

Greifen von Objekten mit einem 7-DoF-Roboterarm mithilfe von Deep Reinforcement Learning

Bachelorarbeit

von

Vincent Müller

Studiengang: Wirtschaftsingenieurwesen

Matrikelnummer: 2211843

Institut für Angewandte Informatik und Formale
Beschreibungsverfahren (AIFB)
KIT-Fakultät für Wirtschaftswissenschaften

Prüfer:	Prof. Dr. Andreas Oberweis
Zweiter Prüfer:	Prof. Dr.-Ing. Johann Marius Zöllner
Betreuer:	M. Sc. Demian Frister
Eingereicht am:	30. Juni 2022

Zusammenfassung

Greifen ermöglicht Robotern praktische Aufgaben von Menschen zu übernehmen und erlaubt so eine schnellere Konfiguration von Produktionslinien und einen Einsatz von Robotern in Wohnumgebungen. Dabei ist Greifen eine immer noch herausfordernde Aufgabe für einen Roboter. Besonders das Greifen von unbekannten Objekten und in dynamischen Umgebungen bereitet einem Roboter Schwierigkeiten. Zur Lösung dieser Aufgabe gibt es verschiedene Ansätze, die Deep Reinforcement Learning-Methoden anwenden. Meistens werden dabei RGB-Kameradaten verwendet, um ein beliebiges Objekt aus einer Kiste mit mehreren Objekten zu greifen.

In dieser Arbeit wird ein Software-System entwickelt und implementiert, das einem Roboterarm mit sieben Freiheitsgraden und einer am Endeffektor montierten RGBD-Kamera ermöglicht, zu lernen ein bestimmtes Objekt auf einer Tischplatte zu greifen. Dafür wird ein Modell mithilfe des Deep Reinforcement Learning Algorithmus Deep Deterministic Policy Gradient trainiert.

Für das Training wird eine Simulationsumgebung implementiert. Beim Training konnte beobachtet werden, dass der Deep Deterministic Policy Gradient-Algorithmus häufig in eine Verklemmung gerät, die weiteres Lernen verhindert, und dass das Training instabil ist. Des Weiteren benötigt es eine große Anzahl an Trainingsepisoden, sodass das Modell anhand von Tiefenbildern lernt. Die beim Training generierten Daten bilden einen Datensatz, der einfach auf weitere Reinforcement Learning Algorithmen angewendet werden kann.

Zur Evaluation des Modells wird eine Schnittstelle zur realen Hardware implementiert. Bei der Evaluation ist außerdem festzustellen, dass der Unterschied zwischen Simulations- und Realumgebung eine verschlechterte Leistung des Modells verursacht. Hier ist ein Simulations-zu-Realitäts-Transfer-Ansatz nötig.

Inhaltsverzeichnis

Inhaltsverzeichnis	II
Abbildungsverzeichnis	IV
Formelverzeichnis	V
1. Einleitung und Motivation.....	7
2. Grundlagen	8
2.1. Deep Learning	8
2.1.1. Künstliche Neuronen	8
2.1.2. Tiefe Neuronale Netze.....	10
2.1.3. Backpropagation.....	12
2.2. Deep Reinforcement Learning	16
2.2.1. Umgebung des Agenten	17
2.2.2. Strategie des Agenten	20
2.2.3. Erwarteter zukünftiger Ertrag.....	21
2.2.4. Policy Optimisation	22
2.2.5. Q-Learning	23
2.2.6. Deep Deterministic Policy Gradient.....	24
2.3. Robot Operating System	26
2.4. Franka Emika Robotersystem.....	28
2.5. Gazebo Simulationssoftware	29
2.6. MoveIt Bewegungsplanungssoftware	30
2.7. Intel RealSense D435 Tiefenkamera	31
3. Forschungsstand.....	31
4. Konzept	32
4.1. Umgebung	32
4.1.1. Markov-Entscheidungsprozess.....	32
4.1.2. Simulations- und Realumgebung	34
4.2. Agent	34
4.2.1. Actor-Netzwerk	34
4.2.2. Critic-Netzwerk	35
4.2.3. Replay Buffer	35
4.2.4. Target Netzwerke	35
5. Implementierung	37
5.1. Simulationsumgebung	37
5.1.1. Konfiguration der Simulationsumgebung	37
5.1.2. Funktionen der Simulationsumgebung.....	38

5.2.	Agent	40
5.2.1.	Konfiguration des Replay Buffers	40
5.2.2.	Konfiguration des Modells	41
5.2.3.	Konfiguration des Trainings	41
5.3.	Realumgebung	43
5.3.1.	Konfiguration der Realumgebung	43
5.3.2.	Funktionen der Realumgebung	44
6.	Evaluation	45
6.1.	Ergebnis	45
6.2.	Einschränkungen	49
7.	Fazit	50
7.1.	Zusammenfassung	50
7.2.	Ausblick	50
	Literaturverzeichnis	52
	Erklärung	54

Abbildungsverzeichnis

Abbildung 1: Beispiel für Approximation einer Funktion mit Deep Learning	8
Abbildung 2: Schematische Darstellung einer Nervenzelle	8
Abbildung 3: Darstellung eines künstlichen Neurons mit seinen Elementen.....	9
Abbildung 4: Struktur eines Tiefen Neuronalen Netzes.....	10
Abbildung 5: Input und Output eines KNNs zur Klassifizierung von handgeschriebenen Ziffern.....	11
Abbildung 6: Beispiel für gelabelte Trainingsdaten.....	12
Abbildung 7: Darstellung des Mountain-Car-Problems	17
Abbildung 8: Interaktionskreislauf des Agenten	19
Abbildung 9: ROS-Kommunikationsstruktur mit ROS-Services und ROS-Topics	27
Abbildung 10: Franka Emika Robotersystem mit Hand	28
Abbildung 11: Gazebo-Benutzeroberfläche mit Beispiel-Objekten.....	29
Abbildung 12: RViz-Benutzeroberfläche mit dargestellter Trajektorie	30
Abbildung 13: Intel RealSense D435 Tiefenkamera.....	31
Abbildung 14: Konzeptueller Überblick über das System	32
Abbildung 15:Überblick über die ROS-Nodes des Systems	37
Abbildung 16: Überblick über die Simulationsumgebung	37
Abbildung 17: Simulationsumgebung mit Roboter, Kamera, Würfel und Tischplatte	38
Abbildung 18: Überblick über die Nodes des Agenten	40
Abbildung 19: Überblick über die Realumgebung.....	43
Abbildung 20: Durchschnittlicher Ertrag der letzten 50 Episoden im Trainingsverlauf.....	46
Abbildung 21: Relativer Greiferfolg der letzten 50 Episoden im Trainingsverlauf.....	46
Abbildung 22: Unterschied der Tiefenbilder zwischen Simulations-(links) und Realumgebung(rechts)	48

Formelverzeichnis

Formel 1: Sigmoid-Funktion	9
Formel 2: Ableitung der Sigmoid-Funktion	9
Formel 3: Output eines künstlichen Neurons	10
Formel 4: Trainingsdatenpaar	12
Formel 5: Output eines KNN	12
Formel 6: Fehler eines Trainingspaares	13
Formel 7: Fehler des KNNs.....	13
Formel 8: Richtung des steilsten Abstiegs des Fehlers	13
Formel 9: Gradient des Fehlers als Durchschnitt der Fehler eines Trainingspaares	13
Formel 10: Gradient des Fehlers eines Trainingspaares.....	14
Formel 11: Anpassungsregel der Gewichte.....	14
Formel 12: Partielle Ableitung des Fehlers eines Trainingspaares	14
Formel 13: Partielle Ableitung des Fehlers mit Kettenregel	14
Formel 14: Ableitungen der Bestandteile der partiellen Ableitung des Fehlers.....	15
Formel 15: Ergebnis der partiellen Ableitung des Fehlers	15
Formel 16: Partielle Ableitung des Fehlers nach Gewichten der vorletzten Ebene	15
Formel 17: Ableitung des Fehlers nach Output der vorletzten Ebene.....	15
Formel 18: Ergebnis der partiellen Ableitung des Fehlers nach Gewichten in der vorletzten Ebene ...	15
Formel 19: Partielle Ableitung des Fehlers nach den Gewichten einer beliebigen Ebene	16
Formel 20: Markov-Entscheidungsprozess	17
Formel 21: Zustandsraum des Mountain-Car-Problems	18
Formel 22: Aktionsraum des Mountain-Car-Problems	18
Formel 23: Belohnungsfunktion des Mountain-Car-Problems	19
Formel 24: Übergangswahrscheinlichkeitsfunktion.....	19
Formel 25: Verteilung des Startzustände des Mountain-Car-Problems	19
Formel 26: Episode	20
Formel 27: Strategie des Agenten	20
Formel 28: Optimale Strategie	20
Formel 29: On-Policy-Wertfunktion	21
Formel 30: On-Policy-Aktions-Wertfunktion	21
Formel 31: Optimale Wertfunktion	21
Formel 32: Optimale Aktions-Wertfunktion	22
Formel 33: Lernregel des Gradientenaufstiegsverfahren	22
Formel 34: Gradient des Erwarteten Ertrags	22
Formel 35: Schätzung des Policy Gradient	22
Formel 36: Optimale Strategie bei Q-Learning.....	23

Formel 37: Vorhersagefehler Q-Learning.....	24
Formel 38: Approximation des maximalen erwarteten zukünftigen Ertrags.....	25
Formel 39: Anpassung der Gewichte der Target-Netzwerke	25
Formel 40: Vorhersagefehler des Critics.....	25
Formel 41: Optimierungsproblem des Critics	26
Formel 42: Ertragsfunktion des Actors	26
Formel 43: Optimierungsproblem des Actors	26
Formel 44: Zustandsraum der Greif-Problems.....	33
Formel 45: Aktionsraum des Greif-Problems	33
Formel 46: Belohnungsfunktion des Greif-Problems.....	33

1. Einleitung und Motivation

Greifen ist eine elementare Aufgabe in der Robotik, da es eine physische Interaktion eines Roboters mit seiner Umgebung ermöglicht. Dabei ist Greifen eine immer noch herausfordernde Aufgabe für einen Roboter (Hodson, 2018). Besonders das Greifen von unbekannten Objekten und in dynamischen Umgebungen bereitet einem Roboter Schwierigkeiten (Kleeberger et al., 2020).

Durch steigende Datenverfügbarkeit und Rechenleistung gibt es einen signifikanten Fortschritt bei auf Maschinellem Lernen basierten Ansätzen zum Robotergreifen (Kleeberger et al., 2020). Diese Ansätze können generalisieren und auch neue, unbekannte Objekte greifen. Das erlaubt eine effizientere Inbetriebnahme von Roboteranwendungen (El-Shamouty et al., 2019), was in flexiblen Produktionssystemen mit kurzen Produktlebenszyklen und hoher Produktdiversität immer wichtiger wird (Reinhart et al., 2011).

Die Anwendung von Reinforcement Learning bietet Vorteile beim Greifen. Das Verwenden von Rohsensordaten ermöglicht ein komplexeres Greifverhalten, da auch Manipulationen vor dem Greifen beim Training erlernt werden können (Kleeberger et al., 2020). Die Möglichkeit das Modell in einer Simulation zu trainieren erlaubt eine große Anzahl an korrekt annotierten Trainingsdaten mit weniger Aufwand, ohne Verschleiß am Roboter und ohne Sicherheitsrisiko zu erstellen (Kleeberger et al., 2020) und macht so den effizienteren Einsatz von Robotern möglich (El-Shamouty et al., 2019). Da die Performance eines in einer Simulation trainierten Modells nicht der Realität entspricht (Weibel et al., 2019), sollte ein Ansatz zum Transfer der Simulation in die Realität angewandt werden.

Durch die Kombination von Reinforcement Learning mit Deep Learning erlernen Roboter bei auf Deep Reinforcement Learning basierten Ansätzen verschiedenste komplexe Manipulationsstrategien mit einem hohen Greiferfolg (Kalashnikov et al., 2018). Außerdem ist hier eine Steuerung mit geschlossenem Regelkreis möglich, die es erlaubt dynamisch auf Störungen zu reagieren (Kalashnikov et al., 2018).

2. Grundlagen

2.1. Deep Learning

Deep Learning (DL) ist eine Form von Maschinellem Lernen. Dabei werden tiefe Netze aus Künstlichen Neuronen genutzt, um Funktionen zu approximieren. Die Approximation erfolgt in einem Lernprozess mithilfe des sogenannten Backpropagation-Algorithmus. Mit DL können anspruchsvolle Aufgaben unter anderem in Bildverarbeitung (Shi et al., 2017) oder Spracherkennung (Amirsina Torfi, 2020) mit geringer Fehlerquote ausgeführt werden. So kann man zum Beispiel eine Funktion approximieren, die digitale Bilder von per Hand geschriebenen Ziffern auf die von ihnen dargestellten Ziffern abbilden (Lecun et al., 1998) (Abbildung 1).

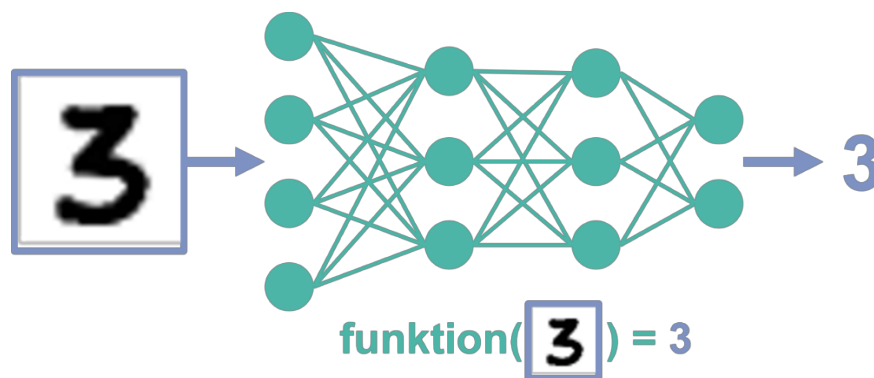


Abbildung 1: Beispiel für Approximation einer Funktion mit Deep Learning

2.1.1. Künstliche Neuronen

Künstliche Neuronen sind menschlichen Neuronen nachempfunden (Haykin, 1994). Menschliche Neuronen sind Nervenzellen, die auf Übertragung von Signalen spezialisiert sind. Ein menschliches Neuron besteht dazu aus einem Zellkern und zwei Arten von Zellfortsätzen: den Dendriten und dem

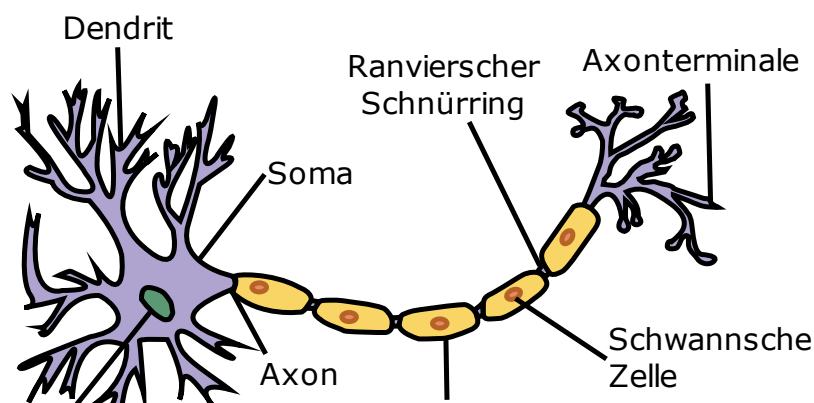


Abbildung 2: Schematische Darstellung einer Nervenzelle

Axon (Abbildung 2¹). Über die Dendriten werden Signale von anderen Zellen aufgenommen. Im Zellkern werden diese eingehenden Signale zu einem Ausgangssignal umgewandelt. Über das Axon wird das Ausgangssignal der Zelle weitergeleitet.

Ein Künstliches Neuron wird ebenfalls zur Übertragung von Signalen verwendet (McCulloch & Pitts, 1943). Dazu besteht es aus einem Zellkörper mit beliebig vielen Eingangsverbindungen und einer Ausgangsverbindung (Abbildung 3). Die Eingangsverbindungen verbinden ein Künstliches Neuron j mit anderen Künstlichen Neuronen $i = 1, \dots, n$. Jedes Eingangssignal x_i wird mit einem Gewicht w_{ij} multipliziert. Im Zellkörper werden die gewichteten Eingangssignale in der Übertragungsfunktion zu einem Signal net_j verrechnet. Meist ist die Übertragungsfunktion die Summe aller gewichteten Eingangssignale. Die Aktivierungsfunktion φ bestimmt anhand dieses Signals das Ausgangssignal o_j .

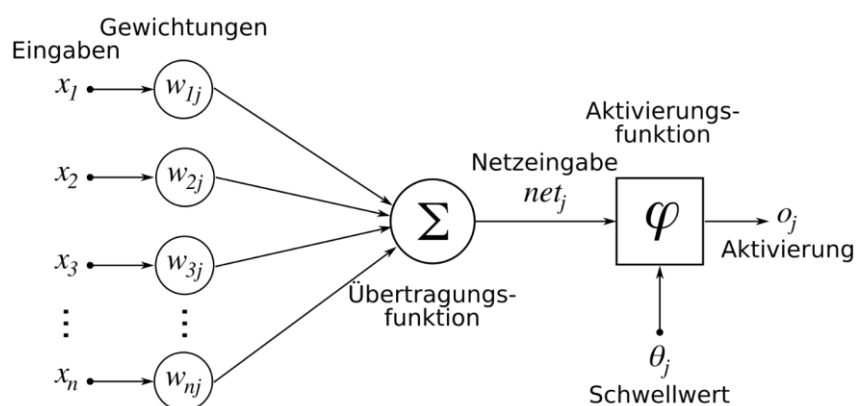


Abbildung 3: Darstellung eines künstlichen Neurons mit seinen Elementen²

Eine Art der Aktivierungsfunktion ist die Sigmoid-Funktion σ (Formel 1).

$$\sigma(x) = \frac{1}{1 + e^x}$$

Formel 1: Sigmoid-Funktion

Diese wird als Aktivierungsfunktion verwendet, da sie einfach differenzierbar ist (Formel 2).

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Formel 2: Ableitung der Sigmoid-Funktion

Ein Neuron kann als Funktion dargestellt werden. Mit der Sigmoid-Funktion als Aktivierungsfunktion und der Summenfunktion als Übertragungsfunktion hat diese Funktion die folgende Form (Formel 3).

¹ Schematische Darstellung einer Nervenzelle, [https://commons.wikimedia.org/wiki/File:Neuron_\(deutsch\)-1.svg](https://commons.wikimedia.org/wiki/File:Neuron_(deutsch)-1.svg), 30.06.2022

² Darstellung eines künstlichen Neurons mit seinen Elementen, https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_deutsch.png, 30.06.2022

$$o_j = \sigma(\text{net}_j) = \sigma\left(\sum_i x_i w_{ij}\right)$$

Formel 3: Output eines künstlichen Neurons

Dabei sind die Gewichte w_{ij} die Parameter dieser Funktion.

2.1.2. Tiefe Neuronale Netze

Ein Künstliches Neuronales Netz (KNN) ist eine Funktion, die eine Menge von Input-Signalen auf eine Menge von Output-Signalen abbildet (Rosenblatt, 1961).

Diese Funktion besteht aus verschiedenen Ebenen von Künstlichen Neuronen (Abbildung 4). Die erste Ebene ist die Input-Ebene. Dort werden die Input-Signale in das Netz eingespeist. Danach gibt es beliebig viele sogenannte Versteckte Ebenen. Diese geben dem KNN eine gewisse Tiefe. Deshalb werden KNNs mit mehreren Versteckten Ebenen Tiefe Neuronale Netze (TNN) genannt. Nach den Versteckten Ebenen kommt zuletzt die Output-Ebene. Diese gibt die Output-Signale aus.

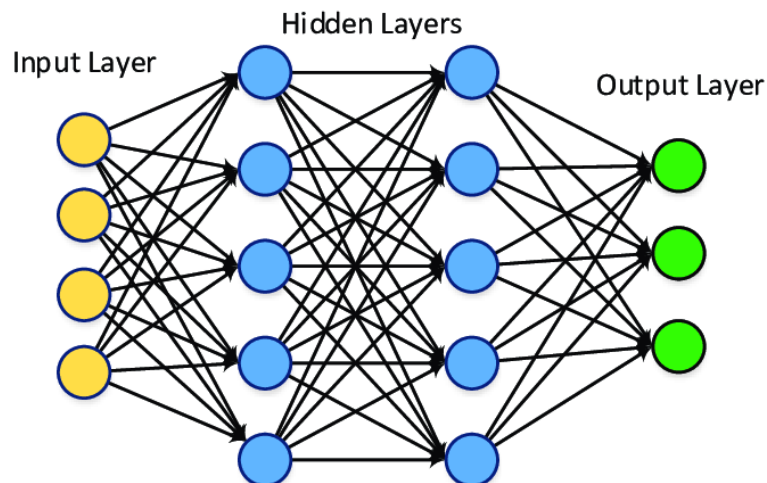


Abbildung 4: Struktur eines tiefen neuronalen Netzes³

Der Input wird Ebene für Ebene an die Output-Ebene weitergegeben. Die Ebenen eines KNNs sind miteinander verbunden. Ein Neuron aus einer Ebene benutzt die Output-Signale der Neuronen der vorherigen Ebene als Input-Signal. Ein Output-Signal eines Neurons wird an Neuronen in der nächsten Ebene weitergeleitet. So wird Information in Form der jeweiligen Output-Signale der einzelnen Neuronen Ebene für Ebene von der Input-Ebene zur Output-Ebene weitergeleitet.

Die Parameter des KNN sind die Gewichte der Künstlichen Neuronen aller Ebenen. So wie ein Künstliches Neuron j von den Gewichten w_{ij} abhängig ist, ist ein Künstliches Neuronales Netz von den Gewichten aller einzelnen Neuronen abhängig. Diese nennt man die Parameter des KNNs. Damit ein

³Architektur eines neuronalen Netzwerks, https://www.researchgate.net/figure/Architecture-of-a-deep-neural-network_fig2_328440759, 30.06.2022

Künstliches Neuronales Netz Input-Signale auf die passenden Output-Signale abbildet, müssen für diese Parameter die passenden Werte gefunden werden.

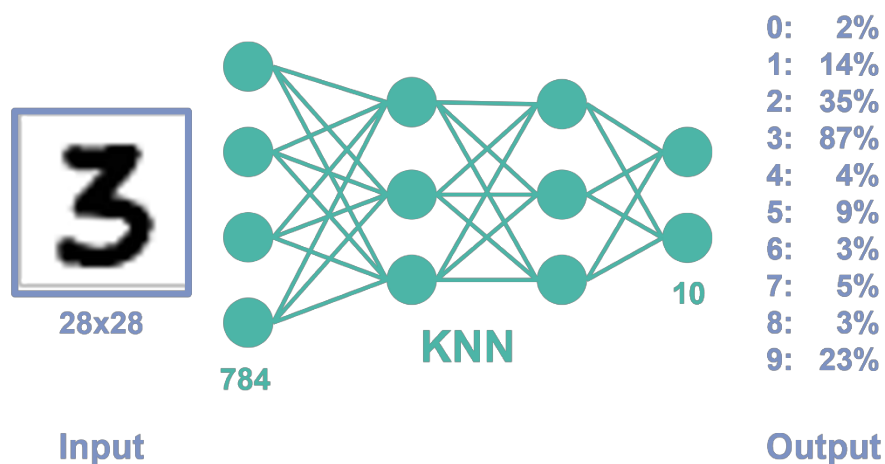


Abbildung 5: Input und Output eines KNNs zur Klassifizierung von handschriebenen Ziffern

Im Beispiel mit den handschriebenen Ziffern ist der Input ein Bild, welches 28 Pixel breit und 28 Pixel hoch ist (Abbildung 5). Da es sich um ein Schwarz-Weiß-Bild handelt, hat jeder Pixel nur einen Helligkeitswert⁴. Alle Werte der Pixel bilden als ein langer Vektor die Input-Ebene des Netzes. Die Input Ebene besteht also aus 784 Neuronen (=28x28).

Der Output des Netzes ist die Wahrscheinlichkeiten, mit denen der Input zu einer bestimmten Ziffer zugeordnet wird. Es gibt 10 verschiedene Ziffern (0, 1, 2, ...9). So gibt es in der Output-Ebene 10 verschiedenen Neuronen. Deren Output ist ein Wert zwischen Null und Eins. Ein Wert von 1 beim ersten Neuron, würde bedeuten, dass das Netz das Input-Bild mit einer Wahrscheinlichkeit von 100% als eine Darstellung der Ziffer Null klassifiziert. Das Neuron mit der höchsten Wahrscheinlichkeit bestimmt die Klassifizierung.

Die Versteckten Ebenen können eine beliebige Anzahl von Neuronen beinhalten. Es gibt verschiedene Arten von Versteckten Ebenen. Bei der sogenannten Fully-Connected-Ebene ist jedes Neuron mit jedem Neuron aus der nächsten Ebene verbunden. Bei einer Convolution-Ebene werden Filter über die Neuronen der Ebene davor gelegt. Dies wird für die Verarbeitung von Bildern verwendet, um so Eigenschaften der Bilder über die Filter zu extrahieren. Die Max-Pooling-Ebene berechnet das Maximum bestimmter Neuronen in der vorigen Ebene. So wird die Dimension im Vergleich zur vorigen Ebene reduziert. Dies verschnellert den Lernprozess, aber könnte Informationen ungewollt herausfiltern. Die Parameter des Netzwerks, also die Gewichte und Bias der Neuronen, werden am Anfang des Trainings meist zufällig zugewiesen. Diese werden dann mit einem Lernalgorithmus angepasst. Ein solcher Lernalgorithmus ist der Backpropagation-Algorithmus.

⁴ Bei einem Farbbild hätte jeder Pixel jeweils einen Wert für den Rot-, Grün- und Blau-Anteil.

2.1.3. Backpropagation

Ein Künstliches Neuronales Netz lernt, indem seine Gewichte angepasst werden. Zunächst hat ein KNN meist zufällig initialisierte Gewichte. Ein Input-Signal würde anhand dieser Gewichte auf ein beliebiges Output-Signal abbilden. Im Beispiel würde ein Bild der Ziffer 6 beliebig auf eine der möglichen Ziffern abgebildet. Damit das KNN einen Input auf den passenden Output abbildet, müssen die Gewichte des KNN angepasst werden. Diese Anpassung der Gewichte ist der Lernprozess eines KNN.

Die Gewichte werden mithilfe eines Datensatzes von gelabelten Trainingsdaten angepasst. Ein Datensatz von gelabelten Trainingsdaten D ist eine Menge aus Input-Signalen, die mit den passenden Output-Signalen als Label (dt. Etikett) versehen sind. Der Label-Wert, also das passende Output-Signal, wird auch als Target (dt. Ziel) bezeichnet. Input x_d und passender Output t_d bilden ein Trainingspaar d (Formel 4).

$$d = (x_d, t_d) \in D$$

Formel 4: Trainingsdatenpaar

Im Beispiel ist der Input ein digitales Bild von einer handgeschriebenen Ziffer mit dem Label, welches die dargestellte Ziffer beinhaltet (Abbildung 6).

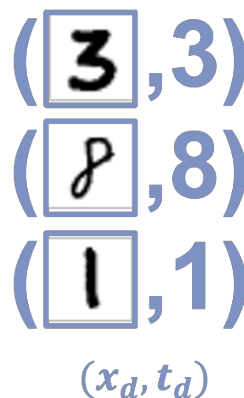


Abbildung 6: Beispiel für gelabelte Trainingsdaten

Mit den gelabelten Trainingsdaten wird der Fehler (engl. Error) des KNNs berechnet. Dazu werden die Input-Signale x_d der Trainingsdaten in das KNN f_w eingespeist. Anhand der Gewichte w wird das Output-Signal y_d des KNNs berechnet (Formel 5).

$$y_d = f_w(x_d)$$

Formel 5: Output eines KNN

Dieses Output-Signal wird mit dem Target t_d verglichen. Der Unterschied von Output-Signal und Target ist der Fehler E_d (engl. Error) eines Trainingspaares d aus den Trainingsdaten D . Dieser wird meist als quadrierter Fehler zwischen Output y_{do} und Target t_{do} aller Output-Neuronen o aus der

Output-Ebene O berechnet (Formel 6). Der Faktor von $\frac{1}{2}$ vereinfacht die später notwendige Ableitung, verändert allerdings nicht das Optimierungsproblem.

$$E_d(w) = \sum_{o \in O} \frac{1}{2} (y_{do} - t_{do})^2, \quad d \in D$$

Formel 6: Fehler eines Trainingspaares

Die Fehler aller Trainingspaare zusammen ergeben den gesamten Fehler E als mittleren quadrierten Fehler (engl. Mean Squared Error MSE). Das heißt, der gesamte Fehler E ist der Durchschnitt der Fehler aller Trainingspaare (Formel 7).

$$E(w) = \frac{1}{|D|} \sum_{d \in D} \sum_{o \in O} \frac{1}{2} (y_{do} - t_{do})^2$$

Formel 7: Fehler des KNNs

Die Anpassung der Gewichte sollte so erfolgen werden, dass der Fehler minimiert wird. Wenn der Fehler kleiner wird, ist der Unterschied zwischen Target und Output kleiner. Somit kann das KNN einen Input genauer auf den passenden Output abbilden. Deshalb sollten die Gewichte so angepasst werden, dass der Fehler minimiert wird.

Gradient Descent (dt. Gadientenabstiegsverfahren) nutzt den Gradienten des Fehlers, um diesen schrittweise zu minimieren. Bei dem Minimierungsalgorithmus Gradient Descent geschieht die Anpassung der Gewichte in die Richtung, in der der Fehler am schnellsten kleiner wird. Diese Richtung ist der negative Gradient des Fehlers bezüglich der Gewichte. Der Gradient ∇E ist der Vektor aller partiellen Ableitungen nach den Gewichten w und beschreibt die Richtung des steilsten Anstiegs von E. Dementsprechend ist der negative Gradient die Richtung des steilsten Abstiegs von E (Formel 8).

$$-\nabla E(D) = - \begin{pmatrix} \frac{\delta E}{\delta w_1} \\ \frac{\delta E}{\delta w_2} \\ \vdots \\ \frac{\delta E}{\delta w_{|w|}} \end{pmatrix}$$

Formel 8: Richtung des steilsten Abstiegs des Fehlers

Der Gradient von E ist nach Formel 8 und der Summenregel der Durchschnitt aller Gradienten des Fehlers aller Trainingspaare (Formel 9).

$$-\nabla E = -\frac{1}{|D|} \sum_{d \in D} \nabla E_d$$

Formel 9: Gradient des Fehlers als Durchschnitt der Fehler eines Trainingspaares

Der Gradient ∇E_d ist wiederum der Vektor aller partiellen Ableitungen nach den Gewichten w (Formel 10).

$$\nabla E_d = \begin{pmatrix} \frac{\delta E_d}{\delta w_1} \\ \frac{\delta E_d}{\delta w_2} \\ \vdots \\ \frac{\delta E_d}{\delta w_{|w|}} \end{pmatrix}$$

Formel 10: Gradient des Fehlers eines Trainingspaares

Die Anpassung der Gewichte bei Gradient Descent geschieht schrittweise (Formel 11). Das heißt E wird Schritt für Schritt kleiner, wenn man die Parameter in Richtung des negativen Gradienten verändert. Die Lernrate η skaliert die Größe der Anpassung pro Schritt. So kann man sich einem lokalen Minimum des Errors annähern.

$$w' = w - \eta \cdot \nabla E$$

Formel 11: Anpassungsregel der Gewichte

Backpropagation (Rumelhart et al., 1986) ist eine Methode den Gradienten ∇E des Errors zu berechnen. Dafür wird angefangen mit der letzten Ebene nacheinander nach den Gewichten jeder einzelnen Ebene abgeleitet. Somit wird der Fehler zurückverfolgt.

Man sucht bei Backpropagation zunächst die partiellen Ableitungen des Errors nach den Gewichten in der Output-Ebene O für ein bestimmtes Trainingspaar d (Formel 12). Die Neuronen in der direkt vorangehenden Ebene P werden mit p beschrieben. Die Gewichte werden nun mit zwei Indexzahlen beschrieben. Diese stehen für die Neuronen an den Enden der zum Gewicht korrespondierenden Verbindung. Das Gewicht w_{po} verbindet demnach ein bestimmtes Neuron p in der vorangehenden Ebene mit einem bestimmten Neuron in der Output-Ebene.

$$\frac{\delta E_d}{\delta w_{po}}$$

Formel 12: Partielle Ableitung des Fehlers eines Trainingspaares

Nach der Kettenregel kann man die Ableitung als Produkt der einzelnen Ableitungen der Error-, Aktivierungs- und Übertragungsfunktion berechnen (Formel 13).

$$\frac{\delta E_d}{\delta w_{po}} = \frac{\delta E_d}{\delta y_o} \cdot \frac{\delta y_o}{\delta net_o} \cdot \frac{\delta net_o}{\delta w_{po}}$$

Formel 13: Partielle Ableitung des Fehlers mit Kettenregel

Mit der Sigmoid-Funktion σ als Aktivierungsfunktion (Formel 1) und der Summenfunktion als Übertragungsfunktion ergeben sich für die einzelnen Ableitungen der Bestandteile (Formel 14).

$$\frac{\delta E_d}{\delta y_o} = (y_o - t_o)$$

$$\frac{\delta y_o}{\delta net_o} = \sigma'(net_o)$$

$$\frac{\delta net_o}{\delta w_{po}} = y_p$$

Formel 14: Ableitungen der Bestandteile der partiellen Ableitung des Fehlers

Daraus ergibt sich die partielle Ableitung nach den Gewichten der Output-Ebene (Formel 15).

$$\frac{\delta E_d}{\delta w_{po}} = (y_o - t_o) \cdot \sigma'(net_o) \cdot y_p$$

Formel 15: Ergebnis der partiellen Ableitung des Fehlers

Danach wird die Partielle Ableitung nach den Gewichten in der vorherigen Ebene mithilfe der Ableitung nach einem Output eines Neurons in der vorherigen Ebene. Die Ableitung des Errors nach Gewichten in der vorherigen Ebene P ergibt sich erneut aus der Kettenregel (Formel 16). Die Ebene vor P wird mit S beschrieben.

$$\frac{\delta E_d}{\delta w_{sp}} = \frac{\delta E_d}{\delta y_p} \cdot \frac{\delta y_p}{\delta net_p} \cdot \frac{\delta net_p}{\delta w_{sp}}$$

Formel 16: Partielle Ableitung des Fehlers nach Gewichten der vorletzten Ebene

Die einzige bisher unbekannte Ableitung ist die Ableitung des Errors eines Trainingspaares d nach einem Output y_p eines Neurons in der vorherigen Ebene P. Da der Output y_p mit allen Neuronen in der Output-Ebene O verbunden ist, hat dieser Auswirkung auf die einzelnen Teile des Fehlers jedes der Output-Neuronen. Die gesuchte Ableitung ist deshalb die Summe der einzelnen Ableitungen über alle Output-Neuronen (Formel 17).

$$\frac{\delta E_d}{\delta y_p} = \sum_{o \in O} \frac{\delta E_d}{\delta y_o} \cdot \frac{\delta y_o}{\delta net_o} \cdot \frac{\delta net_o}{\delta y_p} = \sum_{o \in O} (y_o - t_o) \cdot \sigma'(net_o) \cdot w_{po}$$

Formel 17: Ableitung des Fehlers nach Output der vorletzten Ebene

Somit ergibt sich die partielle Ableitung nach Gewichten in der vorigen Ebene (Formel 18).

$$\begin{aligned} \frac{\delta E_d}{\delta w_{sp}} &= \left(\sum_{o \in O} \frac{\delta E_d}{\delta y_o} \cdot \frac{\delta y_o}{\delta net_o} \cdot \frac{\delta net_o}{\delta y_p} \right) \cdot \frac{\delta y_p}{\delta net_p} \cdot \frac{\delta net_p}{\delta w_{sp}} \\ &= \left(\sum_{o \in O} (y_o - t_o) \cdot \sigma'(net_o) \cdot w_{po} \right) \cdot \sigma'(net_p) \cdot y_s \end{aligned}$$

Formel 18: Ergebnis der partiellen Ableitung des Fehlers nach Gewichten in der vorletzten Ebene

Ähnlich kann man nun die Ableitung bezüglich aller Gewichte in einer beliebigen Ebene L vor der vorletzten Ebene mit einer vorhergehenden Ebene P und einer nachfolgenden Ebene S rekursiv berechnen (Formel 19).

$$\begin{aligned}\frac{\delta E_d}{\delta w_{pl}} &= \frac{\delta E_d}{\delta y_l} \cdot \frac{\delta y_l}{\delta net_l} \cdot \frac{\delta net_l}{\delta w_{pl}} \\ \frac{\delta E_d}{\delta y_l} &= \sum_{s \in S} \frac{\delta E_d}{\delta y_s} \cdot \frac{\delta y_s}{\delta net_s} \cdot \frac{\delta net_s}{\delta y_l} \\ \frac{\delta E_d}{\delta y_s} &= \dots\end{aligned}$$

Formel 19: Partielle Ableitung des Fehlers nach den Gewichten einer beliebigen Ebene

Ausdrücke für die Ableitungen von y und net sind bekannt (Formel 14). Die Rekursion geschieht bis zur letzten Ebene, wo die Ableitung des Fehlers bekannt ist als $y_o - t_o$ (Formel 14).

Somit kann man die partiellen Ableitungen des Errors eines Trainingspaares nach allen Gewichten bestimmen. Diese bilden zusammen in einem Vektor den Gradienten des Errors eines Trainingspaares. Der Durchschnitt der Gradienten aller Trainingspaare ist der Gradient des Errors (Formel 9). Damit lassen sich nun die Gewichte Schritt für Schritt anpassen (Formel 11). Meist wird nicht der gesamte Trainingsdatensatz D für einen Schritt des Gradienten-Verfahrens verwendet. Stattdessen wird er in gleichgroße Teilmengen aufgeteilt und für jede Teilmenge ein Schritt des Gradienten-Verfahrens ausgeführt. Die Teilmengen können bis zu lediglich einem Trainingsdatenpaar groß sein. Sobald jede Teilmenge einmal für einen Schritt im Gradienten-Verfahren verwendet wurde, endet eine Lernepisode. Es können beliebig viele Lernepisoden folgen. Danach sollte das TNN Inputs auf passendere Outputs abbilden können.

2.2. Deep Reinforcement Learning

Reinforcement Learning (RL) ist eine Form des Maschinellen Lernens. Dabei soll eine Software durch Interaktion mit ihrer **Umgebung** eine **Strategie** lernen, um bestimmte Aufgaben zu lösen. Diese Software wird allgemein **Agent** genannt. Der Agent interagiert mit seiner Umgebung, indem er eine Aktion anhand einer Strategie wählt. Er erhält daraufhin eine Belohnung. Auf Basis dieser Belohnung passt der Agent seine Strategie an. Die angepasste Strategie wählt eher Aktionen, die in der Vergangenheit zu einer Belohnung geführt haben. Nach mehreren Anpassungen der Strategie ist der Agent in der Lage die Aufgabe zu lösen. Somit lernt der Agent durch Anpassung seiner Strategie.

Mit Reinforcement Learning Methoden können Agenten Aufgaben in komplexen Umgebungen lösen. Es konnten zum Beispiel performante Agenten für komplexen Strategiespiele wie Go (Silver et al., 2016) oder Dota (Christopher Berner, 2019) und Robotersteuerungen Lillicrap et al. (2019) erstellt werden. Diese weisen eine ähnliche Komplexität wie das Greifproblem auf.

Ein grundlegendes Beispiel für eine mit Reinforcement Learning lösbare Aufgabe ist das sogenannte Mountain-Car-Problem (dt. „Berg-Auto“-Problem) (Moore, 1990). Die Aufgabe besteht darin, ein Auto,

welches in einem Tal steht, zu einem Zielpunkt auf einem Berg zu befördern (Abbildung 7). Der Agent ist hier das Auto und die Umgebung ist der Berg mit dem Tal und dem Zielpunkt auf dem Berg.

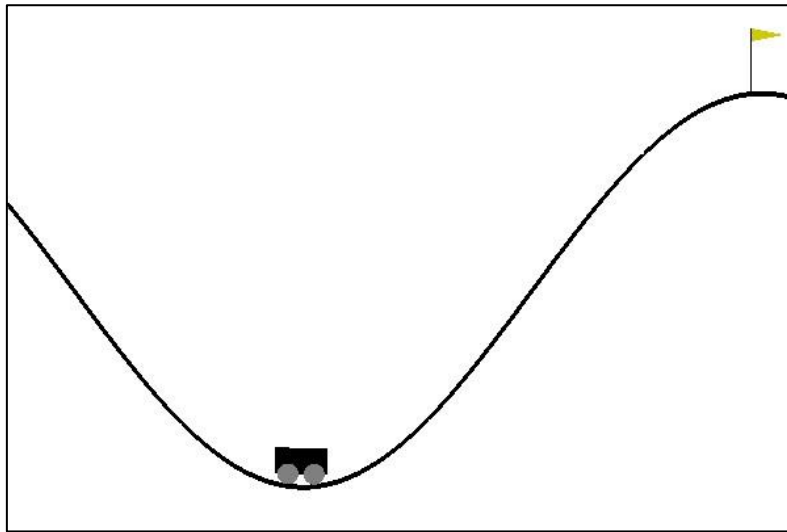


Abbildung 7: Darstellung des Mountain-Car-Problems

Der beschriebene Ansatz, komplexe Aufgaben durch Anpassung der Strategie zu lösen nennt sich modell-freies RL. Ein weiterer Ansatz ist das modell-basierte RL, welches ein Modell der Umgebung lernt, um das Problem zu lösen. Mit einem Modell der Umgebung, kann man bestimmen, welche Aktion zu welchem nächsten Zustand und zu welcher Belohnung führt. In modell-freiem RL ist das Modell der Umgebung unbekannt. In dieser Arbeit wird lediglich modell-freies RL betrachtet.

2.2.1. Umgebung des Agenten

Die Umgebung (engl. Environment) des Agenten bestimmt, wie der Agent mit dieser interagieren kann, was der Agent von dieser wahrnehmen kann und welche Belohnungen er erhält. Mathematisch kann die Umgebung des Agenten (engl. Agent) als Markov-Entscheidungsprozess (Bellman, 1957) beschrieben werden. Ein Markov-Entscheidungsprozess besteht aus einem Zustandsraum S , einem Aktionsraum A , einer Belohnungsfunktion R , einer Zustandsübergangs-Wahrscheinlichkeitsfunktion P und einer Verteilung von Startzuständen ρ_0 (Formel 20).

$$\langle S, A, R, P, \rho_0 \rangle$$

Formel 20: Markov-Entscheidungsprozess

Der Zustandsraum S ist die Menge aller möglichen Zustände s (engl. States) der Umgebung. Bei dem Mountain-Car-Problem ist der Zustand ein Vektor mit zwei Werten. Der erste Wert ist die horizontale Position des Autos. Diese ist als Wert von $-1,2$ bis $0,6$ kodiert. Der zweite Wert ist die Geschwindigkeit des Autos. Dieser hat einen maximalen Wert in beide Richtungen von $0,07$. Daraus ergibt sich die Beschreibung des Zustandsraums S (Formel 21).

$$S = \{s = (p, v) \mid p \in [-1, 2; 0, 6], v \in [-0, 07; 0, 07]\}$$

Formel 21: Zustandsraum des Mountain-Car-Problems

Die Zustände reichen aus der Sicht des Agenten aus, die Umgebung so zu beschreiben, dass dieser Aktionen anhand der Zustände bestimmen kann.

Ein Zustand beinhaltet meist nur einen Teil des realen Zustands der Umgebung. Zum einen ist dies der Fall, da in Umgebungen in der Realität es meist unmöglich ist, den gesamten Zustand der Umgebung wahrzunehmen. In einer Roboteranwendung in der Realität steht dem Roboter als Agenten meist nur seine eigenen Gelenkwinkelzustände und Kamerabilder zur Verfügung. Zum anderen, weil Werte, die wahrscheinlich irrelevant für den Agenten sind, zur Vereinfachung absichtlich ausgelassen werden. Irrelevant könnte ein Wert sein, wenn er wahrscheinlich keine Auswirkung auf die Wahl der Aktion des Agenten hat. Die Farbe des Autos zum Beispiel wäre aus diesem Grund irrelevant. Werte können außerdem irrelevant sein, da sie schon in anderen Werten enthalten sind. Die Beschleunigung des Autos ist zum Beispiel schon in der Geschwindigkeit enthalten, da diese das Ergebnis der Beschleunigung ist. Aus den genannten Gründen wird der Zustand auch manchmal auch lediglich Beobachtung (engl. Observation) der Umgebung genannt.

Der Aktionsraum A ist die Menge aller möglichen Aktionen a (engl. Actions) des Agenten. Bei dem Mountain-Car-Problem sind die Aktionen „nach links beschleunigen“, „nach rechts beschleunigen“ oder „nicht beschleunigen“. Diese Aktionen werden in Form eines Werts dargestellt, der einen bestimmten diskreten Wert annimmt. Dieser diskrete Wert ist einer bestimmten Aktion zugeordnet. Eine „0“ bedeutet zum Beispiel „nach links beschleunigen“, eine „1“ bedeutet „nicht beschleunigen“ und eine „2“ demnach „nach rechts beschleunigen“. Daraus ergibt sich der Aktionsraum (Formel 22).

$$A = \{a \mid a \in \{0, 1, 2\}\}$$

Formel 22: Aktionsraum des Mountain-Car-Problems

Der Aktionsraum wird hier als diskret bezeichnet, da eine Aktion durch einen diskreten Wert beschrieben wird. Der Aktionsraum kann allerdings auch kontinuierlich sein. Dies wäre beim Mountain-Car-Problem der Fall, wenn man einen kontinuierlichen Wert für die Beschleunigung als Aktion auswählt. Das wäre zum Beispiel ein Wert zwischen -1 und 1 .

Die Belohnungsfunktion R bestimmt die Belohnung r (engl. Reward) abhängig vom Zustand. Zusätzlich kann die Belohnung auch von der gewählten Aktion oder dem nächsten Zustand abhängig sein. Im Beispiel erhält der Agent eine Belohnung von -1 , wenn die Position kleiner als $-0,5$ ist, das heißt wenn das Auto links vom Zielpunkt ist. Außerdem erhält er eine Belohnung von 0 , wenn die Position gleich $0,5$ ist, das heißt wenn das Auto den Zielpunkt erreicht hat (Formel 23).

$$r = R(s) = \begin{cases} -1, & p < 0,5 \\ 0, & p = 0,5 \end{cases}$$

Formel 23: Belohnungsfunktion des Mountain-Car-Problems

Die Übergangswahrscheinlichkeitsfunktion P bestimmt die Wahrscheinlichkeit, mit der ein Zustand s in einen anderen Zustand s' übergeht, wenn man eine bestimmte Aktion a wählt (Formel 24).

$$P(s'|s, a)$$

Formel 24: Übergangswahrscheinlichkeitsfunktion

Die dem Markov-Entscheidungsprozess namensgebende Markov-Eigenschaft besagt, dass der Zustandsübergang nur vom aktuellen Zustand und der Aktion abhängt und nicht von vorherigen Zuständen. Im Beispiel spielt es keine Rolle, was das Auto vor dem aktuellen Zustand gemacht hat. Ein anderes Auto in der gleichen Umgebung an der gleichen Position mit der gleichen Geschwindigkeit würde, wenn beide Autos die gleiche Aktion wählen, die gleichen Wahrscheinlichkeiten für den nächsten Zustand bedeuten. Die Übergangswahrscheinlichkeitsfunktion ist dem Agenten unbekannt.

Die Verteilung von Startzuständen ρ_0 bestimmt, in welchem Zustand sich der Agent zum Startzeitpunkt der Aufgabe befindet. Im Beispiel hat das Auto am Anfang immer eine Geschwindigkeit von 0. Die Position am Anfang wird über eine Gleichverteilung U im Intervall von $-0,6$ bis $-0,4$ bestimmt (Formel 25).

$$\rho_0 = \begin{pmatrix} p \\ v \end{pmatrix} = \begin{pmatrix} U(-0,6; -0,4) \\ 0 \end{pmatrix}$$

Formel 25: Verteilung des Startzustände des Mountain-Car-Problems

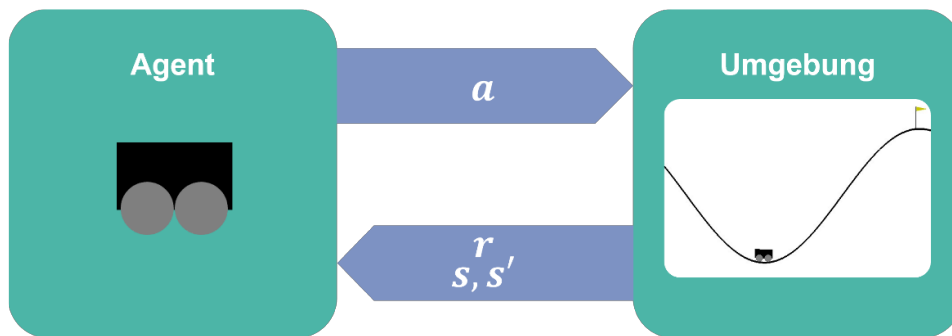


Abbildung 8: Interaktionskreislauf des Agenten

Somit ist die Umgebung des Agenten durch Zustandsraum, Aktionsraum, Belohnungsfunktion, Übergangswahrscheinlichkeitsfunktion und Startzustandsverteilung beschrieben. Man kann anhand dessen die Interaktion des Agenten zu einem beliebigen Zeitpunkt t wie folgt beschreiben. Der Agent nimmt zum Zeitpunkt t den Zustand s_t seiner Umgebung wahr. Dann wählt der Agent eine Aktion a_t . Abhängig von dem neuen Zustand der Umgebung erhält der Agent eine Belohnung r_t . Außerdem nimmt

der Agent den nächsten Zustand s_{t+1} wahr. Damit startet die nächste Interaktion. Dieser Interaktionskreislauf ist in Abbildung 8 dargestellt.

Lernen und Evaluation eines Agenten geschieht in sogenannten Episoden. Eine Episode ist ein Versuch, die Aufgabe zu lösen. Am Anfang einer Episode wird die Umgebung in einen der Startzustände ρ_0 versetzt. Dann wird der oben beschriebene Interaktionskreislauf gestartet. Eine Interaktion im Kreislauf wird ein Zeitschritt genannt. Somit ist eine Episode eine Reihe von Zuständen, Aktionen und Belohnungen (Formel 26).

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2 \dots)$$

Formel 26: Episode

Der Interaktionskreislauf wird unterbrochen, wenn in einem Zeitschritt die Umgebung in einen Zustand gebracht wird, der die Terminierungsbedingungen erfüllt. Solche Bedingungen können die erfolgreiche Lösung der Aufgabe sein oder das Erreichen einer maximalen Anzahl an Zeitschritten für die aktuelle Episode. Im Beispiel wird die Episode terminiert, wenn die Position des Autos größer als 0,5 ist oder wenn 200 Zeitschritte erreicht wurden.

2.2.2. Strategie des Agenten

Die Strategie (engl. Policy) des Agenten bestimmt, welche Aktion der Agent in einem bestimmten Zustand wählt. Durch Anpassung der Strategie lernt der Agent seine Aufgabe zu lösen.

Die Strategie ist eine Funktion μ_θ , die den Zustand s_t auf eine Aktion a_t abbildet (Formel 27). Wie diese Abbildung geschieht, wird durch die Parameter θ der Funktion beeinflusst.

$$a_t = \mu_\theta(s_t)$$

Formel 27: Strategie des Agenten

Die Strategie wird angepasst, indem die Parameter der Strategie angepasst werden. Die Parameter werden so angepasst, dass die Belohnung über alle Episoden maximiert wird. Die kumulativen Belohnungen über eine Episode nennt man Ertrag $R(\tau)$ (englisch: Return). Da man den Ertrag in der Zukunft allerdings nicht kennt, spricht man vom erwarteten Ertrag J . Ziel der Anpassung der Parameter θ ist die Auswahl einer optimalen Strategie μ_θ^* , die bei Ausführung den erwarteten Ertrag J maximiert (Formel 28).

$$\mu_\theta^* = \arg \max_{\mu_\theta} J(\mu_\theta) = \arg \max_{\mu_\theta} \mathbb{E}_{\tau \sim \mu_\theta} [R(\tau)]$$

Formel 28: Optimale Strategie

Mit welchem Ansatz die Strategie erlernt wird, ist abhängig von der Art der Strategie und der Art des Aktionsraums. Strategien können deterministisch oder stochastisch sein. Deterministische Strategien bestimmen für den gleichen Zustand immer die gleiche Aktion. Stochastische Strategien sind eine Wahrscheinlichkeitsverteilung, von der die Aktion gezogen wird. Somit kann im gleichen Zustand unterschiedliche Aktionen gewählt werden. Aktionsräume können wie oben beschrieben diskret oder kontinuierlich sein. Des Weiteren wird zwischen on-policy und off-policy-Lernen unterschieden. Beim

on-policy-Lernen, werden Trainingsdaten verwendet, die nur mit der aktuellen Version der Strategie gesammelt wurde. Bei off-policy-Lernen können diese mit einer beliebigen Strategie gesammelt werden.

Die zwei Hauptansätze von modell-freiem RL sind Policy Optimisation und Q-Learning. Es gibt verschiedene Ansätze zum Lernen der Strategie. Hier werden nur modell-freie Ansätze behandelt. Diese Ansätze können in zwei Gruppen unterteilt werden. Bei Policy Optimisation wird die Strategie explizit repräsentiert. Bei Q-Learning ist die Strategie indirekt durch eine Funktion für den Erwarteten zukünftigen Ertrags repräsentiert. Aber auch Policy Optimisation nutzt meistens eine Approximation einer Funktion für den erwarteten zukünftigen Ertrag. Im Folgenden werden die zwei Hauptansätze Policy Optimisation und Q-Learning vorgestellt und danach ein RL-Algorithmus, der beide Ansätze nutzt. Zunächst werden jedoch die üblichen Funktionen für den erwarteten zukünftigen Ertrag aufgestellt.

2.2.3. Erwarteter zukünftiger Ertrag

Es gibt verschiedene Ansätze wie die Strategie repräsentiert wird. Viele dieser Ansätze nutzen dabei Funktionen für den erwarteten zukünftigen Ertrag J . Solche Funktionen werden Wertfunktionen (engl. Value-Functions) genannt. Der Wert (engl. Value) ist dabei der erwartete zukünftige Ertrag.

Die On-Policy-Wertfunktion bestimmt den erwarteten Ertrag, wenn man ab Zustand s nach Strategie π handelt (Formel 29).

$$V^\pi(s) = E_{\tau \sim \pi}[R(\tau) | s_0 = s]$$

Formel 29: On-Policy-Wertfunktion

Die On-Policy-Aktions-Wertfunktion bestimmt den erwarteten Ertrag einer Aktion a im Zustand s , wenn man danach nach Strategie π handelt (Formel 30).

$$Q^\pi(s, a) = E_{\tau \sim \pi}[R(\tau) | s_0 = s, a_0 = a]$$

Formel 30: On-Policy-Aktions-Wertfunktion

Die Optimale Wertfunktion bestimmt den erwarteten Ertrag, wenn man ab Zustand s nach der optimalen Strategie handelt (Formel 31).

$$V^*(s) = \max_{\pi} E_{\tau \sim \pi}[R(\tau) | s_0 = s]$$

Formel 31: Optimale Wertfunktion

Die Optimale Aktions-Wertfunktion bestimmt den erwarteten Ertrag einer Aktion a im Zustand s , wenn danach nach der optimalen Strategie handelt (Formel 32).

$$Q^*(s, a) = \max_{\pi} E_{\tau \sim \pi}[R(\tau) | s_0 = s, a_0 = a]$$

Formel 32: Optimale Aktions-Wertfunktion

Die Wertfunktionen sind Bellmann Gleichungen. Das bedeutet, dass der erwartete Wert des Anfangszustands die Summe aus dem erhaltenen Wert des Anfangszustands und dem Wert des nächsten Zustands ist.

2.2.4. Policy Optimisation

Policy Optimisation(Sutton et al., 1999) ist ein Ansatz, um die Strategie zu bestimmen. Die Strategie wird explizit repräsentiert. Wir gehen in diesem Kapitel von einer stochastischen Strategie aus. Der Output des TNN ist also die Wahrscheinlichkeitsverteilung, von der die Aktion gezogen wird. Die Gewichte des TNN sollen so angepasst werden, dass der Erwartete Ertrag bei Ausführung der Strategie maximiert wird.

Die Gewichte der Strategie werden mit Gradient Ascent (dt. Gradientenaufstiegsverfahren) schrittweise anhand des Gradienten des Erwarteten Ertrags angepasst. Der Gradient des Erwarteten Ertrags ist der Vektor der partiellen Ableitungen nach den Gewichten und somit auch die Richtung des steilsten Anstiegs des Erwarteten Ertrags. Beim Gradientenaufstiegsverfahren werden die Gewichte schrittweise anhand des Gradienten des Erwarteten Ertrags angepasst. Schrittweise geschieht dies nach der Lernrate α (Formel 33).

$$\theta_{(k+1)} = \theta_k + \alpha \nabla_{\theta} J(\pi_{\theta}) |_{(\theta_k)}$$

Formel 33: Lernregel des Gradientenaufstiegsverfahren

Nach Anpassung der Gewichte wird der Erwartete Ertrag bei Ausführung der Strategie vergrößert. Der Gradient des Erwarteten Ertrag wird auch Policy Gradient genannt.

Der Gradient des Erwarteten Ertrags wird wie folgt beschrieben. Er ergibt sich als Erwartungswert der Summe aller Produkte aus dem Logarithmus des Gradienten der Strategie und dem Ertrag der Trajektorie (Formel 34). Die Trajektorie ist unter der Strategie π_{θ} gesammelt, somit handelt es sich um einen On-Policy-Ansatz.

$$\nabla_{\theta} J(\pi_{\theta}) = E_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau) \right]$$

Formel 34: Gradient des Erwarteten Ertrags

Der Gradient des Erwarteten Ertrags (engl. Policy Gradient) wird anhand mehrerer Trajektorien D geschätzt. Dafür wird der Durchschnitt der Gradienten über alle Trajektorien berechnet (Formel 35).

$$\nabla_{\theta} \hat{J}(\pi_{\theta}) = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) R(\tau)$$

Formel 35: Schätzung des Policy Gradient

Der Policy Gradient kann auf verschiedenen Arten erweitert werden. Zum Beispiel kann statt dem Ertrag der Trajektorie nur der zukünftige Ertrag nach einer Aktion betrachtet werden. Außerdem kann dieser auch mit einer Wertfunktion verrechnet werden. Die Wertfunktion sagt aus, welcher zukünftige Ertrag erwartet wird. Die Strategie wird nur verändert, wenn die Erwartungen an den zukünftigen Ertrag nicht dem entsprechen, was tatsächlich vorgefunden wurde. Dies erhöht die Effizienz der Schätzung des Policy Gradient. Die verschiedenen Möglichkeiten, den Policy Gradient zu erweitern, werden in dieser Arbeit nicht weiter ausgeführt.

Beim Deep Reinforcement Learning Algorithmus **Policy Gradient** wird die Strategie mit einem TNN approximiert. Der Input des TNN ist der Zustand. Der Output des TNN ist die Wahrscheinlichkeitsverteilung, von der die Aktion gezogen wird. Die Gewichte des TNN sollen so angepasst werden, dass der Erwartete Ertrag bei Ausführung der Strategie maximiert wird. Mit Backpropagation kann der Gradient der Strategie berechnet werden.

Zusammenfassend wird bei Policy Optimisation die Strategie verbessert, indem deren Gewichte in Richtung des Gradienten des Erwarteten Ertrags verändert werden. Um den Gradienten effizienter zu Schätzen kann eine Approximation einer Wertfunktion zusätzlich gelernt werden.

2.2.5. Q-Learning

Ein weiterer Ansatz die Strategie zu bestimmen ist Q-Learning (Watkins, 1989). Hier wird die Strategie mithilfe des Erwarteten Ertrags ausgedrückt. Der Agent soll die Aktion wählen, die den maximalen zukünftigen Erwarteten Ertrag hat. Der Erwartete Ertrag wird durch die optimale Aktions-Wert-Funktion $Q^*(s, a)$ beschrieben. Die optimale Strategie wählt in Zustand s immer die Aktion, die den erwarteten zukünftigen Ertrag von s aus maximiert (Formel 36).

$$a^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$$

Formel 36: Optimale Strategie bei Q-Learning

Beim Deep Reinforcement Learning Algorithmus **Deep Q-Network** (Mnih et al., 2013) wird eine Approximation $Q_\phi(s, a)$ der optimalen Aktions-Wert-Funktion $Q^*(s, a)$ gesucht. Deep Q-Network wird für diskrete Aktionsräume verwendet. Diese Approximation geschieht mit einem TNN. Der Input des TNN ist der aktuelle Zustand und eine Aktion. Der Output des TNN ist der erwartete zukünftige Ertrag. Die Trainingssamples d bestehen aus Zustand s_d , Aktion a_d , Belohnung r_d und nächstem Zustand s'_d . Diese Samples können mit einer beliebigen Strategie gesammelt worden sein. Q-Learning geschieht demnach off-policy. Die Zielfunktion für die Anpassung der Gewichte ist der Vorhersagefehler. Der Vorhersagefehler ist der Unterschied zwischen dem vorhergesagten Wert und dem tatsächlichen Wert. Der vorhergesagte Wert für den erwarteten zukünftigen Ertrag ist der Output des TNN. Der tatsächliche Wert, auch Zielwert, ist allerdings unbekannt. Deshalb wird dieser Zielwert geschätzt als Summe aus Belohnung r und dem diskontierten erwarteten zukünftigen Ertrag des nächsten Zustands. γ dabei der Faktor der Diskontierung. Der Vorhersagefehler ist die Mittlere quadratische Abweichung zwischen vorhergesagtem Wert und Zielwert. Die Fehlerfunktion ist der Durchschnitt (Formel 37).

$$E_{pred}(D) = \frac{1}{|D|} \sum_{d \in D} (Q_{\varphi}(s_d, a_d) - (r_d + \gamma \max_a Q_{\varphi}(s'_d, a)))^2, \quad d = (s_d, a_d, r_d, s'_d) \in D$$

Formel 37: Vorhersagefehler Q-Learning

Zur Stabilisierung des Trainings werden Target-Netzwerke verwendet. Da der vorhergesagte Wert und der geschätzte Wert beide mithilfe von Q_{φ} berechnet werden, kann das Training instabil sein. Ein Fehler bei der Vorhersage des zukünftigen erwarteten Ertrags ist auch im geschätzten zukünftigen erwarteten Ertrag zu finden. Um dies stabiler zu machen werden sogenannte Target-Netzwerk eingeführt. Ein Target-Netzwerk ist eine Kopie des TNN, dessen Gewichte zeitversetzt angepasst werden. Dieses Target-Netzwerk wird dann zur Berechnung des geschätzten zukünftigen erwarteten Ertrags benutzt. So verändert sich der Zielwert langsamer und wird nicht von jeder Gewichtsanpassung beeinflusst.

Trainingssamples werden in einem Replay Buffer gespeichert und wiederverwendet. Da Q-Learning off-policy geschieht, können Trainingssamples von einem beliebigen Zeitpunkt für das Training verwendet werden. Um Trainingssamples zu speichern, wird ein Replay Buffer verwendet. Von diesem werden dann kleine Batches von Samples gezogen, die für einen Anpassungsschritt des Netzwerks verwendet werden. Die Kapazität des Replay Buffers ist die maximale Anzahl an Trainingssamples, die im Replay Buffer gespeichert werden können. Wenn der Replay Buffer seine Kapazität überschreitet, werden alte Trainingssamples gelöscht. Es gibt unterschiedliche Ansätze, welche Trainingssamples gelöscht werden sollen. Durch das Löschen von Trainingssamples könnte vorher Gelerntes vergessen werden. Die Kapazität des Replay Buffers sollte groß genug sein, um vielfältige Trainingssamples zu sammeln. Eine zu große Kapazität kann das Training verlangsamen, da alte Trainingssamples weniger relevant sein könnten für die Anpassung der aktuellen Strategie.

Zusammenfassend wird bei Q-Learning eine Approximation der Aktions-Wertfunktion gelernt, deren Maximum über alle möglichen Aktionen die von der Strategie gewählte Aktion ist. Da Q-Learning off-policy geschieht, können Daten wiederverwendet werden. Somit ist die Trainingsdatensammlung effizienter. Ein Nachteil von Q-Learning ist, dass es instabiler ist. Dafür wird ein Target Netzwerk verwendet.

2.2.6. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015) ist ein Deep Reinforcement Learning-Algorithmus, der Elemente von Policy Optimisation und Q-Learning enthält. Dieser erlernt eine explizite deterministische Strategie und eine Approximation einer Aktions-Wertfunktion. Beide werden zum Verbessern der jeweils anderen verwendet.

DDPG ermöglicht Q-Learning in kontinuierlichen Aktionsräumen anzuwenden. Bei Q-Learning in diskreten Aktionsräumen wird das Maximum des Erwarteten zukünftigen Ertrags über alle möglichen Aktionen berechnet, um die optimale Aktion zu bestimmen. In kontinuierlichen Aktionsräumen ist die Berechnung des Maximums über den gesamten Aktionsraum ressourcenaufwändig. Bei DDPG wird

neben der Approximation der Aktions-Wertfunktion Q eine explizite Strategie μ approximiert, welche die Aktion mit dem maximalen erwarteten zukünftigen Ertrag wählt (Formel 38). So kann die Bestimmung des Maximums über die Strategie vorgenommen werden.

$$\max_a Q(s, a) \approx Q(s, \mu(s))$$

Formel 38: Approximation des maximalen erwarteten zukünftigen Ertrags

DDPG nutzt zwei Arten von TNN: Das Actor-Netzwerk für die Approximation der Strategie und das Critic-Netzwerk für die Approximation der Aktions-Wertfunktion. Das Actor-Netzwerk μ_θ (kurz: Actor) bildet den Zustand s auf die Aktion a ab, die den zukünftigen erwarteten Ertrag maximiert. Das Critic-Netzwerk Q_ϕ (kurz: Critic) bildet den Zustand s und Aktion a auf den zukünftigen erwarteten Ertrag ab.

Actor und Critic werden gegenseitig für die Anpassung ihrer Gewichte verwendet und verwenden einen Replay Buffer. Wie bei dem Algorithmus Deep Q-Network kann das Training off-policy geschehen. Deshalb wird bei DDPG ein Replay Buffer (1.3.2) verwendet. Wenn die Kapazität des Replay Buffers überschritten ist, werden die ältesten Samples gelöscht. Bei der Anpassung wird ein kleiner Satz (engl. batch) von Trainingssamples D vom Replay Buffer gleichverteilt gezogen. Ein Trainingssample d besteht aus dem Zustand s_d , Aktion a_d , Belohnung r_d und dem nächsten Zustand s'_d .

Die Gewichte des Critic werden wie bei Q-Learning anhand des Vorhersagefehlers mithilfe von Target-Netzwerken angepasst. Zur Anpassung der Gewichte des Critic wird der Vorhersagefehler (Formel 21) als Zielfunktion verwendet. Allerdings wird der Zielwert mit zwei Target-Netzwerken (Siehe 1.3.2) berechnet. Es gibt ein Target-Netzwerk für den Actor $\mu_{\theta'}$ (kurz: Actor-Target) und für den Critic $Q_{\phi'}$ (kurz: Critic-Target). Die Gewichte der Target-Netzwerke werden schrittweise an die aktuellen Gewichte ihrer Äquivalente angepasst (Formel 39).

$$\begin{aligned}\theta' &\leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta', \tau \ll 1 \\ \phi' &\leftarrow \tau \cdot \phi + (1 - \tau) \cdot \phi', \tau \ll 1\end{aligned}$$

Formel 39: Anpassung der Gewichte der Target-Netzwerke

Der Zielwert wird mit dem Output des Critic-Targets, welcher als Input den Zustand s und den Output des Actor-Targets benutzt. Mit dem berechneten Zielwert wird der Vorhersagefehler bestimmt (Formel 40).

$$E_{critic}(D) = \frac{1}{|D|} \sum_{d \in D} (Q_\phi(s_d, a_d) - (r_d + \gamma Q_{\phi'}(s'_d, \mu_{\theta'}(s'_d)))^2, \quad d = (s_d, a_d, r_d, s'_d) \in D$$

Formel 40: Vorhersagefehler des Critics

Der Vorhersagefehler soll minimiert werden (Formel 41). Man leitet dazu den Vorhersagefehler nach den Gewichten des Critics ab und verändert diese in Richtung des negativen Gradienten des Vorhersagefehlers.

$$\min_{\phi} E_{critic}(D)$$

Formel 41: Optimierungsproblem des Critics

Die Gewichte des Actor werden anhand der Ertragsfunktion angepasst. Zur Anpassung der Gewichte des Actor wird eine Ertragsfunktion als Zielfunktion verwendet. Die Ertragsfunktion ist der Output des Critic des aktuellen Outputs des Actors (Formel 42).

$$R_{actor}(D) = \frac{1}{|D|} \sum_{d \in D} Q_{\phi}(s_d, \mu_{\theta}(s_d)), \quad d = (s_d, a_d, r_d, s'_d) \in D$$

Formel 42: Ertragsfunktion des Actors

Die Ertragsfunktion soll maximiert werden. Man leitet dazu die Ertragsfunktion nach den Gewichten des Actors ab und verändert diese in Richtung des positiven Gradienten der Ertragsfunktion (Formel 43).

$$\max_{\theta} R_{actor}(D)$$

Formel 43: Optimierungsproblem des Actors

Um die Exploration des Zustandsraums zu verbessern, wird beim Training Rauschen zu den Aktionen hinzugefügt. Die umfangreiche Exploration des Zustandsraums ist wichtig, um vielfältige Trainingssamples zu sammeln und Belohnungen zu finden. Da der Actor Aktionen deterministisch bestimmt, haben diese eine geringe Variabilität. Dies führt zu begrenzter Exploration. Deshalb wird den Aktionen während des Trainings ein Rauschen hinzugefügt. DDPG nutzt für die Verteilung des Rauschens einen Ornstein-Uhlenbeck-Prozess (Finch, 2004), auf den hier nicht weiter eingegangen wird. Ebenfalls möglich ist eine Gauss-Verteilung oder eine Gleichverteilung.

Zusammenfassend verbindet DDPG Elemente von Q-Learning und Policy Optimisation und ermöglicht eine Anwendung des Ersteren in kontinuierlichen Aktionsräumen. Die zwei Arten von TNNs Actor und Critic werden gegenseitig für das Anpassen ihrer Gewichte verwendet. Es wird ein Replay Buffer, Target-Netzwerke und ein Rauschen für die Aktionen während des Trainings verwendet.

2.3. Robot Operating System

Das Robot Operating System (ROS) (Quigley, 2009) ist ein Framework zur Steuerung von Robotern. Die Steuerung geschieht über mehrere kleine, parallellaufende Unterprogramme, die sogenannten *ROS-Nodes* (dt. Knoten, kurz: Nodes). Nodes können über sogenannte *ROS-Messages* (dt. Nachrichten, kurz: Message) kommunizieren. Eine Message ist eine Sammlung von Variablen mit definiertem Datentyp. Die Variablen werden zur Nachrichtenübertragung einem Wert zugewiesen und versendet. Hierfür gibt es zwei standardisierte Kommunikationswege zwischen den Nodes: *ROS-Topics* (dt. Themen) und *ROS-Services* (dt. Dienst) (Abbildung 9).

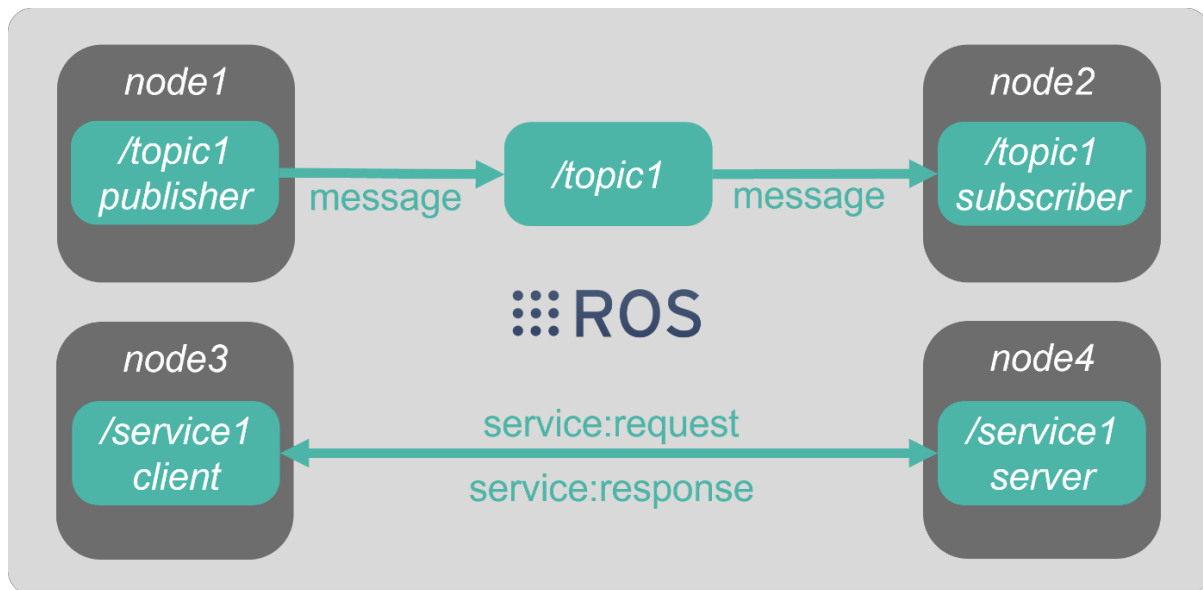


Abbildung 9: ROS-Kommunikationsstruktur mit ROS-Services und ROS-Topics

Nodes können über ROS-Topics (kurz: Topic) Messages wie bei einem Feed versenden. Nodes können *Publisher* (dt. Herausgeber) oder *Subscriber* (dt. Abonnent) eines Topics sein. Als Publisher können sie eine Message an ein bestimmtes Topic veröffentlichen. Als Subscriber können sie ein bestimmtes Topic abonnieren. Sobald eine Node eine Message an ein Topic veröffentlicht, erhalten alle Subscriber die Message und können auf die übergebenen Daten zugreifen. Ein Topic ist demnach ein Feed, auf den veröffentlicht werden kann und der abonniert werden kann. Man benutzt Topics, wenn auf ein Datenstrom kontinuierlich reagiert werden soll. Ein Beispiel für ein Topic sind die Gelenkwinkel des Roboters, die vom Controller veröffentlicht werden.

Nodes können über ROS-Services (kurz: Service) Funktionen anderer Nodes aufrufen wie bei einem Remote Procedure Call. Nodes können ein *Server* (dt. Server) oder ein *Client* (dt. Klient) eines Service sein. Als Server bieten sie einen Service an und machen ihn anderen Nodes damit verfügbar. Als Client eines Service können sie diesen aufrufen. Ein Service ist definiert als Sammlung von Variablen mit definiertem Datentyp, die in *Request* (dt. Anfrage) und *Response* (dt. Antwort) unterteilt ist. Der Client schickt die Request an den Server und erhält die Response zurück. Ein bestimmter Service kann nur einen Server, aber beliebig viele Clients haben. Man benutzt Services, wenn man zu einem bestimmten Zeitpunkt eine Funktion einer anderen Node ausführen möchte, die eine Antwort zurückgibt. Eine Simulationssoftware könnte zum Beispiel einen Service anbieten, um die Position eines 3D-Modells in der Simulation zu verändern. Die Request könnte dafür den Namen des Modells und die gewünschte Position sein. Die Response könnte eine Statusnachricht beinhalten.

ROS bietet die Möglichkeit mit verschiedenen Programmiersprachen, auf verschiedenen Systemen Roboter zu programmieren und zu steuern und ist in weitere Software für Robotik integriert. Es besteht ein Python und ein C++-Interface für ROS. Da die Nodes über standardisierte Kommunikationswege interagieren, können Nodes in Python einfach mit Nodes in C++ kommunizieren. Da die verschiedenen Nodes unabhängig voneinander ausgeführt werden, können diese im gleichen

Netzwerk auch auf unterschiedlichen Systemen ausgeführt werden. Ein weiterer Vorteil von ROS ist, dass Software für Robotik häufig in ROS integriert ist. So gibt es zum Beispiel ROS-Schnittstellen für Simulations- und Bewegungsplanungssoftware.

2.4. Franka Emika Robotersystem

Das *Franka Emika-Robotersystem*⁵ besteht aus einem Arm und einem Controller. Es wird von dem Unternehmen *Franka Emika* produziert. Der Arm wird auch *Panda* genannt und verfügt über 7 Gelenke. An jedem Gelenk befindet sich ein Drehmomentsensor. Als Endeffektor wird in dieser Arbeit die *Panda Hand* verwendet, welches der Zwei-Finger-Greifer von Franka Emika ist (Abbildung 10). Der Roboter wird in Forschung und Produktion angewandt. Er hat eine Nutzlast von 3kg und einer Reichweite von 855mm.



Abbildung 10: Franka Emika Robotersystem mit Hand

Der Panda bietet einen einfachen Einstieg in die Roboterprogrammierung, ist für eine Greifanwendung geeignet und in bestehende Robotik-Software integriert. Der Panda wird in Robotik-Tutorials als Standardroboter verwendet. Dies ermöglicht einen einfacheren Einstieg in die Programmierung des Roboters. Der Endeffektor des Roboters ist ein Zwei-Finger-Greifer. Dieser ist im Vergleich zu Greifern mit mehr Fingern weniger komplex anzusteuern. Somit eignet er sich für eine grundlegende Greifanwendung. Mit dem Franka Control Interface bietet der Panda die C++-Schnittstelle *libfranka* und eine Integration von ROS über das Paket *franka_ros*. Dies erlaubt eine Steuerung des Roboters mit Nodes, Messages und Services. Das *franka_gazebo*-Paket enthält des Weiteren Dateien zur Integration des Roboters in der Simulationssoftware Gazebo. Das Paket

⁵ Quelle: <https://www.franka.de/research>

panda_moveit_config integriert den Panda in die Bewegungsplanungssoftware MoveIt. Somit ist der Panda in bestehende Robotik-Software integriert.

2.5. Gazebo Simulationssoftware

Gazebo(Koenig & Howard, 2004) ist eine Software zur Simulation von Roboteranwendungen in ihrer Umgebung. Durch die Spezialisierung auf Robotik bietet sie besondere Eigenschaften, die für diese Arbeit nötig sind.

Die Simulation von Roboteranwendungen ist sicherer und effizienter als die Ausführung in der realen Umgebung. Bei der Ausführung von Roboteranwendungen in der realen Umgebung gibt es Sicherheitsrisiken. Unerwartete Bewegungen können zu Verletzungen und Beschädigungen führen. Außerdem ist die Ausführung in der realen Umgebung kosten- und zeitaufwändiger. Zu den Anschaffungskosten kommen außerdem Betriebskosten. Des Weiteren kommt es während der Ausführung zu Verschleiß. In der Simulation kann eine Anwendung schneller als in Echtzeit ausgeführt und getestet werden. Außerdem gibt es keinen Verschleiß und nur geringe Betriebskosten. Sicherheitsrisiken bestehen nicht in der Simulation.

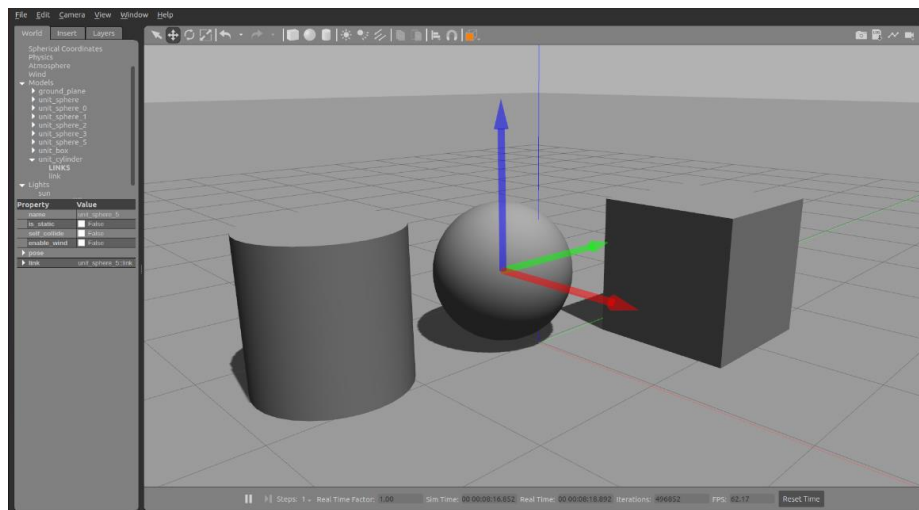


Abbildung 11: Gazebo-Benutzeroberfläche mit Beispiel-Objekten

Gazebo simuliert und visualisiert verschiedene Elemente mithilfe einer Physik-, Sensor- und Grafik-Engine. Die Elemente, die simuliert werden sollen, sind Roboter, Lichter, Sensoren und weitere Objekte. Diese Elemente sind in verschiedenen XML-Formaten beschrieben. Alle Elemente sind in der sogenannten World-Datei (dt. Welt-Datei) beschrieben. Diese ist im Simulation Description Format (SDF) verfasst. Roboter werden im Unified Robot Description Format (URDF) beschrieben. Die Beschreibungen der Elemente werden von der Physik- und Sensoren-Engine ausgelesen, welche die Elemente anhand dessen simuliert. Die Grafik-Engine visualisiert die Simulation in einer Benutzeroberfläche (Abbildung 11).

Plug-Ins ermöglichen unter anderem die Steuerung eines Roboters und die Generierung von Sensordaten. Es können in den Beschreibungen der Elemente Plug-Ins hinzugefügt werden. Diese werden von Gazebo geladen und interpretiert. Plug-Ins können zur Steuerung des Roboters oder zur

Generierung von Kameradaten verwendet werden. Dies ermöglicht eine Interaktion des Roboters mit seiner Umgebung.

Gazebo ist speziell für Roboteranwendungen geeignet durch die realistische Physiksimulation, die Integration von ROS und die Simulation von Sensoren. Gazebo bietet eine realistische Physiksimulation. Dies ist besonders wichtig für das Testen von Roboteranwendungen in der Simulation, da dies unter realistischen Bedingungen durchgeführt werden kann. Gazebo ist mit den *gazebo_ros_packages* in ROS integriert. Diese Pakete enthalten Schnittstellen, um Gazebo als Node zu starten, den Roboter über Messages und Services zu steuern, die Simulation über Messages und Services zu konfigurieren und Sensordaten über Topics zu veröffentlichen. Die Simulation von Sensoren ist entscheidend für Robotik, da dies die einzige Möglichkeit der Wahrnehmung für Roboter ist. Zusammenfassend bietet Gazebo Eigenschaften, die speziell für die Robotik geeignet sind.

Die Simulation in dieser Arbeit nutzt die verschiedenen Eigenschaften von Gazebo. Die in dieser Arbeit verwendete Simulationsumgebung basiert auf einer leeren World-Datei geladen. Dort sind nur die Physik-Parameter und die Lichter beschrieben. In diese wird darauf zunächst der Roboter mit einer Tiefenkamera am Endeffektor geladen. Der Roboter und die Kamera beinhalten ebenfalls Plug-Ins zur Robotersteuerung und Sensorsimulation mit ROS. Des Weiteren werden eine Tischplatte und ein Würfel geladen. Nun kann der Roboter angesteuert werden und die Roboteranwendung simuliert werden. Für das Greifen des Würfels ist die realistische Physiksimulation entscheidend.

2.6. MoveIt Bewegungsplanungssoftware

MoveIt (Coleman et al., 2014) ist eine Bewegungsplanungssoftware für Roboter. Sie kommuniziert mit der Robotersteuerung und bietet Funktionen zur Planung und Ausführung von Bewegungen. Es können beliebige Posen angefragt werden. MoveIt berechnet die Trajektorie zu der gewünschten Position und führt diese mit der Robotersteuerung aus. Dabei berechnet MoveIt auch die Inverse Kinematik des Roboters. Mit RViz kann die Roboterkonfiguration und die geplante Trajektorie visualisiert werden (Abbildung 12).

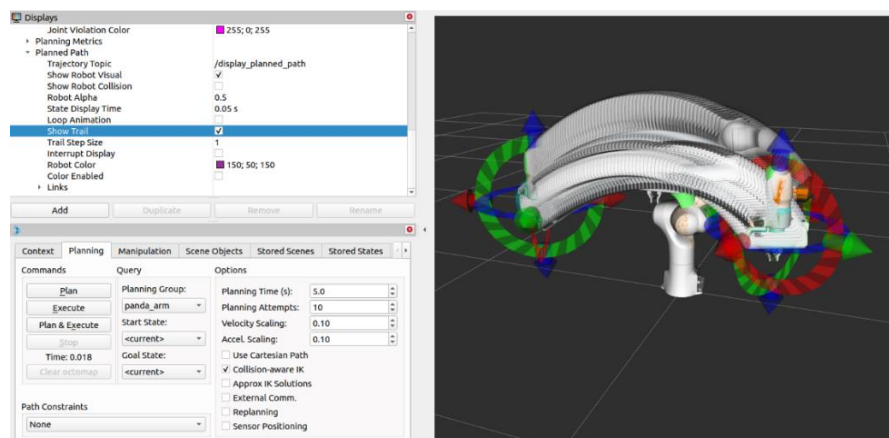


Abbildung 12: RViz-Benutzeroberfläche mit dargestellter Trajektorie

2.7. Intel RealSense D435 Tiefenkamera

Die *Intel RealSense D435*⁶ ist eine Tiefenkamera, die ein weites Sichtfeld hat und besonders für Anwendungen mit schnellen Bewegungen geeignet ist. Durch ihr weites Sichtfeld eignet sie sich besonders für Roboteranwendungen, da ein Großteil der Umgebung wahrgenommen werden kann. Die Kamera kann RGB- und RGBD-Bilder aufnehmen. Die Tiefenkamera hat eine Reichweite von 10m. Über die *Intel RealSense SDK2.0* ist die Tiefenkamera über Schnittstellen in zahlreichen Frameworks und Programmiersprachen integriert. Unter anderem existiert eine Integration in ROS, OpenCV, Python und C++.



Abbildung 13: Intel RealSense D435 Tiefenkamera

3. Forschungsstand

In der Literatur gibt es verschiedene Beispiele für Robotersysteme zum Greifen, die Deep Reinforcement Learning-Ansätze nutzen. Popov et al. (2017) entwickelten ein System, welches mithilfe von Deep Q-Learning lernt, in einer Simulationsumgebung Lego Duplo Steine zu greifen und zu stapeln. Song et al. (2020) und Chen et al. (2021) implementierten ein System in einer realen Laborumgebung mit einer am Endeffektor montierten RGBD-Kamera, welches auch mit Deep Q-Learning lernte. Für ein ähnliches System nutzte Kalashnikov et al. (2018) RGB-Bilder von einer „über der Schulter“ montierten Kamera und eine eigens entwickelte Deep Q-Learning Methode. Zum Transfer von der Simulation in die Realität wurden verschiedene Ansätze von Domain Randomisation (Peng et al. (2018), (2017), Tobin et al. (2017)) und Domain Adaption (Bousmalis et al. (2017)) implementiert. Domain Randomisation umfasst eine Variation verschiedener Umgebungsparameter in der Simulation, sodass die Realität nur als weitere Variation der Umgebung aufgefasst wird. Domain Adaption versucht die Simulation und die reale Umgebung anzupassen, benötigt dafür allerdings Trainingsdaten aus der Realität (Kleeberger et al., 2020).

⁶ Quelle: <https://www.intelrealsense.com/depth-camera-d435/>

4. Konzept

Das System besteht aus dem Agenten und seiner Umgebung. Die Umgebung soll als Markov-Entscheidungsprozess definiert sein und muss in einer Simulationsumgebung und einer Realumgebung implementiert werden. Der Agent soll eine Strategie mithilfe des Deep Deterministic Policy Gradient Algorithmus erlernen. Dazu müssen ein Actor- und ein Critic-Netzwerk definiert werden. Ein Replay Buffer sammelt Trainingssamples. Außerdem sollen Target-Netzwerke implementiert werden.

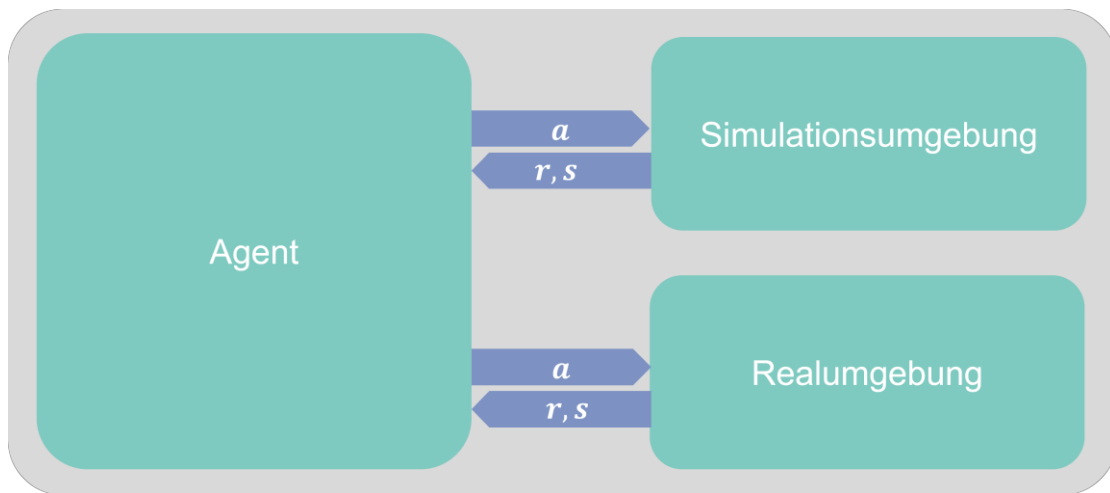


Abbildung 14: Konzeptueller Überblick über das System

4.1. Umgebung

Die Umgebung des Agenten besteht aus einem Roboterarm und dem zu greifenden Objekt. Am Endeffektor des Roboterarms ist ein Zwei-Finger-Greifer und eine RGBD-Kamera montiert. Das zu Greifende Objekt ist ein Würfel und liegt auf einer flachen Oberfläche. Die Umgebung kann als Markov-Entscheidungsprozess (Siehe 2.2.1) beschrieben werden. Die Umgebung muss in der Simulation und der Realität implementiert werden und Funktionen zum Zurücksetzen und Ausführen einer Aktion haben.

4.1.1. Markov-Entscheidungsprozess

Die Umgebung kann als **Markov-Entscheidungsprozess** beschrieben werden (siehe 2.2.1). Dazu werden im Folgenden der Zustandsraum, der Aktionsraum, die Belohnungsfunktion und die Verteilung der Startzustände definiert.

Der **Zustandsraum** der Umgebung ist die Menge aller möglichen Zustände der Umgebung (Formel 44). Ein Zustand besteht aus den Pixeln der letzten drei Tiefenbilder der RGBD-Kamera, den kartesischen Koordinaten der Endeffektorposition und den Gelenkwinkeln. Die Anzahl der Variablen, die ein Tiefenbild beschreibt, entspricht der Anzahl der Pixel. Die Anzahl der Pixel ist das Produkt aus Bildhöhe h und Bildbreite w . Jede dieser Variablen kann Werte zwischen 0 und 255 annehmen. Die Koordinaten der Endeffektorposition können Werte zwischen den Grenzen des Arbeitsraums in die

jeweilige Richtung haben. Die minimalen Grenzen werden als $x_{min}, y_{min}, z_{min}$ und die maximalen Grenzen als $x_{max}, y_{max}, z_{max}$ bezeichnet. Die Gelenkwinkel $\omega_1, \dots, \omega_7$ können Werte zwischen $-\pi$ und π annehmen.

$$S = \left\{ s \mid \begin{array}{l} p_1, \dots, p_{h,w} \in [0; 255], \omega_1, \dots, \omega_7 \in [-\pi, \pi], \\ x_{ee} \in [x_{min}, x_{max}], y_{ee} \in [y_{min}, y_{max}], z_{ee} \in [z_{min}, z_{max}], \end{array} \right\}$$

$$s = (p_1, \dots, p_{h,w}, x_{ee}, y_{ee}, z_{ee}, \omega_1, \dots, \omega_7)$$

Formel 44: Zustandsraum der Greif-Problems

Der **Aktionsraum** des Agenten ist die Menge aller möglichen Aktionen des Agenten (Formel 45). Eine Aktion besteht aus der kartesischen Bewegung des Endeffektors und der Rotation um die z-Achse. Die kartesische Bewegung besteht aus den drei Variablen t_x, t_y, t_z . Diese beschreiben die Translation entlang der drei kartesischen Achsen x, y, z . Die Rotation um die z-Achse wird mit der Variablen r_z beschreiben. Die Variablen für die Translation können negative und positive Werte mit Betrag kleiner als t_{max} annehmen. Die Variable für die Rotation können negative und positive Werte mit Betrag kleiner als r_{max} annehmen. Das Greifen geschieht automatisch, wenn die Endeffektorposition unter einer bestimmten Höhe ist.

$$A = \{ a = (t_x, t_y, t_z, r_z) \mid t_x, t_y, t_z \in [-t_{max}, t_{max}], r_z \in [-r_{max}, r_{max}] \}$$

Formel 45: Aktionsraum des Greif-Problems

Die **Belohnungsfunktion** bestimmt die Belohnung des Agenten in Abhängigkeit vom Zustand, der Aktion und dem nächsten Zustand. Der Agent erhält eine positive Belohnung von $r_{success}$, wenn er nach einem Greifbefehl das Objekt erfolgreich gegriffen hat. Der Agent erhält eine Belohnung von r_{failed} , wenn er nach einem Greifbefehl das Objekt nicht erfolgreich gegriffen hat. Außerdem erhält der Agent eine positive Belohnung von r_{height} abhängig von der Greiferhöhe bei jedem Zeitschritt. Je niedriger der Greifer, desto größer ist die Belohnung. Dies soll das Lernen erleichtern, da eine Belohnung dadurch häufiger vorkommt und man dem Agenten dadurch das Wissen vermittelt, dass er sich zunächst auf Höhe der Tischplatte bewegen muss, um ein Objekt greifen zu können. Zuletzt erhält der Agent eine Belohnung r_{out} , wenn eine relative Bewegung außerhalb des Arbeitsraums führt oder von der Robotersteuerung nicht ausgeführt werden kann (Formel 46).

$$r = R(s) = \begin{cases} r_{success}, & \text{Greifen erfolgreich} \\ r_{failed}, & \text{Greifen fehlgeschlagen} \\ r_{out}, & x_{ee} \notin [x_{min}, x_{max}], y_{ee} \notin [y_{min}, y_{max}], z_{ee} \notin [z_{min}, z_{max}] \\ r_{height}, & \text{Sonst} \end{cases}$$

Formel 46: Belohnungsfunktion des Greif-Problems

Die **Verteilung von Startzuständen** bestimmt den Zustand der Umgebung am Anfang einer Episode. Der Agent befindet sich immer in einem festen Startzustand. Dieser besteht aus einer Startpose und dem Status des Greifers. Die Startpose ist bestimmt durch die Startposition und die Startorientierung. Die Startposition in kartesischen Koordinaten wird durch $x_{init}, y_{init}, z_{init}$ beschrieben. Diese werden so gewählt, dass der Greifer zentriert über der Tischplatte positioniert ist. Die Startorientierung wird durch

$r_{x,init}$, $r_{y,init}$, $r_{z,init}$ beschrieben. Diese werden so gewählt, dass der Greifer der Tischplatte zugewandt ist. Die Würfelposition ist randomisiert um das Zentrum der Tischplatte. Dabei wird die Randomisierung der x und y Position jeweils durch eine Gleichverteilung zwischen $-t_{cube,max}$ und $+t_{cube,max}$ beschrieben.

4.1.2. Simulations- und Realumgebung

Die Umgebung des Agenten wird für das Training mithilfe einer Simulationssoftware simuliert. Das trainierte Modell wird danach auch in einer Realumgebung evaluiert. Die Realumgebung ist eine Laborumgebung in der Realität. Für Simulationsumgebung und Realumgebung müssen unterschiedliche Schnittstellen implementiert werden. Diese Schnittstellen benötigen Funktionen zum Zurücksetzen der Umgebung und zum Ausführen einer Aktion.

4.2. Agent

Der Agent soll eine Strategie unter Anwendung des Deep Deterministic Policy Gradient-Algorithmus (siehe 2.2.6) erlernen. Der Agent besteht aus einem Modell zum Erlernen seiner Strategie und einem Replay Buffer, der Trainingssamples speichert.

Das Modell besteht aus zwei Typen von Funktionen, die approximiert werden sollen: der Actor μ und der Critic Q . Der Critic bestimmt den erwarteten zukünftigen Ertrag eines Zustands s und einer Aktion a . Der Actor bestimmt anhand des Zustands s die Aktion a , die den erwarteten zukünftigen Ertrag maximiert.

Die beiden Funktionen werden durch TNN's approximiert: Das Actor-Netzwerk μ_θ mit Gewichten θ und das Critic-Netzwerk Q_φ mit Gewichten φ . Die Gewichte des Critic-Netzwerks werden angepasst anhand des Fehlers im Vergleich zum tatsächlichen zukünftigen Ertrag. Die Gewichte des Actor-Netzwerk werden angepasst, indem der erwartete zukünftige Ertrag der von ihm bestimmten Aktionen maximiert wird. Für die Berechnung des tatsächlichen zukünftigen Ertrags werden Target-Netzwerke verwendet.

4.2.1. Actor-Netzwerk

Das Actor Netzwerk besteht aus einem Inputnetzwerk für die Tiefenbilder, einem Inputnetzwerk für die Greiferhöhe, einem Hauptnetzwerk und der Output-Netzwerk.

Das **Input-Netzwerk für die Tiefenbilder** verarbeitet die Bildinputs und extrahiert Merkmale aus ihnen. Dafür werden mehrere Input-, Convolution- und Max-Pooling-Ebenen verwendet. Das Netzwerk besteht aus 17 Ebenen. Die erste Ebene ist eine Vereinigung aus jeweils einer Input-Ebene für die Tiefenbilder. Darauf folgen 13 Convolution-Ebenen und 3 Max-pooling-Ebenen (siehe Grafik Ebenen).

Das **Input-Netzwerk für die Endeffektorposition** verarbeitet die Endeffektorposition. Dafür wird eine Input- und eine Fully-Connected-Ebene verwendet. Das Netzwerk besteht aus 2 Ebenen. Die erste Ebene ist eine Input-Ebene für die Endeffektorposition. Darauf folgt eine Fully-Connected-Ebene.

Das **Haupt-Netzwerk** verarbeitet die vereinigten Outputs der beiden Input-Netzwerke und bereitet sie für den Output vor. Dafür werden mehrere Fully-Connected-Ebenen und eine Flatten-Ebene verwendet. Das Netzwerk besteht aus 4 Ebenen. Die erste Ebene ist eine Vereinigung der Outputs der beiden Input-Netzwerke. Darauf folgen 2 Convolution-Ebenen und eine Flatten-Ebene

Das **Output-Netzwerk** definiert die Output-Ebene des Netzwerks. Es besteht aus einer Ebene. Diese ist eine Output-Ebene mit vier Outputs.

Für alle Fully-Connected- und Convolution-Ebenen wird die ReLu-Aktivierungsfunktion verwendet. In der Output-Ebenen wird der Tangens Hyperbolicus als Aktivierungsfunktion verwendet.

4.2.2. Critic-Netzwerk

Das Critic-Netzwerk besteht aus einem Inputnetzwerk für die Tiefenbilder, einem Input-Netzwerk für die Endeffektorposition, einem Input-Netzwerk für die Aktion, einem Hauptnetzwerk und dem Output-Netzwerk.

Dabei sind die Input-Netzwerke für die Tiefenbilder und Endeffektorposition und das Hauptnetzwerk identisch aufgebaut wie beim Actor-Netzwerk.

Das **Input-Netzwerk für die Aktion** verarbeitet die Variablen, die die Aktion beschreiben. Dafür werden eine Input- und eine Fully-Connected-Ebene verwendet. Das Netzwerk besteht aus 3 Ebenen. Die erste Ebene ist eine Input-Ebene für die Aktion. Darauf folgen zwei Fully-Connected-Ebenen.

Das **Output-Netzwerk** definiert die Output-Ebene des Netzwerks. Es besteht aus einer Ebene. Diese ist eine Output-Ebene mit einem Output-Neuron.

Für alle Fully-Connected- und Convolution-Ebenen wird die ReLu-Aktivierungsfunktion verwendet. In der Output-Ebene wird keine Aktivierungsfunktion verwendet.

4.2.3. Replay Buffer

Es wird ein Replay-Buffer verwendet. In diesem werden die Trainingssamples gespeichert und beim einem Anpassungsschritt der Gewichte als Minibatches abgerufen. Wenn die Kapazität des Replay-Buffers erreicht ist, wird das älteste Sample gelöscht. Die Kapazität des verwendeten Replay Buffers beträgt 100.000 Samples. Ein Minibatch besteht aus 16 Samples.

4.2.4. Target Netzwerke

Es werden zwei Target-Netzwerke (siehe 2.2.5) verwendet, welche Kopien des Actor- und des Critic-Netzwerks sind. Sie haben zu Beginn des Trainings die gleichen Gewichte. Während die Gewichte der Actor- und Critic-Netzwerke angepasst werden, werden die Gewichte der Target-Netzwerke nur langsam an diese angepasst. So entsteht eine zeitversetzte Version der Actor- und Critic-Netzwerke.

Diese werden zur Berechnung des Zielwerts in der Fehlerfunktion des Critic-Netzwerks verwendet. So hängen der Zielwert und der vom Critic bestimmte Wert nicht von den gleichen Parametern ab. Dies macht die Minimierung der Fehlerfunktion und somit das Training stabiler. Für die Anpassung der Target-Gewichte wird die Anpassungsrate τ von 0,001 für das Actor-Target-Netzwerk und das Critic-Target-Netzwerk gewählt (siehe 2.2.6).

5. Implementierung

Die Implementierung umfasst das Erstellen des Agenten, der Simulationsumgebung und der Realumgebung. Das System basiert auf ROS und ist demnach als System von Nodes implementiert, die über Topics und Services miteinander kommunizieren. Einen Überblick über alle Nodes bietet Abbildung 15.

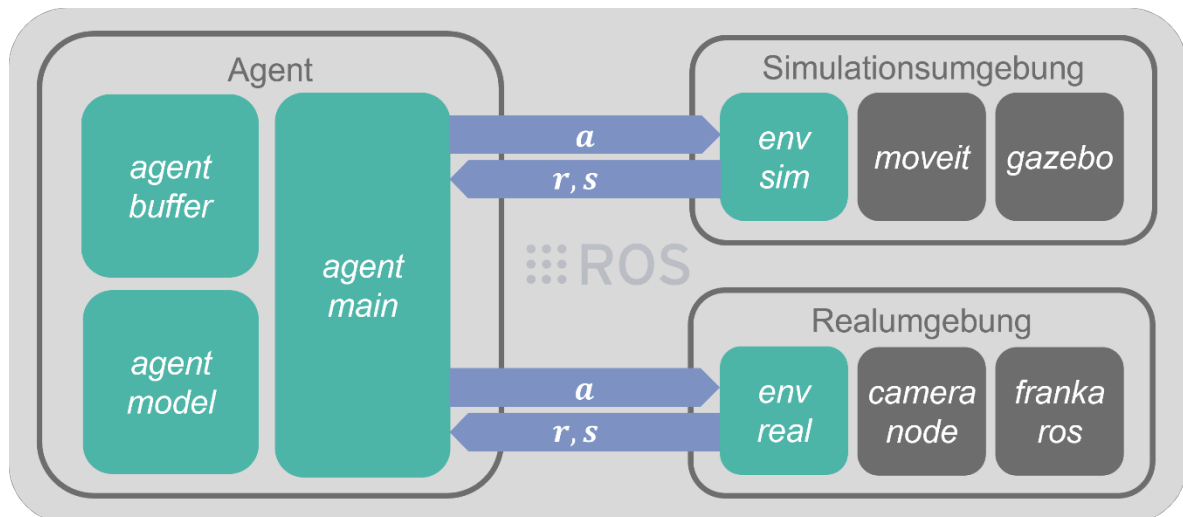


Abbildung 15: Überblick über die ROS-Nodes des Systems

5.1. Simulationsumgebung

Die Simulationsumgebung basiert auf der Simulationssoftware Gazebo in Verbindung mit der Bewegungsplanungssoftware MoveIt. Sie besteht aus der environment_sim-Node, der Gazebo-Node und der MoveIt-Node. Einen Überblick über die Nodes und wie diese interagieren bietet Abbildung 16.

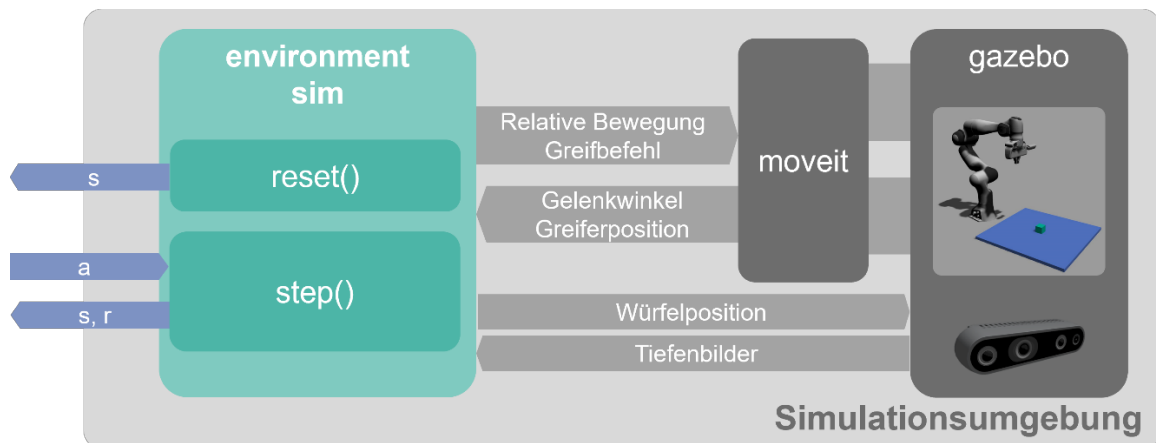


Abbildung 16: Überblick über die Simulationsumgebung

5.1.1. Konfiguration der Simulationsumgebung

Die Simulationsumgebung wurde mit der Simulationssoftware Gazebo implementiert. Diese ist mit dem Paket `gazebo_ros_pkgs` in ROS integriert. Zur Konfiguration der Simulationsumgebung müssen der

Panda Roboter, die Tiefenkamera, die Tischplatte und der Würfel in die Simulation geladen werden. Der Roboter muss ansteuerbar sein und die Kamerabilder müssen ausgelesen werden können.

Um den **Roboter in die Simulation zu laden**, benötigt man ein Modell des Roboters. Der Panda ist mit dem Paket `franka_ros` in ROS integriert. In diesem befindet sich ein geeignetes Modell des Roboters für die Simulation in Gazebo. Um den Roboter anzusteuern, wird die Bewegungsplanungssoftware MoveIt verwendet. Die dafür benötigten Konfigurationsdateien sind in dem Paket `panda_moveit_config` enthalten. Eine darin enthaltene ROS-Launch-Datei startet Gazebo und lädt das Robotermodell mit Steuerung in Gazebo. So kann der Roboter über die MoveIt-Schnittstellen angesteuert werden. Hierfür wurde die C++ Schnittstelle benutzt. Diese erlaubt es ebenfalls die Gelenkwinkel des Roboters und den Status des Greifers zu bestimmen.

Um die **Kamera in die Simulation zu laden**, wird ein Modell der Kamera benötigt. Ein geeignetes Modell der Intel RealSense D435 Tiefenkamera ist in der Gazebo Modell-Datenbank verfügbar. Zur Integration in Gazebo musste neben dem Modell auch ein Kamera-Plugin zum Modell des Roboters hinzugefügt werden. Dafür wird das `libgazebo_openni_kinect`-Plugin geladen und auf die Intel Realsense D435 angepasst. Des Weiteren werden dort die ROS-Topics definiert, zu denen die Kamerabilder veröffentlicht werden.

Die **Modelle des Würfels und der Tischplatte** bestehen jeweils aus einem Quader, welches als Modell in Gazebo erstellt werden kann. Beide Modelle sind in einer SDF-Datei beschrieben. Beide werden dann mithilfe des Topics `spawn_sdf` vom Paket `gazebo_ros` in die Simulation geladen.

Die gesamte Simulation ist in Abbildung 17 dargestellt.

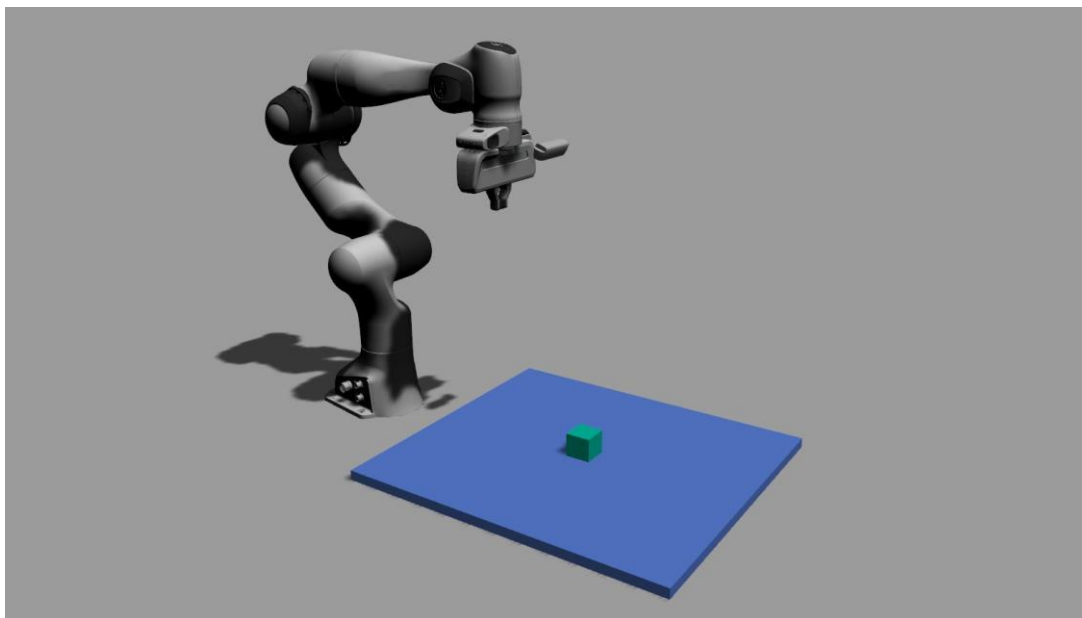


Abbildung 17: Simulationsumgebung mit Roboter, Kamera, Würfel und Tischplatte

5.1.2. Funktionen der Simulationsumgebung

Die Simulationsumgebung bietet zwei Funktionen: Das Zurücksetzen der Umgebung und das Ausführen einer Aktion in der Umgebung. Beide Funktionen sind als Service implementiert.

Beim Zurücksetzen der Umgebung wird der Endeffektor in seine Startpose bewegt, der Greifer geöffnet, der Würfel in seine Startpose gebracht und der Zustand der Umgebung bestimmt.

Die Bewegung zur Startpose geschieht über die MoveIt-Schnittstelle unter Angabe der gewünschten Pose des Endeffektors. MoveIt berechnet dann die genaue Trajektorie als lineare Bewegung. Die Startposition ist mittig über der Tischplatte. Der Greifer ist mit den Backen nach unten ausgerichtet. Die Backen bilden eine Linie die orthogonal zur Verbindungslinie des Endeffektormittelpunkts zur Roboterbasis verläuft.

Der Würfel wird über den von Gazebo angebotenen Service *gazebo/set_model_state* in seine Startpose gebracht. Die Startpose ist gemäß der Verteilung der Startzustände randomisiert auf der Tischplatte. Die maximale Abweichung von der Tischmitte beträgt 3cm in x- oder y-Richtung. Die Abweichung folgt einer Gleichverteilung.

Die Bestimmung des Zustands umfasst das Abspeichern der Tiefenbilder und dem Bestimmen der Gelenkwinkel und Endeffektorposition.

Die Tiefenbilder werden über ein Topic ausgelesen. Dafür wird das Paket *image_transport* verwendet. Dieses bietet optimierte Kommunikation mit Bildern über ROS. Es werden laufend die letzten drei Bilder in einem FIFO-Stack gespeichert. Zum Zeitpunkt des Zurücksetzens werden die aktuellen letzten drei Bilder mithilfe von OpenCV im PNG-Dateiformat abgespeichert. Die Dateipfade werden in der Zustands-Message gespeichert. Die Gelenkwinkel und Endeffektorposition werden mit der MoveIt-Schnittstelle ausgelesen. Diese werden ebenfalls der Zustands-ROS-Message hinzugefügt. Die Zustands-Message wird als Response des Service zurückgegeben. Der Request des Service ist leer.

Bei der Aktionsausführung in der Umgebung werden die relative Bewegung und der Greifbefehl ausgeführt und die Belohnung und der Zustand nach der Aktion bestimmt.

Die relative Bewegung geschieht über die MoveIt-Schnittstelle. Zunächst wird die aktuelle Pose des Endeffektors bestimmt. Die neue Pose wird mit dieser und der relativen Bewegung berechnet.

Der Greifbefehl wird mit dem *franka_gripper/grasp_action/goal*-Topic ausgeführt. Falls der Greifbefehl positiv ist, werden die Greifer-Finger geschlossen. Die Greiferweite beträgt nicht exakt 0cm, da dies selten zu Abstürzen in der Simulation geführt hat. Die Geschwindigkeit der Greifer-Finger ist 0,1 m/s und die Greifkraft beträgt 30N. Da der Greifbefehl manchmal nicht erfolgreich ausgeführt wurde, wird nach jedem Greifbefehl die Greiferweite überprüft. Stimmt diese nicht mit dem gewünschten Wert überein, wird der Greifbefehl wiederholt.

Die Belohnung wird nicht direkt als Wert bestimmt, sondern es wird zwischen verschiedenen Fällen unterschieden. Die Fälle sind mit diskreten Werten kodiert. Dies ermöglicht eine Wiederverwendung der Trainingssamples bei Anwendung von verschiedenen Fehlerfunktionen. Es gibt die vier Fälle „erfolgreicher Greifversuch“, „fehlgeschlagener Greifversuch“, „Fehler“ und „Standard“. Der Belohnungsfall abhängig von der Aktion unterschiedlich bestimmt. Falls der Greifbefehl ausgeführt

wurde, fährt der Roboter zurück in die Startposition. Dort wird erneut der Greifbefehl ausgeführt. Ist dann die Greiferweite ungleich der geschlossenen Greiferweite, wird dies als „erfolgreicher Greifversuch“ klassifiziert. Falls die Greiferweite gleich der geschlossenen Greiferweite ist, wird dies als „fehlgeschlagener Greifversuch“ klassifiziert. Falls kein Greifbefehl ausgeführt wurde, wird zunächst geprüft, ob MoveIt einen Fehler bei der Bewegungsausführung gemeldet hat oder ob sich der Endeffektor außerhalb der definierten Arbeitsraumgrenzen befindet. Dies wird als „Fehler“ klassifiziert. Sonstige Situationen werden als „Standard“ klassifiziert.

Der Zustand nach der Aktion wird so bestimmt, wie beim Zurücksetzen der Umgebung.

5.2. Agent

Der Agent besteht aus drei Nodes: Der *agent_main*, *agent_buffer* und *agent_model*. Die *agent_main*-Node regelt die Steuerung des Trainingsablaufs. Die *agent_buffer*-Node speichert die Trainingssamples im Replay Buffer und erstellt die Minibatches von Trainingssamples. Die *agent_model*-Node initialisiert die Actor- und Critic-Netzwerke und regelt die Anpassung der Gewichte. Einen detaillierten Überblick über die Nodes und die verwendeten Topics und Services bietet Abbildung 18.

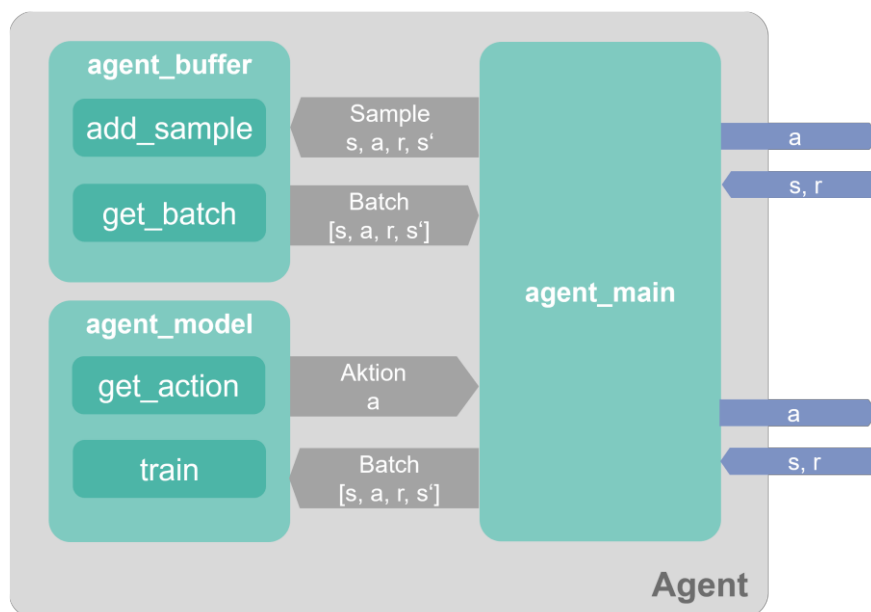


Abbildung 18: Überblick über die Nodes des Agenten

5.2.1. Konfiguration des Replay Buffers

Der Replay Buffer ist in der *agent_buffer*-Node implementiert. Sie speichert die Trainingssamples und erstellt die Minibatches von Trainingssamples.

Der Replay Buffer hat eine Kapazität von 100.000 Trainingssamples. Ein Sample besteht aus Zustand, Aktion, Belohnung und nächstem Zustand.

Beim Hinzufügen eines Sample wird eine Sample-Message über ein Topic veröffentlicht. Diese wird darauf in ein serialisierbares Objekt umgewandelt und dem Replay Buffer hinzugefügt. Dieser wird

regelmäßig als Boost-Archiv im Dateiformat .txt gespeichert, sodass er bei einem Systemneustart erhalten bleibt. Boost⁷ ist die Standardbibliothek für Serialisierung in C++. Für die Tiefenbilder des Zustands werden lediglich die Dateipfade gespeichert. Sie werden erst im Modell ausgelesen. Falls die Kapazität des Replay Buffers überschritten ist, werden die ältesten Samples überschrieben. Dies folgt somit dem FIFO-Prinzip.

Beim Erstellen eines Minibatch werden zufällig eine bestimmte Anzahl an Samples gezogen und als Batch zurückgegeben. Die Minibatch-Größe beträgt 16 Samples. Die Samples werden nach einer Gleichverteilung über die aktuelle Replay-Buffer-Größe gezogen.

5.2.2. Konfiguration des Modells

Das Modell muss während eines Trainingsschritts die Zielfunktionen für die Anpassung der Gewichte berechnen, den Gradient beider Funktionen bestimmen, ein Schritt des Gradientenabstiegsverfahren durchführen und die Gewichte der Target Netzwerke anpassen.

Der Actor nutzt als Zielfunktion für die Anpassung der Gewichte eine Verlustfunktion. Diese basiert auf der DDPG-Actor-Zielfunktion, ist jedoch negiert, da dann Gradientenabstiegsverfahren angewandt werden kann. Der Critic nutzt als Zielfunktion für die Anpassung der Gewichte die DDPG-Vorhersagefehlerfunktion. Diese ist wie folgt beschrieben.

Für die Berechnung der Gradienten beider Funktionen wird die Tensorflow-Funktion GradientTape verwendet.

Für den Schritt des Gradientenabstiegsverfahren wird der Adam-Optimizer verwendet. Dieser ist in Tensorflow implementiert. Der Actor hatte zunächst die Lernrate 10^{-4} . Diese wurde anhand des ursprünglichen DDPG-Algorithmus (Lillicrap et al., 2015) gewählt. Da der Actor auch nach der Änderung der Belohnungsfunktion weiterhin nach wenigen Hundert Lernschritten zu einem suboptimalen Punkt konvergierte, wurde die Lernrate des Actors verringert. Die Lernrate betrug dann 10^{-5} . Dies führte zu einem Lernerfolg. Die Lernrate des Critics beträgt 10^{-3} . Als Faktor zur Diskontierung des zukünftigen erwarteten Ertrags wurde 0.99 verwendet. Beide Werte entspringen dem ursprünglichen DDPG-Algorithmus. Der Faktor für die Anpassung der Target-Netzwerke beträgt 10^{-3} .

5.2.3. Konfiguration des Trainings

Das Training besteht aus 100.000 Episoden, welche jeweils aus maximal 25 Zeitschritten bestehen.

Ein Zeitschritt besteht aus dem Wahrnehmen des Zustands der Umgebung, der Wahl einer Aktion, dem Ausführen der Aktion, dem Wahrnehmen des nächsten Zustands der Umgebung und der Belohnung, dem speichern eines Trainingssamples im Replay Buffer, dem Ziehen eines Minibatches aus dem Replay Buffer und dem Anpassen der Gewichte der Netzwerke.

⁷ Quelle: <https://www.boost.org/>

Eine Episode des Trainings besteht aus dem Zurücksetzen der Umgebung, und dem mehrmaligen Wiederholen eines Zeitschrittes bis die Umgebung terminiert. Dies geschieht, wenn der Endeffektor aus dem Arbeitsraum fährt, einen Greifversuch unternimmt oder wenn 25 Zeitschritte unternommen wurden.

Das Training besteht aus 100.000 Episoden. Die ersten 50 Episoden wurden mit einer explizit programmierten Strategie π_{simple} ausgeführt. Diese wählt eine Aktion mit maximaler Bewegung in die negative z-Richtung und randomisierter Bewegung in x- und y-Richtung. Die Bewegung in x- und y-Richtung folgt einer Gleichverteilung in dem Intervall $[-0,6\text{cm}, 0,6\text{cm}]$. Danach folgt eine randomisierte Strategie π_{noise} . Diese Strategie wählt die Aktion, die durch den Actor bestimmt wird und fügt ein randomisiertes Rauschen zu dieser hinzu. Das Rauschen folgt einer Gleichverteilung in dem Intervall $[-1\text{cm}, 1\text{cm}]$. Dies soll die Erkundung des Zustandsraums verbessern.

Während des Trainings wurde oft ein Konvergieren zu einer suboptimalen Strategie beobachtet.

Während der ersten Trainingsdurchläufe kam es häufig dazu, dass der Actor in wenigen hundert Episoden zu einer Strategie konvergierte, bei der alle Aktionsvariablen einen Maximalwert annahmen. Diese Strategie führte zu einem geringen Ertrag, da der Endeffektor sich in wenigen Zeitschritten zu einer Arbeitsraumgrenze bewegte und damit die Episode terminierte. Auch nach mehreren weiteren Episoden veränderte sich die Strategie nicht.

Das Konvergieren zu einer suboptimalen Strategie könnte durch ein Verklemmungsmechanismus verursacht werden. Das beschriebene Phänomen von einer Konvergenz zu einer suboptimalen Strategie bei DDPG hat auch schon (Matheron et al., 2019) betrachtet. Dort wird festgestellt, dass in Umgebungen mit seltenen Belohnungen der Actor zu einer suboptimalen Strategie konvergiert. Außerdem wird beschrieben, wie eine weitere Veränderung des Actors verhindert werden könnte, selbst wenn er eine belohnte Aktion wahrnimmt. Die beschriebene Verklemmung wird wie folgt beschrieben. Zunächst findet der Actor keinen oder selten eine Belohnung. Danach stabilisiert er sich zu einer suboptimalen Strategie. Dies ist der Einstieg in einen Kreislauf. Der Critic tendiert bei der Anpassung seiner Strategie daraufhin nicht den erwarteten zukünftigen Ertrag der optimalen Strategie zu approximieren, sondern den erwarteten zukünftigen Ertrag der suboptimalen Strategie. Der erwartete zukünftige Ertrag der suboptimalen Strategie ist stückweise konstant, da der zukünftige Ertrag einer suboptimalen Strategie meistens Null ist. Da der Critic diese Funktion approximiert ist dieser auch stückweise konstant. Somit ist der Gradient des Critic meistens Null. Da die Gewichte des Actors anhand des Gradienten des Critic angepasst werden, gibt es bei einem Gradienten von Null auch keine Anpassung des Actors. Dies ist wiederum der Einstieg in den Kreislauf. So kommt es zu einer Verklemmung, in der die mangelnde Anpassung des Actors dazu führt, dass der Critic stückweise konstant wird und damit der Actor wiederum nicht angepasst wird.

Es gibt verschiedene Lösungsansätze, um die Verklemmung zu umgehen. X stellen verschiedene Lösungsansätze vor. Einer davon ist, die Belohnung weniger selten zu machen. Dafür wurde die Belohnung abhängig von der Endeffektorhöhe hinzugefügt. Somit können Belohnungen öfter gefunden

werden und der Actor hat weniger Zeit, sich bei einer suboptimalen Strategie zu stabilisieren. Des Weiteren wurde dazu die einfache Strategie μ_{simple} eingeführt.

5.3. Realumgebung

Die Realumgebung ist implementiert mit dem Franka Emika Panda Roboter und der Intel RealSense D435 Tiefenkamera in Verbindung mit der Bewegungsplanungssoftware MoveIt. Sie muss Funktionen zum Zurücksetzen der Umgebung und zum Ausführen einer Aktion bieten. Sie besteht aus der *environment_real*-Node, der *MoveIt*-Node, der Node für den realen Roboter und der *realsense_camera*-Node. Einen Überblick über die Realumgebung bietet Abbildung 19.

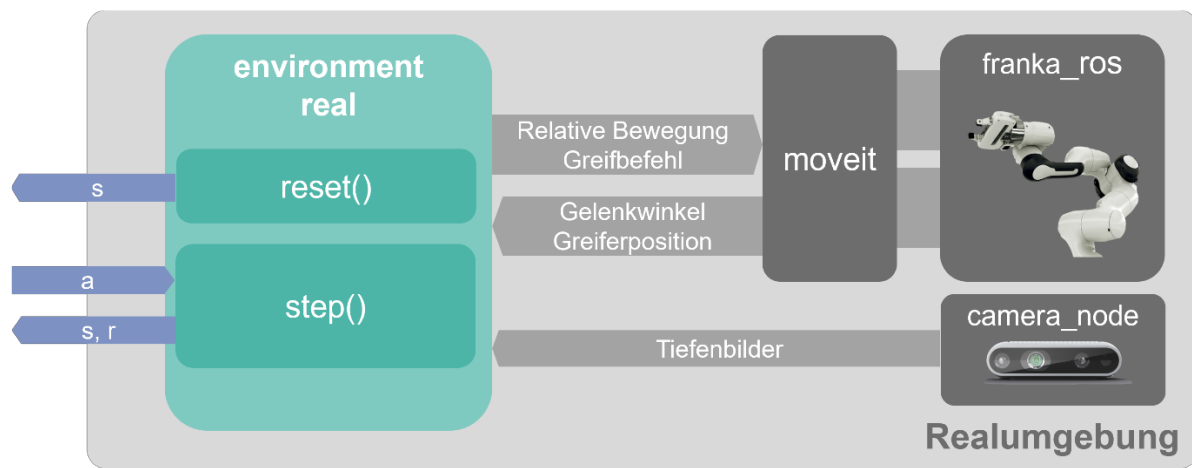


Abbildung 19: Überblick über die Realumgebung

5.3.1. Konfiguration der Realumgebung

Die Realumgebung wurde mit dem Franka Emika Panda Roboter und der Intel RealSense D435 RGBD-Kamera implementiert. Zur Konfiguration der Realumgebung muss der Roboter ansteuerbar sein und die Kamerabilder müssen ausgelesen werden können.

Der Panda ist mit dem *franka_ros*-Paket in ROS integriert. Um den Roboter anzusteuern, wird die Bewegungsplanungssoftware MoveIt verwendet. Die dafür benötigten Konfigurationsdateien sind in dem Paket *panda_moveit_config* enthalten. Eine darin enthaltene ROS-Launch-Datei startet die für die Steuerung mit MoveIt benötigten ROS-Nodes. So kann der Roboter über die MoveIt-Schnittstellen angesteuert werden. Hierfür wurde die C++ Schnittstelle benutzt. Diese erlaubt es ebenfalls die Gelenkwinkel des Roboters und den Status des Greifers zu bestimmen.

Die Intel RealSense D435 ist über das Paket *realsense2_camera* in ROS integriert. Dabei werden die Kamerabilder über ein ROS-Topic zur Verfügung gestellt. Des Weiteren muss die Frequenz der Kamerabilder mit der Frequenz der Kamerabilder in der Simulation übereinstimmen. Um die Frequenz von 30 FPS auf 15 FPS abzusenken, wurde jeder zweite Nachricht von dem Kamera-ROS-Topic nicht verarbeitet. Die Kamera hat eine minimale Distanz, ab derer diese die Distanz wahrnimmt. Die Position

der Kamera wurde so gewählt, dass diese trotz dieser minimalen Distanz den Würfel auch noch wahrnehmen kann, wenn dieser im Greifer ist.

Der Würfel hat dieselbe Größe wie der simulierte Würfel und besteht aus Plastik.

5.3.2. Funktionen der Realumgebung

Die Realumgebung bietet wie die Simulationsumgebung zwei Funktionen: Das Zurücksetzen der Umgebung und das Ausführen einer Aktion in der Umgebung. Beide Funktionen sind als ROS-Service implementiert.

Beim Zurücksetzen der Umgebung wird der Endeffektor in seine Startpose bewegt, der Greifer geöffnet und der Zustand der Umgebung bestimmt.

Die Bewegung zur Startpose geschieht wie in der Simulationsumgebung über die MoveIt-Schnittstelle unter Angabe der gewünschten Pose des Endeffektors. Die Startposition ist ebenfalls äquivalent zur Simulationsumgebung.

Der Würfel wird am Anfang einer Episode per Hand platziert. Dabei wird die gleiche maximale Abweichung verwendet wie in der Simulation. Durch die manuelle Platzierung wird die Verteilung der Würfelposition bestimmt. Beim Testen sollte man darauf achten, dass diese einer Gleichverteilung folgt.

Die Bestimmung des Zustands geschieht wie in der Simulationsumgebung.

Bei der Aktionsausführung in der Umgebung werden die relative Bewegung und der Greifbefehl ausgeführt und die Belohnung und der Zustand nach der Aktion bestimmt.

Dies geschieht grundsätzlich wie in der Simulationsumgebung.

Der Greifbefehl wird in der Realumgebung allerdings mit dem *franka_gripper/gripper_action/goal*-Topic ausgeführt. Die Greiferweite beträgt im geschlossenen Zustand 0cm. Die Geschwindigkeit der Finger ist 0.1 m/s und die Greifkraft beträgt 25N.

6. Evaluation

Das Ziel dieser Arbeit war ein System zu implementieren, welches einem Roboter mit am Endeffektor montierter Tiefenkamera ermöglicht, das Greifen von Objekten zu lernen. Das System sollte dann auch in einer Realumgebung implementiert werden. Die Umgebung wurde in der Simulation und in der Realität als Markov-Entscheidungsprozess implementiert. Es wurde der Algorithmus DDPG für das Lernen angewandt. Während des Trainings zeigten sich Lernerfolge, aber auch Instabilität des Trainings.

6.1. Ergebnis

Das Training in der Simulation ist instabil, aber führt zu einem Lernerfolg. Betrachtet wurden der durchschnittliche Ertrag der letzten 50 Episoden und der relative Greiferfolg der letzten 50 Episoden. Beim Training konnte ein Lernerfolg festgestellt werden, wie der Anstieg im durchschnittlichen Ertrag der letzten 50 Episoden in Abbildung 20 bis ca. 50 Tausend Episoden zeigt. Allerdings kann man danach ein kurzes Einbrechen des durchschnittlichen Ertrags bis ungefähr zum sechzigtausendsten Batch-Update beobachten. Nach einer kurzen Phase des weiteren Anstiegs kommt eine weitere längere Phase, wo der Durchschnittliche Ertrag der letzten 50 Episoden wieder abfällt bis zum neunzigtausendsten Batch Update. Dabei ist zu beobachten, dass der Endeffektor die Rotation um die z-Achse erhöht, welches zu weniger Ertrag führt, da der rotierte Greifer öfter an die Würfelkanten stößt und so die Bewegung abbrechen könnte beim Greifversuch den Würfel verschieben könnte, sodass dieser nicht mehr zwischen den Fingern platziert ist. Nach Neunzigtausend Batch-Updates steigt der durchschnittliche Ertrag der letzten 50 Episoden wieder, bleibt aber unter dem Niveau des ersten Anstiegs.

Dies zeigt, dass das Training mit DDPG sehr instabil ist. Dies ist auf das Zusammenspiel von Actor und Critic zurückzuführen. Dies haben Fujimoto et al. (2018) beschrieben. Der erwartete zukünftige Ertrag wird bei einer suboptimalen Strategie überbewertet. Dieser überbewertete erwartete zukünftige Ertrag führt beim Actor-Update zu einer suboptimalen Strategie.



Abbildung 20: Durchschnittlicher Ertrag der letzten 50 Episoden im Trainingsverlauf

Beim relativen Greiferfolg (Abbildung X) zeigt sich ein äquivalenter Verlauf. Das Maximum des Relativen Greiferfolgs der letzten 50 Episoden liegt bei 48,0%. Betrachtet man die letzten 100 Episoden ergibt sich ein relativer Greiferfolg von 45,55%. Letzterer könnte weniger aussagekräftig sein, da die Strategie auf einer Spanne von 1000 Batch-Updates stärker variiert als die der letzten 500.

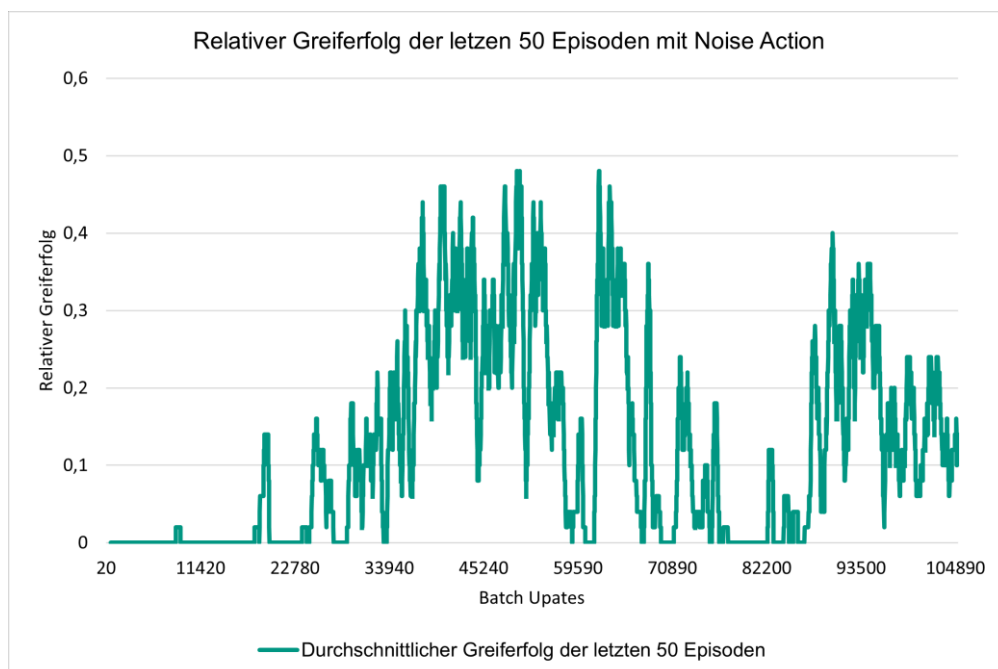


Abbildung 21: Relativer Greiferfolg der letzten 50 Episoden im Trainingsverlauf

Beim Testen in der Simulationsumgebung erreicht das Modell einen schlechteren Greiferfolg als während des Trainings. Zum Testen in der Simulationsumgebung wurde ein Modell nach 94130 Batch-Updates gewählt. Zu diesem Zeitpunkt während des Trainings hatte das Modell einen

durchschnittlichen Ertrag der letzten 50 Episoden von 0,43 und einen durchschnittlichen Greiferfolg von 28%. Das Modell wurde nun ohne Rauschen getestet. Nach 426 Episoden konnte ein durchschnittlicher Ertrag der letzten 50 Episoden von 0,3143 und ein durchschnittlicher Greiferfolg von 16% beobachtet werden. Somit war die Leistung des Modells während des Trainings zu hoch eingeschätzt worden. Gleichzeitig konnte beobachtet werden, dass der Endeffektor sich immer zu einer ähnlichen Position bewegt, unabhängig von der Position des Würfels. Dies könnte darauf hindeuten, dass das Modell lediglich anhand der Endeffektorposition gelernt hat, in die Mitte der Tischplatte zu fahren. Die Tiefenbilder könnten nur einen geringen oder keinen Einfluss auf die Wahl der Aktion haben. Dies könnte durch weiteres Training verbessert werden. Womöglich werden die Eigenschaften der Tiefenbilder nicht ausreichend wahrgenommen. Dies könnte durch eine geänderte Netzwerkstruktur verbessert werden.

Beim Testen in der Realumgebung gibt es kaum eine Reaktion auf die Tiefenbilder. In der Realumgebung wurde dasselbe Modell nun getestet. Auch hier sieht man, dass das Modell Aktionen wählt, die den Endeffektor immer zu einer ähnlichen Position über dem Tisch bewegt. Nur wenn der Würfel sich zufällig auch in dieser Position befindet, ist das Greifen erfolgreich.

Es ist fraglich, ob das gelernte Modell anhand der Tiefenbilder gelernt hat oder nur anhand der Endeffektorposition. Wenn eine Gleichverteilung der Würfelposition wie in der Simulation angenommen wird, ist die Fläche, auf der der Würfelmittelpunkt gleichverteilt positioniert ist das Quadrat der Größe des Intervalls der Gleichverteilung (6cm) und beträgt somit 36 cm^2 . Wenn die Endeffektorposition beim Greifen unabhängig von der Würfelposition unverändert innerhalb dieser Fläche wäre, ergibt sich die vom Greifer abgedeckte Fläche wie folgt. Die Greiferweite von 8cm führt auf der Verbindungslinie der Greifer-Finger zu einer maximalen Abweichung von 1,5cm in beide Richtungen für die Position des Würfelmittelpunkts. Orthogonal zur Verbindungslinie der Finger kann die Position des Würfelmittelpunkts die halbe Seitenlänge des Würfels in beide Richtungen abweichen und dennoch erfolgreich Greifen. Die vom Greifer abgedeckte Fläche beträgt somit 15 cm^2 . Bei einer Gleichverteilung und einer festen Greifposition innerhalb der Verteilung der Position des Würfelmittelpunkts ergibt sich ein theoretischer durchschnittlicher Greiferfolg von 41,67%. Bei einem Versuch, bei dem der Endeffektor sich lediglich maximal in die negative z-Richtung bewegt, wird über 790 Episoden ein durchschnittlicher Greiferfolg von 46,46% beobachtet. Dies deckt sich mit den theoretischen Berechnungen. Der leicht höhere Wert könnte dadurch entstehen, dass der simulierte Greifer eine Nachgiebigkeit aufweist. Dies führt dazu, dass bei einem Greifversuch, bei dem der Greifer von oben auf den Würfel stößt, der Würfel manchmal in eine bessere Lage für das Greifen verschoben wird. In der Realität führt dies zum Abbruch der Trajektorienausführung, da die maximalen Kräfte überschritten werden. So könnte die Abweichung entlang der Verbindungslinie der Greifer-Finger größer sein als angenommen. Womöglich reicht auch nur ein geringer Kontakt mit dem Würfel um ihn zu greifen, sodass auch die Abweichung orthogonal zur Verbindungslinie der Greifer-Finger etwas größer als angenommen sein könnte. In Anbetracht des durchschnittlichen Greiferfolgs der einfachen

Strategie muss in Frage gestellt werden, ob das Modell anhand der Tiefenbilder gelernt hat. Der maximale durchschnittliche Greiferfolg während des Trainings von 48% könnte bedeuten, dass der Greifer lediglich zu einer Position innerhalb der Verteilung der Würfelposition bewegt wurde. Zufällig kam es dann zum Greiferfolg.

Der Unterschied zwischen Simulationsumgebung und Realumgebung könnte die mangelnde Beachtung der Tiefenbilder verstärken. Der Agent könnte in der Evaluation weniger auf die Kamerabilder reagieren als in der Simulation, da die simulierten Tiefenbilder von den realen Tiefenbildern verschieden sind. Man nennt diesen Unterschied die Reality-Gap (Peng et al., 2018). Dieses Phänomen beschreibt, dass Reinforcement Learning Agenten, die in der Simulation trainiert wurden, weniger erfolgreich in der realen Umgebung sind.

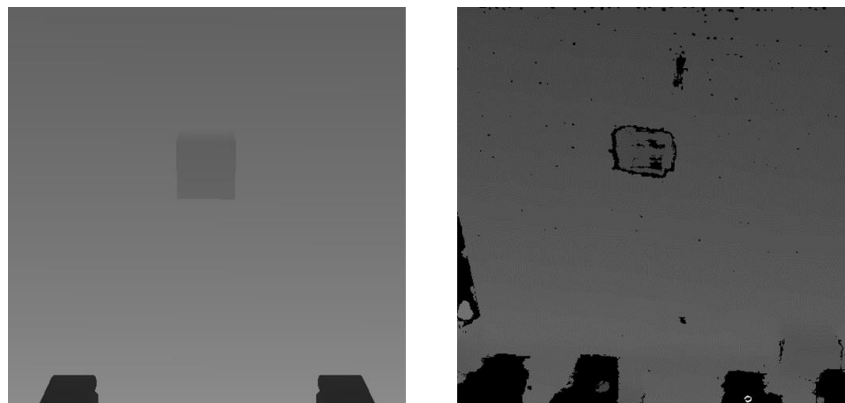


Abbildung 22: Unterschied der Tiefenbilder zwischen Simulations-(links) und Realumgebung(rechts)

Der Unterschied zwischen der hier verwendeten Simulationsumgebung und Realumgebung basiert auf den Tiefenbildern und der Kamera- und Tischposition. Die Tiefenbilder aus der Realumgebung weisen einen geringeren Kontrast und Helligkeit als die der Simulationsumgebung (Abbildung 22). Außerdem kommt es in der Realumgebung zu Rauscherscheinungen die sich als schwarze Punkte ausdrücken. An Kanten wird ebenfalls ein schwarzes Rauschen wahrgenommen. Nahe Objekte werden außerdem zweimal dargestellt, wie man an den Greifer-Finger am unteren Rand der Tiefenbilder sieht. Die Tischposition trägt ebenfalls zum Reality-Gap bei. Die Tischposition ist in der Realumgebung gering niedriger im Verhältnis zur Roboterposition. Somit ist das Tiefenbild im Startzustand unterschiedlich. Wenn das Modell in der Simulation nicht mit diesem Zustand konfrontiert war, könnte dies den Erfolg des Modells in der Realität beeinflussen. Die Kameraposition könnte ebenfalls zur Reality-Gap beitragen. Die Kameraposition in der Realität könnte leicht unterschiedlich zu der in der Simulation sein. Somit würde der gleiche Zustand unterschiedliche Tiefenbilder erzeugen.

Um die Reality-Gap zu überbrücken, werden Domain Adaptation und Domain Randomisation angewandt. Bei Domain Adaption wird die Realumgebung an die Simulationsumgebung angepasst. Man könnte die Tischposition auf gleicher Höhe wie in der Simulation montieren. Man könnte auch die Startposition des Endeffektors randomisieren, was einen ähnlichen Effekt haben sollte. Ebenfalls die Kameraposition könnte randomisiert werden, sodass leichte Verschiebungen in der Realumgebung weniger Auswirkung auf die Realumgebung haben. Außerdem könnte man die wahrgenommenen

Tiefenbilder in der Realität hinsichtlich Helligkeit und Kontrast verändern, sodass diese den simulierten Tiefenbildern ähnlicher sind.

6.2. Einschränkungen

Das Modell hat gelernt sich zu der Position zu bewegen, wo die Würfelposition sein könnte. Allerdings hat das Modell womöglich nicht die Position des Würfels anhand der Tiefenbilder erkannt.

Das Modell wurde bisher nur mit einem Objekt trainiert, der Greiferfolg auf andersartigen Objekten sollte demnach gering ausfallen.

Die Würfelposition wurde zur Vereinfachung nur gering randomisiert. Somit sollte der Greiferfolg bei einer Position außerhalb der Verteilung gering ausfallen.

Durch die Beschränkung des Arbeitsraums ist gewährleistet, dass der Endeffektor sich nicht in eine Position bewegt, die ein Sicherheitsrisiko bedeutet. Allerdings wird die genaue Gelenkwinkeltrajektorie von der Bewegungsplanungssoftware bestimmt. Somit könnten unerwartete Gelenkwinkelbewegungen ausgeführt werden, vor allem in einer Konfiguration, in der Gelenkwinkelgrenzen fast erreicht sind. Auch das Fallenlassen des Würfels in der Realumgebung könnte ein Sicherheitsrisiko darstellen.

7. Fazit

7.1. Zusammenfassung

In dieser Arbeit sollte ein Software-System entwickelt und implementiert werden, das einem Roboterarm mit sieben Freiheitsgraden und einer am Endeffektor montierten Tiefenkamera ermöglicht, zu lernen ein bestimmtes Objekt auf einer Tischplatte zu greifen. Dafür sollte ein Modell mithilfe des Deep Reinforcement Learning Algorithmus Deep Deterministic Policy Gradient trainiert.

Für das Training wurde eine Simulationsumgebung implementiert. Diese basiert auf der Simulationssoftware Gazebo und der Bewegungsplanungssoftware MoveIt. Beim Training konnte beobachtet werden, dass der Deep Deterministic Policy Gradient-Algorithmus häufig in eine Verklemmung gerät, die weiteres Lernen verhindert. Diese kann durch Anpassung der Umgebung des Agenten vermieden werden. Außerdem konnte festgestellt werden, dass das Training instabil ist. Dies liegt an der Actor-Critic-Struktur des DDPG-Algorithmus. Um dies zu umgehen, ist das Finden der korrekten Hyperparameter entscheidend. Des Weiteren benötigt es eine große Anzahl an Trainingsepisoden, sodass das Modell anhand von Tiefenbildern lernt. Die am Endeffektor montierte Tiefenkamera macht das Problem komplexer, indem eine größere Variabilität an Inputdaten vorkommt. Die beim Training generierten Daten bilden einen Datensatz, der einfach auf weitere Reinforcement Learning Algorithmen angewendet werden kann. Dies könnte in weiteren Arbeiten fortgeführt werden. Zur Evaluation des Modells wurde eine Schnittstelle zur realen Hardware implementiert. Die Integration der realen Hardware konnte nahezu äquivalent zur Integration der Simulation geschehen. Nur wenige Anpassungen mussten unternommen werden. Dies erleichtert die Implementierung des Transfers von Simulation zu Realität. Bei der Evaluation ist außerdem festzustellen, dass der Unterschied zwischen Simulations- und Realumgebung eine verschlechterte Leistung des Modells verursacht. Hier ist ein Simulations-zu-Realitäts-Transfer -Ansatz nötig. Ohne einen solchen Ansatz können gelernte Modelle nicht auf das Problem in der Realität angewandt werden.

7.2. Ausblick

Das entwickelte System bildet eine Grundlage zum Lernen des Greifens mit Deep Reinforcement Learning. Es kann in folgenden Bereichen erweitert werden, um das Problem des Greifens eines Objektes verlässlich zu lernen.

Die Performance des Trainings könnte verbessert werden. Zunächst sollten weitere Untersuchungen zur Wahl der Hyperparameter unternommen werden. Die Hyperparameter könnten einen großen Einfluss auf den Lernerfolg haben. Des Weiteren sollte das Training über mehr Episoden weitergeführt werden. Dies gibt Aufschluss darüber, ob das System auch von den Tiefenbildern lernen kann, und ob sich das Lernen stabilisiert. Außerdem könnten stabilere Deep Reinforcement Learning-Ansätze wie zum Beispiel QT-Opt implementiert werden. Diese könnten mit dem erstellten Datensatz geschehen.

Das DRL-Problem könnte erweitert werden durch eine variabelere Umgebung und einem Ansatz zum Simulations-zu-Realitäts-Transfer. Die maximale zufällige Abweichung der Würfelposition von der Tischmitte könnte erhöht werden. Dies würde dem System erlauben eine allgemeinere Aufgabe zu lösen. Eine weitere Maßnahme zur Verallgemeinerung des Problems wäre, verschiedene Objekte während des Trainings zu verwenden. Um die Leistung des Systems in der Realumgebung zu erhöhen, sollte ein Simulations-zu-Realitäts-Transfer-Ansatz angewendet werden. Dafür würde sich Domain Randomisation eignen. Beispielsweise könnte die Helligkeit und der Kontrast der Tiefenbilder beim Training randomisiert werden.

Die Sicherheit des Systems könnte verbessert werden. Um das Sicherheitsrisiko während der Evaluation in der Realumgebung zu reduzieren, sollte der Arbeitsraum der Bewegungsplanungssoftware begrenzt werden. Außerdem könnte der Roboter nach einem erfolgreichen Greifen das Objekt automatisch in eine zufällige Startposition bewegen, sodass kein herunterfallendes Objekt ein Sicherheitsrisiko darstellt.

Das System könnte umfänglicher getestet werden. Man könnte das Objekt in eine dem Modell unbekannte Startposition bringen, es während der Ausführung verschieben oder ein dem Modell unbekannten Objekt verwenden. So könnten die Grenzen des Systems besser beschrieben werden.

Literaturverzeichnis

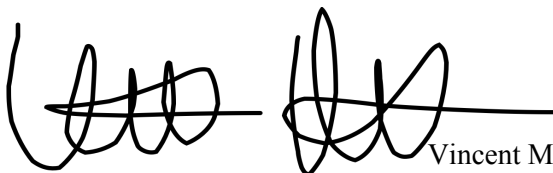
- Amirsina Torfi, R. A. S., Yaser Keneshloo, Nader Tavaf, Edward A. Fox. (2020). Natural Language Processing Advancements By Deep Learning: A Survey.
- Bellman, R. (1957). A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5), 679-684. <http://www.jstor.org/stable/24900506>
- Bousmalis, K., Irpan, A., Wohlhart, P., Bai, Y., Kelcey, M., Kalakrishnan, M., Downs, L., Ibarz, J., Pastor, P., Konolige, K., Levine, S., & Vanhoucke, V. (2017, 2018). Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping.
- Chen, B., Su, J., Wang, L., & Gu, Q. (2021). Learning Robot Grasping from a Random Pile with Deep Q-Learning. In (pp. 142-152). Springer International Publishing. https://doi.org/10.1007/978-3-030-89098-8_14
- Christopher Berner, G. B., Brooke Chan, Vicki Cheung, Przemysław “Psyho” Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, Susan Zhang. (2019). Dota 2 with Large Scale Deep Reinforcement Learning. <https://doi.org/10.48550/arxiv.1912.06680>
- Coleman, D., Sucan, I., Chitta, S., & Correll, N. (2014). Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study.
- El-Shamouty, M., Kleeberger, K., Lämmle, A., & Huber, M. (2019). Simulation-driven machine learning for robotics and automation.
- Finch, S. R. (2004). Ornstein-Uhlenbeck Process.
- Fujimoto, S., Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. International conference on machine learning,
- Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR.
- Hodson, R. (2018). A gripping problem: designing machines that can grasp and manipulate objects with anything approaching human levels of dexterity is first on the to-do list for robotics. *Nature*.
- James, S., Andrew, & Johns, E. (2017). Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task. *arXiv pre-print server*. <https://arxiv.org/abs/1707.02267v2>
- Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., & Levine, S. (2018). QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. *arXiv pre-print server*. <https://doi.org/None> arxiv:1806.10293
- Kleeberger, K., Bormann, R., Kraus, W., & Huber, M. F. (2020). A Survey on Learning-Based Robotic Grasping. *Current Robotics Reports*, 1(4), 239-249. <https://doi.org/10.1007/s43154-020-00021-6>
- Koenig, N., & Howard, A. (2004, 28 Sept.-2 Oct. 2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566),
- Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324. <https://doi.org/10.1109/5.726791>
- Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2019). Continuous control with deep reinforcement learning. *arXiv pre-print server*. <https://arxiv.org/abs/1509.02971>
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.
- Matheron, G., Perrin, N., & Sigaud, O. (2019). The problem with DDPG: understanding failures in deterministic environments with sparse rewards. *arXiv preprint arXiv:1911.11679*.
- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115-133. <https://doi.org/10.1007/BF02478259>
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
- Moore, A. W. (1990). Efficient memory-based learning for robot control.

- Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018, 2018). Sim-to-Real Transfer of Robotic Control with Dynamics Randomization.
- Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., Lampe, T., Tassa, Y., Erez, T., & Riedmiller, M. (2017). Data-efficient Deep Reinforcement Learning for Dexterous Manipulation. *ArXiv, abs/1704.03073*.
- Quigley, M. (2009). ROS: an open-source Robot Operating System. ICRA 2009,
- Reinhart, G., Hüttner, S., & Krug, S. (2011). Automatic Configuration of Robot Systems - Upward and Downward Integration. In (pp. 102-111). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-25486-4_11
- Rosenblatt, F. (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. <https://doi.org/10.1038/323533a0>
- Shi, B., Bai, X., & Yao, C. (2017). An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(11), 2298-2304. <https://doi.org/10.1109/tpami.2016.2646371>
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484-489. <https://doi.org/10.1038/nature16961>
- Song, S., Zeng, A., Lee, J., & Funkhouser, T. (2020). Grasping in the Wild: Learning 6DoF Closed-Loop Grasping from Low-Cost Demonstrations. *arXiv pre-print server*. <https://doi.org/None> arxiv:1912.04344
- Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). *Policy gradient methods for reinforcement learning with function approximation* Proceedings of the 12th International Conference on Neural Information Processing Systems, Denver, CO.
- Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *arXiv pre-print server*. <https://arxiv.org/abs/1703.06907>
- Watkins, C. (1989). Learning From Delayed Rewards.
- Weibel, J.-B., Patten, T., & Vincze, M. (2019). Addressing the Sim2Real Gap in Robotic 3D Object Classification. *arXiv pre-print server*. <https://doi.org/None> arxiv:1910.12585

Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 30. Juni 2022



Vincent Müller