle Standardabweichung zum Zeitpunkt t berechnet werden, welche sich zusammen mit der Performanz des Agenten verändern.

Mithilfe des aktuellen Mittelwerts $\bar{x}$ und der Standardabweichung σ kann f folgendermaßen aufgestellt werden, mit ϵ als sehr kleinem Wert und c als fest definierte untere Grenzkonstante 10.

$$f_t(R) = \text{clip}\left(\frac{R - \left|R_{\bar{x}_t}\right|}{R_{\sigma_t} + \epsilon},\ -c,\ -\epsilon\right) \tag{5.10}$$

Zuerst wird die Verteilung von R um den momentanen Mittelwert $\bar{x}_t$ nach links verschoben, sodass die Werte sich nun größtenteils im negativen Bereich aufhalten. Die Division durch die ursprüngliche Standardabweichung staucht die Belohnungen so, dass ihre neue Standardabweichung 1 beträgt. Das ϵ im Dividenden verhindert numerische Instabilität.

Es ist noch nicht sichergestellt, dass die transformierten Belohnungen sich grundsätzlich im negativen Bereich aufhalten, da es starke Ausreißer in der ursprünglichen Belohnung geben könnte. Um dem entgegen zu wirken, kann sich Gleichung (2.39) bedient werden um eine minimale untere und eine maximale obere Grenze zu definieren. Als untere Grenze kann eine Konstante c benutzt werden, welche nicht zu klein sein sollte – vorerst ist 10 ein geeigneter Wert. Wichtiger ist die obere Grenze, welche auf knapp unter Null gesetzt wird. Durch diese Wertebeschneidung wird die Eigenschaft aus Punkt i.) gebrochen, die Ordnungsrelation bleibt aber innerhalb der Grenzen bestehen: für jedes Belohnungstupel r_1, r_2 gilt, dass $f(r_1) \geq f(r_2)$, wenn $r_1 \geq r_2$ – auch analog für $\leq$. Die Funktion f ist also innerhalb des getrimmten Bereiches monoton (d.h. für alle $\{x \mid -c \leq f(x) \leq -\epsilon\}$).

Dadurch, dass das Training auf vielen parallel simulierten Umgebungen abläuft, können $\bar{x}_t$ und σ_t gut berechnet werden, da zu jedem Zeitpunkt eine größere Menge an Datenpunkten vorliegt.

Welfords Algorithmus

Nach Welford (1962) kann die bekannte Varianz σ_n^2 einer Sequenz durch ein neues Element x_n unter Nutzung der Summe der quadratischen Abweichungen von dem Mittelwert $M_{2,n} = \sum_i^n (x_i - \bar{x}_n)^2$ berechnet werden.

$$\bar{x}_n = \frac{(n-1)\,\bar{x}_{n-1} + x_n}{n} = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \tag{5.11}$$

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})\,(x_n - \bar{x}_n) \tag{5.12}$$

$$\sigma_n^2 = \frac{M_{2,n}}{n} \tag{5.13}$$

Chan, Golub und LeVeque (1982) beschreiben eine parallelisierbare Version, welche in dieser Arbeit zum Einsatz kommt, da diese auf einer Menge von neuen Werten X arbeiten kann. Welfords Algorithmus sei als Spezialfall eines höheren Algorithmus' zu betrachten, welcher auf jeder Zerlegung der Menge X nach X_A und X_B arbeitet:

$$\delta = \bar{x}_B - \bar{x}_A \tag{5.14}$$

$$\bar{x}_X = \bar{x}_A + \delta \cdot \frac{n_B}{n_X} \tag{5.15}$$

$$M_{2,X} = M_{2,A} + M_{2,B} + \delta^2 \cdot \frac{n_A n_B}{n_X} \tag{5.16}$$

Die beschriebene Transformation f aus Gleichung (5.10) kann nicht einfach eingesetzt werden, da die Angreifer nun schnellstmöglich ein Episodenende herbeiführen werden. Im Falle des bisher aufgestellten Szenarios wird eine Trajektorie durch vier Geschehnisse beendet (Vgl. Abschnitt 5.1), wovon nur drei in der Entscheidungsgewalt der Angreifer liegen. Es ist aber schon per Definition schneller, den Ball aus den Feldgrenzen oder gegen einen Verteidiger zu schießen als ins Tor; daher muss das Szenario angepasst werden.

Durch die vorgehende Argumentation ist klar, dass eine Episode nur bei Erreichung des Ziels beendet werden darf – also, wenn ein Tor fällt. Das Zeitlimit kann dennoch weiterhin bestehen bleiben, solange es physikalisch möglich ist, vor Ablauf das Ziel der Umgebung zu erreichen.

Zusätzlich muss das Verhalten der Verteidiger weiter modelliert werden, da die Simulation nun nicht einfach beendet wird, wenn sie den Ball erreicht haben. Der Torwart wird so angepasst, dass er ihn erreichende Bälle wieder

ins Spielfeld hineintritt. Der mobile Verteidiger versucht weiterhin, den Ball schnellstmöglich zu erreichen – versucht diesen dann aber in Richtung des Angreifertores zu dribbeln. Andere Hyperparameter bleiben unberührt.

Auswertung

Das hier trainierte Modell legt erstmals den Fokus auf das Schießen eines Tores und stellt dadurch alle vorangehenden Netzwerke in den Schatten. Genau wie erwartet führen dauerhaft negative Belohnungen dazu, dass die Agenten nun das Erreichen eines Ziels im Hauptfokus haben. Die bisher erreichte Abschlussrate von einem Prozent wird um den Faktor 7 geschlagen – von 2048 simulierten Szenarios endeten 157 mit einem Torschuss der Angreifenden Roboter, was einer Rate von etwa 7,67% entspricht.

Abbildung 5.5 stellt eine Übersicht dar, nach wie vielen Zeitschritten die Tore üblicherweise erzielt wurden. Durchschnittlich endete eine Episode in der ein Tor geschossen wurde nach 272,68 Zeitschritten, wobei der Median mit 256 noch darunter liegt. Das lässt darauf schließen, dass es Ausreißer im höheren Spektrum gibt, welche den Durchschnitt beeinflussen.

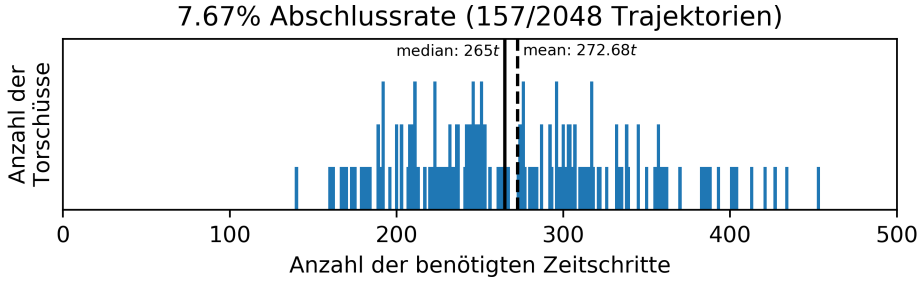


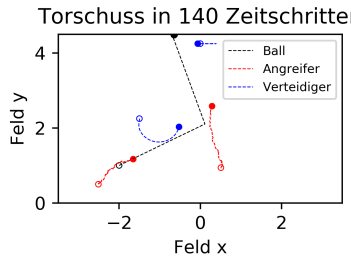
Abbildung 5.5: Eine Darstellung aller Torschüsse und zu welchem Zeitpunkt sie erzielt wurden. Es ist eine Tendenz zum frühen Torschuss abzulesen.

Das am frühesten geschossene Tor ist in Abbildung 5.6a zu sehen und zeigt eine perfekte Ausführung eines Torschusses über einen Pass in wenigen Zeitschritten (140). Interessanter zu betrachten sind die Trajektorien im Mittelfeld der Zeiten, bei denen der Ball auch von den verteidigenden Robotern berührt wird. Sowohl in Abbildung 5.6b als auch in Abbildung 5.6c machen sich die Angreifer die einfache Verhaltensweise der Verteidiger zunutze: zum einen wird der Ball fehlerhafterweise gegen den Torwart geschossen, welcher durch das einfache Zurückschießen des Balles einen

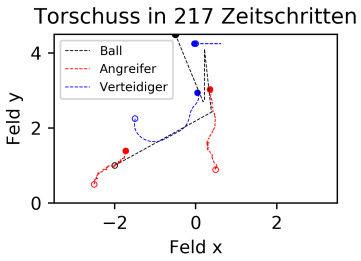
neuen Torschuss erlaubt und zum anderen lässt sich der mobile Verteidiger den Ball wieder abnehmen.

Die oben beschriebenen und vermuteten Ausreißer sind tatsächlich abzulesen. Abbildung 5.6 zeigt eine solche Trajektorie, bei der das Tor durch ein Versehen vom verteidigenden Roboter erzielt wird. Dies ist ein Fall, der nur der simplen Verhaltensweise der verteidigenden Roboter zuzuschreiben ist.

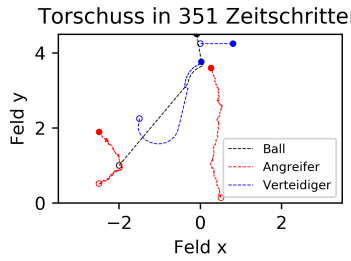
Alles in Allem ist der Einsatz der in Gleichung (5.10) eingeführten Belohnungstransformationsfunktion f ein Erfolg. Auch die Bewegungen der Angreifer sind in diesem Lösungsansatz weit zielstrebiger als die bisherigen vergleichsweise ruckartigen.



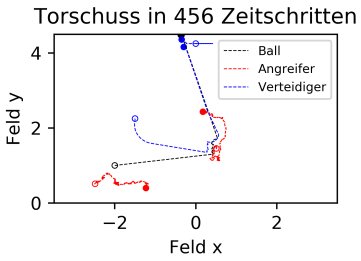
(a) Das schnellstmögliche Tor ist unspektakulär anzusehen.



(b) Im Mittelfeld der Zeiten kommt es häufig vor, dass der Ball versehentlich gegen den Torwart geschossen wird.



(c) Ein Torschuss im zeitlichen Mittelbereich. Hier ist gut zu sehen, wie der Angreifer das Verhalten des Verteidigers ausnutzt und ihm den Ball abnimmt, um selbst ein Tor zu schießen.



(d) Das langsamste Tor. Der stürmende Roboter steht mit dem Rücken zum Ball. Im Zweikampf landet der Ball dann kurz vor dem Tor, aber hinter dem Torwart. Der verteidigende Roboter erzielt daraufhin versehentlich ein Eigentor.

Abbildung 5.6: Beispieltrajektorien der in Abschnitt 5.5 trainierten Policy nach zwei Tagen.

5.6 Geteilte Schichten und Rekurrenz

Obgleich viele der gelernten Verhaltensweisen in den vergangen Experimenten sinnvoll aussehen, bleibt Optimierungsbedarf. Das größte Problem ist, dass sich die neuronalen Netzwerke in jedem Zeitschritt neu orientieren müssen. Die Policy π muss einzig und allein aufgrund des aktuellen Weltzustandes s_t eine Aktion a_t wählen. Interessant wäre es, wenn der Agent auch Zugriff auf zumindest Teile der Menge der früheren Weltzustände $\{s_0, s_1, \ldots, s_t\}$ hätte – dies würde viele Vorgehensweisen erlauben, die bisher unmöglich waren.

Eine gängige Methodik ist das sogenannte *FrameStacking*, bei dem der Beobachtungsraum n mal multipliziert und konkateniert wird – dem Agenten werden zu jedem Zeitschritt die letzten n Weltzustände $\{s_{t-n}, \ldots, s_t\}$ zur Verfügung gestellt. Dieser Ansatz birgt aber viele offensichtliche Probleme und leidet ganz besonders unter der schlechter Skalierbarkeit. Mit jedem einzelnen Zeitschritt, der weiter in die Vergangenheit geguckt werden soll, wird der komplette ursprüngliche Beobachtungsraum ein weiteres Mal benötigt. Infolgedessen muss die Komplexität der Netzwerkarchitektur ebenfalls wachsen. Zudem ist es unmöglich, den ganzen Zeitbereich $\{s_0, s_1, \ldots, s_t\}$ abzudecken, da der Beobachtungsraum dynamisch wachsen würde, eine neuronale Netzwerkarchitektur per Definition aber statische Größen erwartet.

Besser wäre es, wenn der Agent einen internen Zustand hätte, von dem er schreiben und lesen kann. So hätte er die Möglichkeit, sich nur relevante Informationen des kompletten Zeitstrahls zu merken und irrelevantes zu vergessen. Künstliche neuronale Netzwerke mit einem internen Zustand werden als *rekurrente* neuronale Netzwerke (RNNs) bezeichnet und haben genau diese Eigenschaft. Eine Erklärung dieser folgt auf Seite 71.

Zudem sind tiefe neuronale Netzwerke üblicherweise so zu verstehen, dass die ersten Schichten dafür zuständig sind, Merkmale aus den Eingängen zu extrahieren, auf Basis derer die tieferen Schichten Entscheidungen treffen können. Bisher wurden zwei neuronale Netzwerke nebeneinander trainiert; die Policy π und die *Value Function* V. Beide haben jeweils eigene Merkmalsextraktionsschichten verwaltet, was eine Redundanz darstellt.

Sinnvoll wäre es also, die ersten Schichten beider Netzwerke zu teilen, sodass die Merkmalsextraktionsschichten sowohl doppelt so schnell trainiert werden als sich auch konsistent über V und π verhalten. Diese Vorgehens-

weise ist eine Entscheidung, die auch bereits bei OpenAI u. a. (2019) zum Einsatz kam. Die Kombination beider Veränderungen führt zu folgender Netzwerkarchitektur in Abbildung 5.7.

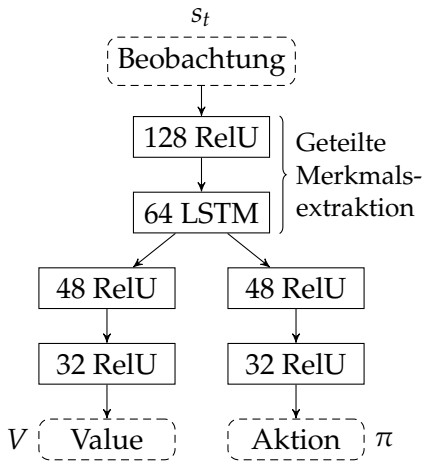
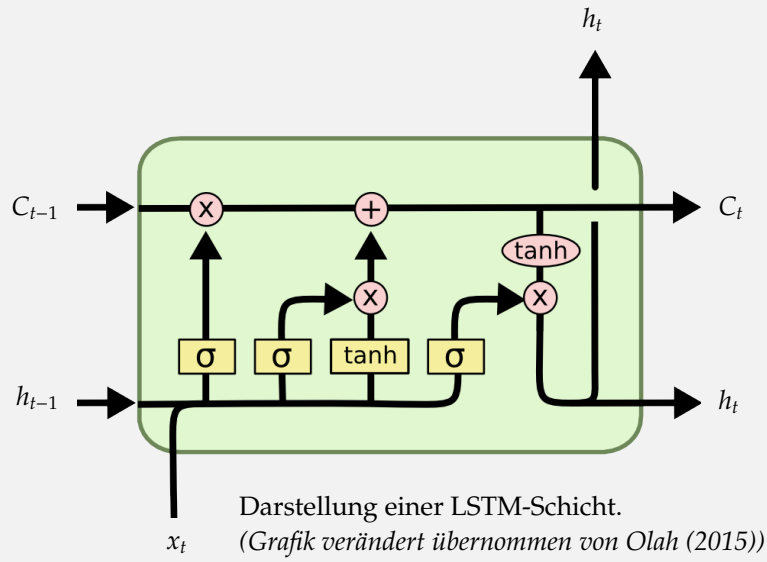


Abbildung 5.7: Die benutzte rekurrente Netzwerkarchitektur, bei der sich die Policy π und Value Function V die selbe Merkmalsextraktion teilen.

Rekurrente Künstliche Neuronale Netzwerke

Ein rekurrentes neuronales Netzwerk (RNN) zeichnet sich dadurch aus, dass Ausgänge einer Schicht auch mit Eingängen einer vorangehenden Schicht verbunden sein können. Die naivste RNN-Schicht, bei dem eine übliche Schicht ihren Ausgang an ihren Eingang koppelt, führt bereits zu sinnvollen Ergebnissen, kann aber noch stark verbessert werden.

Hochreiter und Jürgen Schmidhuber (1997) führen eine neue Art einer rekurrenten Schicht ein, welche aus mehreren Schichten und Operationen besteht. Diese Schicht wird Long-Short-Term Memory (LSTM) genannt, aufgrund ihrer Eigenschaft, Informationen über kurze oder lange Zeit zu speichern. Ähnlich wie beim naïven Ansatz eines RNNs ist der Ausgang h_t der Schicht mit dem Eingang der Schicht im nächsten Zeitschritt verbunden und wird auf den Ausgang der vorangehenden Schicht x_t konkateniert. Über insgesamt vier zusätzliche Schichten mit einfachen Neuronen und den Aktivierungsfunktionen $\sigma(x) = 1/(1 + e^{-x})$ und tanh wird von der *Erinnerungspipeline C* gelöscht (×) oder auf sie geschrieben (+). Die Erinnerung C beschreibt dabei den internen Zustand der Schicht. Olah (2015) beschreibt die Funktionsweise der LSTMs äußerst anschaulich.



Darstellung einer LSTM-Schicht.
(Grafik verändert übernommen von Olah (2015))

Rekurrente neuronale Netzwerke können nicht mehr einfach durch den Algorithmus zur Fehlerrückführung (Vgl. Abschnitt 2.2.1) trainiert werden, wohl aber durch eine Erweiterung namens *Backpropagation through Time*, welche über die Jahre von mehreren Quellen entwickelt wurde. Gut beschrieben wird er von Werbos (1990).

Auswertung

Nach einer mit Abschnitt 5.5 vergleichbaren Trainingszeit erlangt das hier trainierte rekurrente Netzwerk um den Faktor Zwei erhöhte Abschlussraten von 15,77% – abzulesen in Abbildung 5.8. Auch ist eine stärkere Tendenz zu früheren Torschüssen festzustellen, was eindeutig auf die erhöhte Zielstrebigkeit der Agenten zurückzuführen ist.

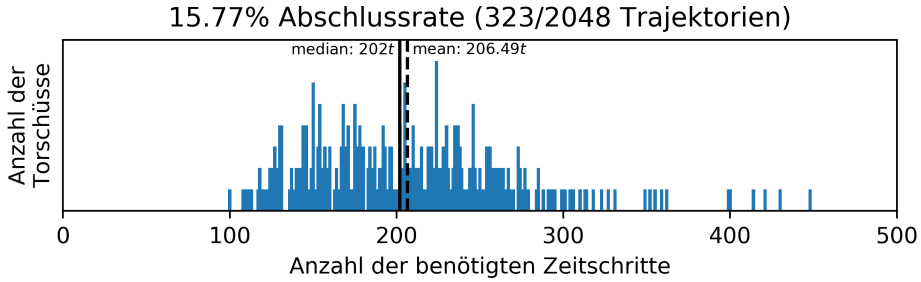
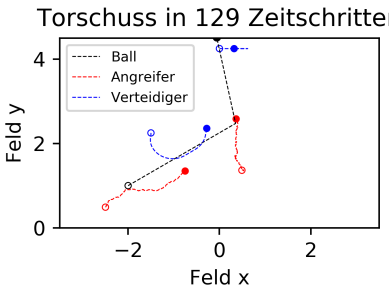


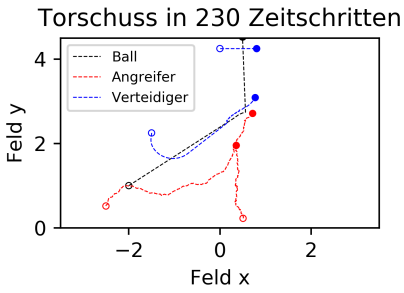
Abbildung 5.8: Beinahe 16% aller Episoden führen zu einem Torschuss.

Die verbesserte Zielstrebigkeit, wie sie in den Beispieltrajektorien in Abbildung 5.9 zu sehen ist, ist auf die LSTM-Schicht zurückzuführen. Durchschnittlich wird ein Tor in etwa 200 Zeitschritten erzielt, was insgesamt etwa 70 Zeitschritte früher ist als der gleiche Ansatz mit einem herkömmlichen neuronalen Netzwerk ohne rekurrente Verbindungen.

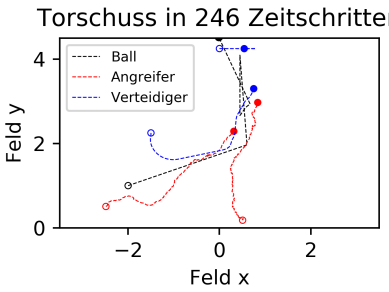
Auch dieses Experiment ist als Erfolg zu verbuchen. Die Verschmelzung von Policy- und *Value*-Netzwerk durch eine gemeinsame Merkmalsextraktionsschicht ist eindeutig eine erstrebenswerte Vorgehensweise und sollte beibehalten bleiben. Auch der Einsatz von rekurrenten Netzwerken hat maßgeblich zur Verbesserung der Ergebnisse beigetragen und sollte daher ebenfalls beibehalten werden.



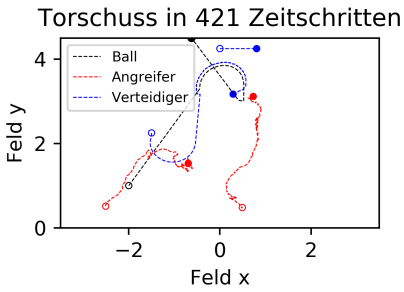
(a) Ein sehr schnelles Tor in beinahe perfekter Ausführung.



(b) Ein normales Tor, an dem gut zu sehen ist, wie zielstrebig sich die Angreifer bewegen.



(c) Erreicht der Torwart den Ball und schlägt ihn zurück, reagieren die Angreifer souverän und erzielen ein schnell folgendes Tor, ohne ihre Zielstrebigkeit aufzugeben.



(d) Auch dieser Ansatz kann den Verteidigern den Ball wieder abnehmen. Die Zielstrebigkeit der Angreifer fällt in dem Fall ein wenig zurück, könnte aber möglicherweise durch weiteres Training verbessert werden.

Abbildung 5.9: Eine Auswahl an Trajektorien des in Abschnitt 5.6 angelernten künstlichen neuronalen Netzwerk.

ABSCHLUSS

Nicht ganz unerwartet ist es natürlich möglich, Fußball mithilfe von (Deep) Reinforcement Learning zu spielen – und diese Arbeit legt sinnvolle Grundsteine dafür. Dieses Kapitel soll ein Fazit über die Herangehensweise und die daraus resultierenden Ergebnisse ziehen. Dabei soll besonders herausgearbeitet werden, welche Schritte nun folgend nötig sind, um in Zukunft ein komplettes Roboterfußballspiel zu simulieren und von neuronalen Netzen austragen zu lassen – in der Hoffnung, dass die Agenten das Fußballspiel als solches verstehen und möglicherweise sogar Verhaltensmuster entwickeln, die Menschen bisher nicht bedacht haben.

6.1 Fazit

Im direkten Fazit sind die Ergebnisse der Arbeit gut. Die entwickelte Simulationssoftware eignet sich dafür, schnell neue und einzigartige Szenarien zu definieren und mit diesen Versuchsreihen durchzuführen.

Besonders gelungen ist der in Abschnitt 5.5 herausgearbeitete Einsatz einer Belohnungstransformationsfunktion $f : \mathbb{R} \to \mathbb{R}^-$, welche das durch eine beliebige Belohnungsfunktion R angereizte Explorationsverhalten beibehält, diese aber so verändert, dass die Agenten das möglichst schnelle Erreichen eines Ziels in den Fokus legen. Diese Erkenntnis kann allgemein für jedes „zielbasierte“ Problem im verstärkenden Lernen eingesetzt werden und bedarf auch fernab des Roboterfußballs unbedingt weiterer Untersuchung.

Das gewählte Zwei-gegen-Zwei Szenario bietet einen guten Einstieg in die Thematik und wurde in starker Anlehnung an Ocana u. a. (2019) definiert. Dies geschah besonders, um direkte Vergleichswerte zu haben. Die Autoren kommen in ihrer Publikation zu der Konklusion, dass ihr Ansatz starker Verbesserung bedarf; ihr Training litt hauptsächlich unter lokalen Minima und daraus folgenden ungewollten Verhaltensweisen der Agenten (Dribbeln statt Passen, Passen statt Schießen, Schießen statt Dribbeln). Der

in dieser Arbeit entwickelte Ansatz leidet nicht darunter, vermutlich zurückzuführen auf die Wahl des Trainingsalgorithmus (DDPG vs. PPO) und der Explorationsmethodik (ϵ-*greedy* vs. stochastische *Policies*). Die Nutzung von stochastischen Policies führt zu sehr starkem Explorationsverhalten in Bereichen, in denen die Konfidenz des neuronalen Netzwerkes noch nicht sehr hoch ist. Proximal Policy Optimization (PPO) sorgt indirekt durch die benötigte höhere Anzahl parallel simulierter Umgebungen zu höheren Chancen zur Exploration und unterdrückt dann unter Nutzung der Advantages Aktionen, welche zu schlechter bewerteten *Trajektorien* führen als andere.

Die direkten Ergebnisse von Ocana u. a. (2019) lassen sich nicht mit den in dieser Arbeit erzielten vergleichen, da die Aktionsräume zu stark voneinander abweichen: die Agenten dieser Arbeit müssen in jedem Zeitschritt filigrane, konsistente Aktionen auswählen, während die Agenten von Ocana u. a. aus vordefinierten Aktionen („laufe dorthin“, „tritt vor Dich“, ...) wählen und bis zum Abschluss der Aktion nicht mehr eingreifen können. Mit dem höheren Ziel, die Agenten komplett allein Fußball spielen zu lassen, ist der in dieser Arbeit benutzte Aktionsraum die bessere Wahl – nur dann kann das neuronale Netzwerk die Kontrolle behalten.

6.2 Erkenntnisgewinn

Wie für Problemlösungen in der Informatik – und besonders in der künstlichen Intelligenz – üblich, sind Probleme häufig komplexer als ursprünglich antizipiert. Während der Entwicklungsphase bin ich dabei auf ein paar Probleme gestoßen und habe viele Ansätze ausprobiert, welche es nicht in dieses Dokument geschafft haben. Infolgedessen möchte ich an dieser Stelle meinen persönlichen Erkenntnisgewinn darlegen, um mögliche weitere Projekte des verstärkenden Lernens auf diese Stolpersteine hinzuweisen.

6.2.1 Entwicklungsaufwand der Umgebung

Zu Beginn sollte der Aufwand der Entwicklung einer Umgebung nicht unterschätzt werden. Gerade zwei- oder dreidimensionale Simulationen benötigen besonders viel Entwicklungszeit und sind anfällig für Programmierfehler, selbst wenn eine stabile und getestete Physiksimulationsbibliothek benutzt wird. Wenn es eine Möglichkeit gibt, die gestellte Simulation

auszutricksen, werden die Agenten den Weg finden und ausnutzen; wie auch schon Baker u. a. (2019) gezeigt haben, bei denen die Agenten sich durch fehlerhafte Kollisionserkennung aus dem Spielraum katapultiert haben um über eine unerklimmbare Mauer zu springen.

Möglicherweise ist es die bessere Wahl, sich auf eine bereits existierende und gut getestete Umgebung zu stützen, wie es OpenAI u. a. (2019) mit dem Videospiel Defense of the Ancients (DOTA 2) getan haben. Dies ist natürlich nicht immer möglich, da nicht jedes erdenkliche Szenario bereits mit einer offen verfügbaren und robusten Simulationssoftware belegt wurde; häufig muss die Umgebung von Grund auf implementiert werden.

In diesem Fall ist zu bedenken, dass PPO eine große Menge an Daten benötigt, die nur direkt aus der Umgebung gezogen werden können. Daher ist es durchaus nötig, eine akzeptable Menge an Zeit für die Entwicklung der Umgebung einzuplanen – je schneller die einzelnen Zustandsübergänge ablaufen, desto kürzer fallen die Trainingszeiten aus.

6.2.2 Komplexität der Problemstellung

Bei der Planung eines so umfangreichen Projektes wie dem Fußballspiel ist es sinnvoll, das Problem in eine Menge von kleineren und weniger komplexen Aufgabenstellungen aufzuteilen. Selbst OpenAI u. a. (2019) entwickelten vor dem vollen Fünf-gegen-Fünf-Spiel erst einen Eins-gegen-Eins-Prototypen.

Das korrekte Verhältnis von Simplizität und Komplexität ist dabei eine schwierige Gratwanderung und stellt bereits in sich eine Herausforderung dar. Es sollte sich auf keinen Fall von Anfang an auf ein viel zu schweres Problem gestürzt werden; besser ist es, das komplexe Problem mit dem gesammelten Erkenntnisgewinn aus vielen kleinen Experimenten zu lösen.

6.2.3 Netzwerkarchitekturen

Genau wie die Komplexität der Problemstellung sollte auch die der neuronalen Netzwerkarchitekturen initial nicht zu hoch ausfallen. Ist noch nicht sicher wie der Rest der kompletten Infrastruktur aufgestellt wird, sind äußerst tiefe Architekturen hinderlich, da die Ausführungszeiten proportional zur Anzahl der Neuronenverbindungen im Netzwerk wachsen. Zudem

hat die erst kürzlich erschienene Publikation von Nakkiran u. a. (2019) gezeigt, dass die Performanz von künstlichen neuronalen Netzwerken unter bestimmten Bedingungen an einer Komplexitätssteigerung des Modells leiden kann.

6.2.4 Initialer Fokus auf Belohnungsfunktionen

Nach den Erfahrungen dieser Arbeit spielt die Belohnungsfunktion im verstärkenden Lernen die tragendste Rolle; einzig und allein die Belohnung bestimmt das Verhalten der Agenten. Daher sollte besonders früh für jedes Experiment eine möglichst simple und elegante Belohnungsfunktion aufgestellt werden, bevor zu viel Arbeitszeit in die Entwicklung von Beobachtungsräumen gesteckt wird.

6.3 Ausblick

Es folgt ein Ausblick, wie es im direkten Anschluss an diese Arbeit weitergehen sollte, um das Ziel zu erreichen, komplett autonom Fußball mit neuronalen Netzwerken zu spielen.

6.3.1 Zwei-gegen-Zwei

Im direkten Anschluss an diese Arbeit sollte das Zwei-gegen-Zwei Szenario weitergeführt werden. Zunächst sollte dieses generalisiert werden: Angreifer und Verteidiger müssen auf zufälligen Positionen auf dem Feld starten, sodass klargestellt wird, dass die neuronalen Netzwerke tatsächlich das Konzept des Fußballspielens verstehen.

Weiterhin haben die erreichten Torschussraten noch viel Luft nach oben. Die grundsätzliche Wahl der Algorithmen, ihrer Stellschrauben und der Belohnungsfunktion sind bereits ziemlich optimal gewählt – die größte Leistungssteigerung lässt sich vermutlich durch einen besseren Beobachtungsraum und eine kompliziertere Netzwerkarchitektur erreichen, welche die Tatsache ausnutzt, dass künstliche neuronale Netzwerke jede azyklische Graphstruktur annehmen können.

Sowohl OpenAI u. a. (2019) als auch Baker u. a. (2019) nutzen in ihren Netzwerkarchitekturen Expertenwissen über den Beobachtungsraum aus.

Verschiedene Teile der Beobachtungsfunktion werden von verschiedenen Schichten verarbeitet, die durch ihre Eigenschaften besonders geeignet für die Art der eingehenden Informationen sind.

Zwei mögliche Zusätze für die Beobachtungsräume sind die Folgenden:

Zunächst wäre es möglich, die Feldkonfiguration als 2D-Matrix (als „Bild“) darzustellen und darauf eine Merkmalsextraktion durch sogenannte Faltungsschichten (engl. *Convolutional Layer*) durchzuführen. Faltungsschichten können besonders effektiv mit mehrdimensionalen Daten umgehen. Dies würde den Agenten einen direkten Überblick über die räumlichen Verhältnisse zwischen einzelnen Robotern geben.

Baker u. a. (2019) statten zudem jeden Agenten mit einer größeren Anzahl an rundherum befestigten Distanzmessern aus und lassen diese durch eine kreisförmige Faltungsschicht (engl. *Circular 1D Convolution*) verarbeiten: Eine Vorgehensweise, die den Agenten erlaubt, ihre direkte Umgebung im Nahbereich besser einschätzen zu können. Kreisförmige Faltungsschichten können besonders gut mit „Wrap-Around“-Daten umgehen, wie beispielsweise 360°-Fotos – oder die hier vorliegenden „LIDAR“-Daten. (Schubert u. a., 2019)

Als Verifikation der Effektivität des Beobachtungs- und Aktionsraumes kann die Versuchsreihe aus Kapitel 5 noch einmal mit dem inversen Szenario getestet werden: Die Angreifer müssen eine fest programmierte Vorgehensweise zugewiesen bekommen und die neuronalen Netzwerke übernehmen die Rolle der Verteidiger.

6.3.2 Generelles Fußballspiel

Mit dem Erkenntnisgewinn aus dem vorangegangenen Szenario ließe sich nun ein volles Fünf-gegen-Fünf Fußballspiel (wie es im Roboterfußball gängig ist) lernen. Das größte Problem wird dort vermutlich wieder die Belohnungsfunktion sein.

Viele moderne Publikationen im Bereich des verstärkenden Lernens kommen zu der Konklusion, dass die handgeformten Belohnungsfunktionen als Konzept aufgegeben und durch bessere Ansätze ausgetauscht sollten – obgleich der Erfolg von OpenAI u. a. (2019) zeigt, dass sie mit der richtigen Nachverarbeitung zu äußerst guten Ergebnissen führen können.

Baker u. a. (2019) zeigen die Effektivität von extrem simplen Belohnungs-

funktionen in Kombination mit kompetitivem *Self-Play* – ein wie für das Fußballspiel geschaffener Ansatz. Spielen zwei verschiedene neuronale Netze gegeneinander, müssen regelmäßig neue Taktiken entwickelt werden, um mit den Fortschritten des anderen mithalten zu können. Vermutlich wird dennoch eine normale Belohnungsfunktion benötigt, die am Anfang des Trainings dafür sorgt, dass die Roboter verstehen, dass sie den Ball in Richtung des gegnerischen Tores bewegen müssen. Diese theoretische Funktion müsste im Laufe des Trainings in eine simple binäre Funktion überblenden, welche nur Punkte für erzielte Tore und Punktabzug für kassierte gibt.

Eine Idee einer ebensolchen Funktion verfolgt mich schon sehr lange; sie kam leider nie zum Einsatz, daher wird sie hier einmal im Groben beschrieben. Die Grundidee ist es, das Fußballspiel als Wegfindungsproblem des Balles zu betrachten. Laue (2004) nutzt im Roboterfußball sogenannte Potentialfelder, die jedem Punkt im Raum eine Richtung und eine Stärke zuordnen, die einen sicheren Weg in Richtung eines Ziels beschreiben und dabei Hindernisse vermeiden (Vgl. Abbildung 6.1).

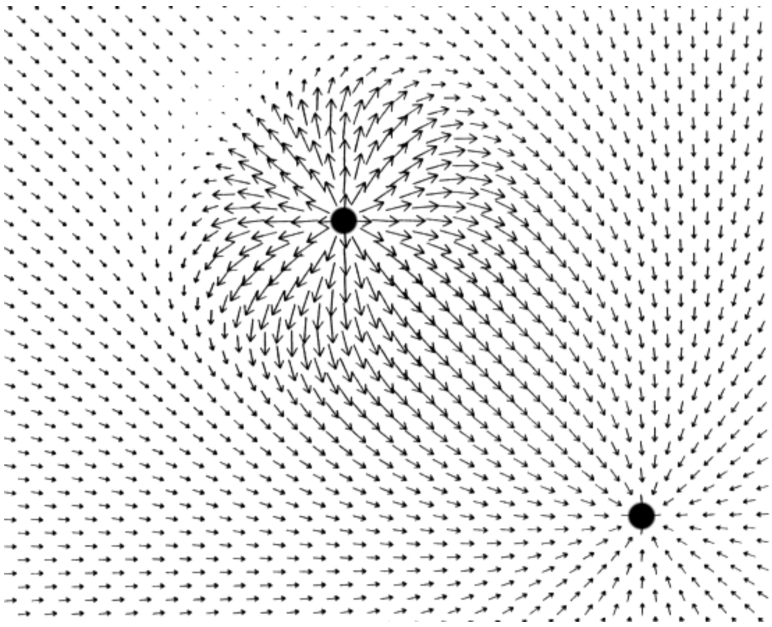


Abbildung 6.1: Ein Potentialfeld mit einem Zielpunkt (rechts unten) und einem Hindernis, welches vermieden werden soll (mittig). Die Grafik wurde aus Laue (2004) übernommen, welcher die Grafik, leicht verändert, aus Arkin (1998) übernommen hat.

In der Arbeit werden Potentialfelder genutzt, um einen möglichst sicheren Pfad eines Punktes zu einem anderen Punkt im Raum zu berechnen und dabei Hindernissen auszuweichen. Mit dem gegnerischen Tor als Zielpunkt und gegnerischen Robotern sowie des eigenen Tores als zu vermeidende Hindernisse würde sich für jeden Punkt im Raum eine generelle Richtung ausrechnen lassen können, wohin sich der Ball optimalerweise bewegen müsste – die Abweichung dessen wäre ein mögliches Belohnungsmaß.

Dadurch, dass sich das Potentialfeld in jedem Zeitschritt mit den Feldkonfigurationen verändert, würden sich die Belohnungswerte auch bei einem stillliegenden Ball verändern. Dennoch wird nicht explizit angereizt, dass sich das Team auf den Ball hinbewegt. In diese Richtung kann und sollte weitergedacht werden – interessante Ansatzpunkte dazu wären die Arbeiten von Burda u. a. (2018), Bansal u. a. (2017) oder Ho und Ermon (2016), um einige aufzuzählen.

Schlussendlich liegt es in nicht allzu ferner Zukunft, tatsächlich ein vollständiges Roboterfußballspiel zu simulieren und die Agenten sinnvolle Taktiken finden zu lassen, es würde nur den möglichen Arbeitsaufwand dieser Masterarbeit überschreiten. Diese Arbeit ist ein voller Erfolg und legt alle dafür nötigen Grundsteine.

LITERATUR

Arkin, Ronald C. (1998). *Behavior-Based Robotics*. MIT Press.

Baker, Bowen u. a. (Sep. 2019). *Emergent Tool Use From Multi-Agent Autocurricula*. Online verfügbar. arXiv: 1909.07528 [cs.LG]. URL: https://arxiv.org/abs/1909.07528.

Bansal, Trapit u. a. (Okt. 2017). *Emergent Complexity via Multi-Agent Competition*. Online verfügbar. arXiv: 1710.03748 [cs.AI]. URL: https://arxiv.org/abs/1710.03748.

Burda, Yuri u. a. (Aug. 2018). *Large-Scale Study of Curiosity-Driven Learning*. Online verfügbar. arXiv: 1808.04355 [cs.AI]. URL: https://arxiv.org/abs/1808.04355.

Campbell, Murray, A. Joseph Hoane Jr. und Feng-hsiung Hsu (Jan. 2002). „Deep Blue“. In: *Artif. Intell.* 134.1-2, S. 57–83.

Chan, T. F., G. H. Golub und R. J. LeVeque (1982). „Updating Formulae and a Pairwise Algorithm for Computing Sample Variances“. In: *COMPSTAT 1982 5th Symposium held at Toulouse 1982*, S. 30–41.

Hahnloser, Richard H. R. u. a. (2000). „Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit“. In: *Nature* 405, S. 947–951.

Hebb, Donald O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press.

Ho, Jonathan und Stefano Ermon (Juni 2016). *Generative Adversarial Imitation Learning*. Online verfügbar. arXiv: 1606.03476 [cs.LG]. URL: https://arxiv.org/abs/1606.03476.

Hochreiter, Sepp und Jürgen Schmidhuber (Dez. 1997). „Long Short-term Memory“. In: *Neural computation* 9, S. 1735–80.

Howard, Ronald A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Kalyanakrishnan, Shivaram und Peter Stone (2010). „Learning Complementary Multiagent Behaviors: A Case Study“. In: *RoboCup 2009: Robot Soccer World Cup XIII*, S. 153–165.

Kingma, Diederik P. und Jimmy Ba (Dez. 2014). „Adam: A Method for Stochastic Optimization“. In: *arXiv e-prints*. Online verfügbar. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

Laue, Tim (Jan. 2004). „Eine Verhaltenssteuerung für autonome mobile Roboter auf der Basis von Potentialfeldern“. Diplomarbeit. Universität Bremen.

McCulloch, Warren S. und Walter Pitts (Dez. 1943). „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4, S. 115–133.

Nakkiran, Preetum u. a. (Dez. 2019). *Deep Double Descent: Where Bigger Models and More Data Hurt*. Online verfügbar. arXiv: 1912.02292 [cs.LG]. URL: https://arxiv.org/abs/1912.02292.

Ocana, Jim Martin Catacora u. a. (2019). „Cooperative Multi-Agent Deep Reinforcement Learning in a 2 Versus 2 Free-Kick Task“. In: *RoboCup 2019: Robot World Cup XXIII*, S. 44–57.

OpenAI u. a. (Dez. 2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. Online verfügbar. arXiv: 1912.06680 [cs.LG]. URL: https://arxiv.org/abs/1912.06680.

Rosenblatt, F. (1958). „The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain“. In: *Psychological Review*, S. 65–386.

Rumelhart, David E., Geoffrey E. Hinton und Ronald J. Williams (1988). „Neurocomputing: Foundations of Research“. In: *Nature* 323, S. 696–699.

Schmidhuber, Jürgen (Apr. 2014). *Deep Learning in Neural Networks: An Overview*. Online verfügbar. arXiv: 1404.7828 [cs.NE]. URL: https://arxiv.org/abs/1404.7828.

Schubert, Stefan u. a. (Juni 2019). „Circular Convolutional Neural Networks for Panoramic Images and Laser Data“. In: *2019 IEEE Intelligent Vehicles Symposium (IV)*, S. 653–660.

Schulman, John, Sergey Levine u. a. (Feb. 2015). *Trust Region Policy Optimization*. Online verfügbar. arXiv: 1502.05477 [cs.LG]. URL: https://arxiv.org/abs/1502.05477.

Schulman, John, Philipp Moritz u. a. (Juni 2015). *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. Online verfügbar. arXiv: 1506.02438 [cs.LG]. URL: https://arxiv.org/abs/1506.02438.

Schulman, John, Filip Wolski u. a. (Juli 2017). *Proximal Policy Optimization Algorithms*. Online verfügbar. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

Silver, David, Aja Huang u. a. (2016). „Mastering the game of Go with deep neural networks and tree search“. In: *Nature* 529, S. 484–503.

Silver, David, Thomas Hubert u. a. (Dez. 2017). *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Online verfügbar. arXiv: 1712.01815 [cs.AI]. URL: https://arxiv.org/abs/1712.01815.

Sutton, Richard S. u. a. (2000). „Policy Gradient Methods for Reinforcement Learning with Function Approximation“. In: *Advances in Neural Information Processing Systems 12*, S. 1057–1063.

Welford, B. P. (1962). „Note on a Method for Calculating Corrected Sums of Squares and Products“. In: *Technometrics* 4.3, S. 419–420.

Werbos, Paul John (Nov. 1990). „Backpropagation through time: what it does and how to do it“. In: *Proceedings of the IEEE* 78, S. 1550–1560.

Werbos, Paul John (1994). *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley-Interscience.

INTERNETQUELLEN

Achiam, Joshua (Juli 2018). *Simplified PPO-Clip Objective*. Mathematischer Beweis, verwendet von OpenAI. Nur online. URL: https://drive.google.com/file/d/1PDzn9RPvaXjJFZkGeapMHbHGiWWW20Ey/ (besucht am 31. 12. 2019).

Catto, Erin (2019). *Box2D – A 2D Physics Engine for Games*. Internetauftritt. Nur online. URL: http://box2d.org/ (besucht am 31. 12. 2019).

Hill, Ashley u. a. (2019). *Stable Baselines*. Softwarebibliothek. Nur online. URL: https://github.com/hill-a/stable-baselines (besucht am 31. 12. 2019).

Kolecki, Joseph C. (2002). *An Introduction to Tensors for Students of Physics and Engineering*. Lehrmaterial der National Aeronautics and Space Administration (NASA). Nur online. URL: https://www.grc.nasa.gov/www/k-12/Numbers/Math/documents/Tensors_TM2002211716.pdf (besucht am 31. 12. 2019).

Kurach, Karol und Oliver Bachem (2019). *Introducing Google Research Football: A Novel Reinforcement Learning Environment*. Blogeintrag. Nur online. URL: https://ai.googleblog.com/2019/06/introducing-google-research-football.html (besucht am 31. 12. 2019).

Olah, Cristopher (Aug. 2015). *Understanding LSTM Networks*. Blogeintrag. Nur online. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (besucht am 31. 12. 2019).

OpenAI (2019a). *Gym*. Internetauftritt. Nur online. URL: https://gym.openai.com (besucht am 31. 12. 2019).

OpenAI (2019b). *Spinning Up in Deep RL*. Internetauftritt. Nur online. URL: https://spinningup.openai.com/en/latest/index.html (besucht am 31. 12. 2019).

RoboCup Technical Committee (Mai 2019). *RoboCup Standard Platform League (NAO) Rule Book*. Regelbuch. Nur online. URL: https://spl.robocup.org/wp-content/uploads/downloads/Rules2019.pdf (besucht am 05. 01. 2020).

Röfer, Thomas u. a. (2019). *B-Human Team Report and Code Release 2019*. Nur online. URL: http://www.b-human.de/downloads/publications/2019/CodeRelease2019.pdf (besucht am 31. 12. 2019).

Stanford University u. a. (2015). *Deep Learning Tutorial*. Internetauftritt. Nur online. URL: http://ufldl.stanford.edu/tutorial/ (besucht am 31. 12. 2019).

The Learning Agents Research Group (2019). *RoboCup 2D Half Field Offense*. Software. Nur online. URL: https://github.com/LARG/HFO (besucht am 31. 12. 2019).

Valve Corporation (2019). *Dota 2*. Internetauftritt. Nur online. URL: http://blog.dota2.com/ (besucht am 31. 12. 2019).

Vinyals, Oriol u. a. (Jan. 2019). *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. URL: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (besucht am 31. 12. 2019).