Technische Hochschule Brandenburg
University of Applied Sciences

Fachbereich Informatik und Medien

MASTERARBEIT

Kombination von Imitation Learning und Reinforcement Learning zur Bewegungssteuerung

Vorgelegt von: Darya Martyniuk

am: 19.12.2019

zum
Erlangen des akademischen Grades

MASTER OF SCIENCE
(M.Sc.)

Erstbetreuer: Dipl.-Inform. Ingo Boersch

Zweitbetreuer: Prof. Dr.-Ing. Jochen Heinsohn

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Kombination von Imitation Learning und Reinforcement Learning zur Bewegungssteuerung

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.
In Bezug auf die sprachliche Gestaltung wurde ich von Fr. Handschuck beraten, dabei erfolgten keinerlei inhaltliche Veränderungen.

Brandenburg an der Havel, den 19.12.2019

Unterschrift

Danksagung

Ich möchte mich recht herzlich bei Herrn Dipl.-Inform. Ingo Boersch und Herrn Prof. Dr.-Ing. Jochen Heinsohn für die Betreuung der vorliegenden Arbeit sowie der zahlreichen Projektarbeiten während des Studiums bedanken. Sie haben mich stets unterstützt und mit ihrem Wissen, den Kompetenzen und spannenden Lehrveranstaltungen motiviert.
Für die sprachliche Konsultation und Unterstützung möchte ich Frau Gerlinde Handschuck danken, die mir viel über das wissenschaftliche Schreiben in der deutschen Sprache beigebracht hat.
Weiterhin danke ich IHK Potsdam für die Förderung im Rahmen des Deutschlandstipendiums.
An dieser Stelle danke ich auch den Kolleginnen und Kollegen am Fraunhofer FOKUS, insbesondere dem Herrn Dr.-Ing. Andreas Billig, für die motivierenden Worte und fachlichen Diskussionen.
Ein besonderer Dank gilt meinem Freund Pascal Mauritz für die Unterstützung während des gesamten Studiums. Ebenfalls danke ich meine Familie und Freunde, die trotz der Entfernung immer für mich da sind.

Zusammenfassung

Eine erfolgreiche Kombination von Imitation Learning (IL) und Reinforcement Learning (RL) zur Bewegungssteuerung eines Roboters besitzt das Potenzial, einem Endnutzer ohne Programmierkenntnisse einen intelligenten Roboter zu Verfügung zu stellen, der in der Lage ist, die benötigten motorischen Fähigkeiten von den Menschen zu erlernen und sie angesichts der aktuellen Rahmenbedingungen und Ziele eigenständig anzupassen. In dieser Masterarbeit wird eine Kombination von IL und RL zur Bewegungssteuerung des humanoiden Roboter NAO eingesetzt. Der Lernprozess findet auf dem realen Roboter ohne das vorherige Training in einer Simulation statt. Die Grundlage für das Lernen stellen kinästhetische Demonstrationen eines Experten sowie die eigene Erfahrung des Agenten, die er durch die Interaktion mit der Umgebung sammelt. Der verwendete Lernverfahren basiert auf den Algorithmen Deep Deterministic Policy Gradient from Demonstration(DDPGfD) und Twin Delayed Policy Gradient (TD3) und wird in einer Fallstudie, dem Spiel Ball-in-a-Cup, evaluiert. Die Ergebnisse zeigen, dass der umgesetzte Algorithmus ein effizientes Lernen ermöglicht. Vortrainiert mit den Daten aus Demonstrationen, fängt der Roboter die Interaktion mit der Umgebung mit einer suboptimalen Strategie an, die er im Laufe des Trainings schnell verbessert. Die Leistung des Algorithmus ist jedoch stark von der Konfiguration der Hyperparameter abhängig. In zukünftigen Arbeiten soll für das Ball-in-a-Cup-Spiel eine Simulation erstellt werden, in der die Hyperparameter und die möglichen Verbesserungen des Lernverfahrens vor dem Training mit dem realen Roboter evaluiert werden können.

Abstract

A successful combination of imitation learning (IL) and reinforcement learning (RL) for robotic motion control has the potential to provide a robot with the ability to learn required motor skills from humans and adapt them autonomously according to the provided goals and the current situation returned by the environment. A combination of IL and RL is used in this master thesis to control the motion of the humanoid robot NAO. A training process is performed on the real robot without previous training in a simulation. The knowledge for learning is provided by the expert's kinaesthetic demonstrations and the agent's own experience collected through interaction with the environment. The applied learning method is based on the following algorithms: Deep Deterministic Policy Gradient from Demonstration (DDPGfD) and Twin Delayed Policy Gradient (TD3). The case study „Ball-in-a-Cup“ is used for the evaluation. Results show that the implemented method enables a sample-efficient learning. Pre-trained with demonstrations, the robot starts interacting with the environment using a suboptimal policy, which can be quickly improved during training. However, the performance of the algorithm depends on the configuration of the hyperparameters. In future work, a simulation of the Ball-in-a-Cup should be created. This provides the possibility to evaluate the hyperparameters and investigate possible improvements of the learning method.

Inhaltsverzeichnis

Abkürzungsverzeichnis

DDPG	Deep Deterministic Policy Gradient
DDPGfD	Deep Deterministic Policy Gradient from Demonstrations
DoF	Degree of Freedom
DPG	Deterministic Policy Gradient
DQN	Deep Q-Learning
IL	Imitation Learning
IS	Importance Sampling
LfD	Learning from Demonstrations
MDP	Markov Decision Process
ML	Machine Learning
PER	Prioritized Experience Replay
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SPG	Stochastic Policy Gradient
TD	Temporal Difference
TD3	Twin Delayed Deep Deterministic Policy Gradient

1 Einleitung

Der Einsatz von Robotersystemen in Industrie, Haushalt und Servicebereichen nimmt stetig zu. Laut Prognosen der International Federation of Robotics IFR steigt die Gesamtanzahl der verkauften Serviceroboter für die professionellen und privaten Anwendungen in den Jahren 2019-2021 um durchschnittlich 21 % und 31 % pro Jahr (vgl. [IFR, 2018], S. 16). Die Anforderungen an den Autonomiegrad der modernen Serviceroboter führen zu neuen Herausforderungen im Bereich der Robotersteuerung: Eine manuelle Übergabe der Zielkoordinaten für die Ausführung der Roboterbewegungen reicht nicht mehr aus. Beispielsweise ist für Roboter, die im Weltraum, in einer Notaufnahme oder im Haushalt ihre Aufgaben erledigen, die Fähigkeit, auf unerwartete Situationen nicht nur vorprogrammiert zu reagieren, ausschlaggebend. Diese Roboter sollen selbst erlernen, den aktuellen Stand der Umgebung und eigene Limitationen (z. B. Körperbau oder Sensorfehler) wahrzunehmen und motiviert vom eigenen Ziel, autonom zu handeln.

1.1 Problembeschreibung

Die Forschungsrichtung Reinforcement Learning (dt.: Bestärktes Lernen, Abk.: RL) beschäftigt sich mit der Entwicklung von Lernverfahren, die es einem Agenten – einem Roboter oder einer Softwarekomponente – ermöglichen, die benötigten Fähigkeiten aus eigener Erfahrung zu erlernen. Die Kernidee des Reinforcement Learning geht auf die Psychologie und ein natürliches Lernen durch Trial-and-Error zurück. Der lernende Agent erkundet seine Umgebung, indem er unterschiedliche Aktionen ausprobiert. Für jede Aktion wird eine Belohnung oder Bestrafung vergeben, die in Analogie zu Menschen als positive oder negative Erfahrung gesehen werden kann. Die Motivation des Agenten besteht darin, eine Abbildung von Zuständen auf Aktionen, auch Strategie genannt, zu erlernen, die den Erwartungswert seiner gesamten Belohnungen maximiert. Dabei findet das Lernen wie bei Menschen oder Tieren lebenslang statt. Ein erfolgreicher Einsatz von Reinforcement Learning zur Bewegungssteuerung eines Roboters kann dazu führen, dass der Roboter in der Lage ist, angesichts der aktuellen Rahmenbedingungen und Ziele eigenständig seine Bewegungen zu steuern.
Obwohl das RL es erlaubt, die Anforderungen an den Autonomiegrad eines Roboters zu erfüllen, ist die Verwendung dieser Lernmethode zur Bewegungssteuerung eines Roboters problematisch. Erstens führt die Komplexität der Aufgabe und die Anzahl der Freiheitsgrade der modernen Roboter zu hochdimensionalen kontinuierlichen Zustands-

und Aktionsräumen, die nur von einer beschränkten Menge von RL-Algorithmen und auch nur im Zusammenspiel mit einem nichtlinearen Funktionsapproximationsalgorithmus, z. B. Deep Learning – neuronale Netze mit mehreren Schichten - gelöst werden können. Die Erfahrung des Roboters wird durch eine endliche Zahl der Netzwichtungen parametrisiert. Dabei muss der Roboter oft mehr als tausend Interaktionen mit seiner Umgebung ausführen, bis er eine optimale Handlungsstrategie bestimmt. Das verursacht jedoch zeitliche und finanzielle Kosten. Zweitens bei der Erforschung der Güte der bestimmten Aktionen kann der Roboter unabsichtlich sich selbst oder der Umgebung Schaden zufügen. Ein Beispiel dafür ist der Anwendungsfall, bei dem ein Roboter lernt, seine Navigation zu planen. Bei der Exploration der Umgebung wird der Roboter mehrmals gegen Hindernisse laufen, bis er anfängt, die Kollisionen zu vermeiden.
Um diese Schwierigkeiten zu umgehen, kann der Roboter in einer simulierten Umgebung trainiert werden. Dieser Ansatz wird häufig in der wissenschaftlichen Literatur benutzt, hat aber einige Nachteile. So ist ein in der Simulation erlerntes Verhalten nicht immer auf die reale Umgebung übertragbar, denn eine Simulation kann die Realität nicht exakt reproduzieren. Eine Handlungsstrategie, die sich in der Simulation als sicher und effektiv erwiesen hat, kann katastrophale Folgen in der realen Umgebung haben. Des Weiteren ist nicht immer eine Simulation verfügbar (vgl. [García & Fernández, 2019], S. 2).
Eine alternative Forschungsrichtung zur RL stellt das Imitation Learning (dt.: Imitationslernen) dar. In Imitation Learning besteht die Aufgabe des Roboters darin, die Strategie eines Experten zu reproduzieren. Der Einsatz des Imitation Learning zur Bewegungssteuerung eines Roboters erlaubt einem Endnutzer ohne Programmierkenntnisse, dem Roboter gewünschte Bewegungen durch das Vorzeigen beizubringen. Die Anwendung des Imitation Learning kann jedoch problematisch sein, wenn die Strategie des Experten nicht optimal ist oder die Limitation des Roboters bei den Demonstrationen nicht berücksichtigt werden. Des Weiteren kann die Erfassung der Demonstrationen aufwendig sein.
Um die Nachteile der beiden Methoden zu umgehen, können Imitation Learning und Reinforcement Learning kombiniert werden. Die durch die Imitation des Experten gelernte Strategie kann anhand der eigenen Erfahrung des Roboters verbessert werden. Da der Roboter die Verbesserung der Strategie durch das Reinforcement Learning mit einer suboptimalen und nicht mit einer zufälligen Strategie anfängt, verläuft der Lernprozess schneller.

1.2 Zielsetzung und Eingrenzung

Im Fokus dieser Arbeit steht die folgende *Forschungsfrage*: Wie kann die Kombination von zwei Lernarten, Imitation Learning und Reinforcement Learning, einem Roboter ermöglichen, ohne die Verwendung einer simulierten Umgebung eine flexible Strategie zur Bewegungssteuerung zu entwickeln. Zur Beantwortung der Forschungsfrage wird das Problem analysiert und an einem konkreten Szenario mit dem humanoiden Roboter NAO

betrachtet.
Das *Ziel* dieser Arbeit ist die Konzeption, Entwicklung und Evaluierung einer Softwareanwendung, die es einem NAO-Roboter ermöglicht, die Bewegungssteuerung für das Spiel Ball-in-a-Cup zu erlernen. Der Schwerpunkt liegt dabei auf der Auswahl und Umsetzung eines Lernalgorithmus.
Im Rahmen dieser Arbeit wird die Transparenz und Erklärbarkeit der gelernten Strategie nicht behandelt. Des Weiteren ist ein menschenähnliches Verhalten des Roboters nicht der Schwerpunkt dieser Arbeit.
Als optional wird die Implementierung einer grafischen Oberfläche zum Starten der Demonstrationenaufzeichnung und des Lernvorgangs eingestuft.

1.3 Fallstudie: Spiel Ball-in-a-Cup

Die Beantwortung der Forschungsfrage wird anhand des Spiels Ball-in-a-Cup mit dem NAO-Roboter V5 vorgenommen. Ball-in-a-Cup ist ein Geschicklichkeitsspiel, das Auge-Hand-Koordination und Feinmotorik erfordert. Die Idee des Spiels ist, einen Ball, der an einer Schnur aus einem Becher hängt, in Bewegung zu bringen und mit dem Becher zu fangen. Dafür muss der Roboter seinen rechten oder linken Arm, der jeweils sechs Freiheitsgrade besitzt, einsetzen. Zur Vereinfachung der Aufgabe wird die Anzahl der beweglichen Armgelenke auf drei reduziert.
Die Abbildung 1.1 zeigt den Versuchsaufbau.

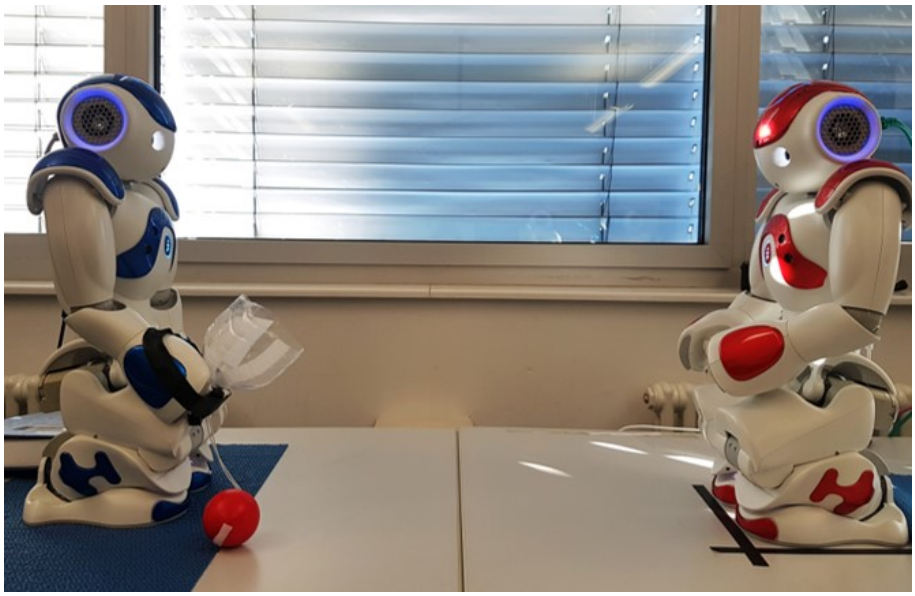


Abbildung 1.1: Versuchsaufbau für das Ball-in-a-Cup-Spiel
Quelle: Eigene Darstellung

Der lernende Roboter (blauer Roboter, Abbildung 1.1) steht auf einer Antirutschmatte und hält einen Becher in seiner rechten Hand. Der Becher ist farblos, transparent und aus Plastik hergestellt. Die Höhe des Bechers beträgt 11 cm, der Durchmesser der Öffnung

umfasst 10 cm. Damit der Roboter den Becher während des Spiels nicht verliert, wird die Hand des Roboters an der Halterung des Bechers mit Isolierband stabilisiert. Der Ball ist aus leichtem Plastik hergestellt, sodass der Roboter bei einer Kollision nicht beschädigt wird. Am inneren Rand des Bechers sowie am unteren Rand des Balls werden Klettbandstreifen angebracht, damit der Ball im Becher bleibt, wenn er gefangen wird. Die Schnur ist ca. 16 cm lang, der Durchmesser des Balls beträgt 6 cm.
Zur Ermittlung der Ballposition wird ein zweiter NAO-Roboter (roter Roboter, Abbildung 1.1) verwendet.

1.4 Anforderungen an die Softwareanwendung

In diesem Abschnitt werden die funktionalen und nicht-funktionalen Anforderungen an die Softwareanwendung definiert. Die Bestimmung der Anforderungen an den Algorithmus, der in der Lernkomponente der Software verwendet wird, erfolgt im Kapitel 4 und basiert auf den theoretischen Grundlagen (Kapitel 3).

Funktionale Anforderungen:

1. Die entwickelte Softwareanwendung soll es dem NAO-Roboter ermöglichen, eine Handlungsstrategie zur Bewegungssteuerung für das Spiel Ball-in-a-Cup zu erlernen und diese auszuführen. Als Kriterien zur Beurteilung der Güte der gelernten Strategie dienen die Lernzeit (gemessen in Episoden) und die durchschnittliche Belohnung des Roboters.
2. Die Anwendung soll eine Komponente zur Erfassung von Demonstrationen bereitstellen.
3. Die Ergebnisse des Lernprozesses sowie die Erfahrung des Roboters sollen in einer externen Datei, wie z. B. einer csv- oder pkl-Datei, gespeichert werden.
4. Die Anwendung soll die Funktionalität zur Evaluierung der Lernergebnisse bereitstellen.
5. Die Softwarekomponente soll in der Programmiersprache Python implementiert werden.

Nicht-funktionale Anforderungen:

1. Änderbarkeit
   Eine Modifikation der Hyperparameter des Lernalgorithmus, der Dimensionalität der Zustände und Aktionen sowie der numerischen Werte der Rewardfunktion soll möglich sein.
2. Erweiterbarkeit
   Der Programmcode soll gut lesbar, analysierbar und leicht erweiterbar sein. Die

wichtigen Entscheidungen und die Architektur der Anwendung sollen dokumentiert werden.

3. Zuverlässigkeit
   Die Anwendung soll resistent gegen Fehler sein und ohne unerwartete oder vorzeitige Beendigung funktionieren. Im Falle eines Programmabbruchs sollen die vom Agenten gesammelte Erfahrung und Wichtungen der neuronalen Netze wiederherstellbar sein.
4. Sicherheit des Roboters
   Die Sicherheit des Roboters soll bei der Entwicklung berücksichtigt werden:
   - Die Softwarekomponente soll die Überhitzung des Roboters vermeiden
   - Bei der Durchführung von Experimenten darf der Roboter nicht beschädigt werden (eine robuste Pose des Roboters und die Kollisionen mit dem Ball sind zu beachten)

1.5 Aufbau der Arbeit

Im *Kapitel 1* wird die Problemstellung beschrieben sowie die Forschungsfrage und das Ziel der Arbeit definiert. Des Weiteren stellt das Kapitel die Fallstudie, die zur Beantwortung der Forschungsfrage eingesetzt wird, und die Anforderungen an die zu entwickelnde Softwareanwendung vor.
Im *Kapitel 2* werden die für den Lösungsansatz relevanten theoretischen Inhalte erläutert. Es werden künstliche neuronale Netze mit dem Fokus auf Deep Feedforward Netze vorgestellt sowie die Themenblöcke Reinforcement Learning und Imitation Learning behandelt.
Das folgende *Kapitel 3* beschäftigt sich mit der kinematischen Struktur und Sensorik des NAO-Roboters. Außerdem wird das NAOqi-Framework zur Steuerung des Roboters vorgestellt.
Im Fokus des *Kapitels 4* steht die Auswahl und Konzeption des Lernalgorithmus für das Spiel Ball-in-a-Cup. Dafür werden zu Beginn des Kapitels ein Überblick über verwandte Ansätze gegeben sowie die Eigenschaften des Lernszenarios analysiert und Anforderungen an den Lernalgorithmus aufgezeigt. Nachdem der Lösungsansatz konzipiert wurde, werden zwei virtuelle Umgebungen vorgestellt, in denen der umgesetzte Algorithmus neben dem Ball-in-a-Cup Spiel evaluiert wird.
Die Implementierung der Softwareanwendung wird im *Kapitel 5* erläutert. Hier wird die Umsetzung des konzipierten Lösungsansatzes in der Programmiersprache Python unter Verwendung des Framework PyTorch geschildert.
Das *Kapitel 6* stellt die Ergebnisse der Experimente in den virtuellen Umgebungen „Pendulum-v0“ und „LunarLanderContinuous-v2“ sowie die Evaluation des Lernprozesses in dem Ball-in-a-Cup-Spiel mit dem NAO-Roboter dar.
Im abschließenden *Kapitel 7* werden die wichtigsten Ergebnisse der vorliegenden Arbeit zusammengefasst und ein Ausblick auf Folgearbeiten gegeben.

2 Der humanoide Roboter NAO

Der humanoide Roboter NAO (Version V5) wird in dieser Arbeit für die experimentellen Versuche eingesetzt, wobei der Roboter die Roller eines Agenten spielt. Dieses Kapitel gibt einen Überblick über die Hardware des Roboters sowie die für die Steuerung verwendete Software.

2.1 Kinematische Struktur

Der NAO-Roboter ist von der Firma SoftBank Robotics entwickelt worden. Der Roboter ist 5,74 cm hoch, 2,75 cm breit und wiegt 5.4 kg. Der Körper des Roboters besteht aus fünf einzelnen kinematischen Ketten, die sich aus untereinander verbundenen und durch Motoren angetriebenen Gelenken zusammensetzen. Diese kinematischen Ketten bilden die beweglichen Körperteile des Roboters: den Kopf, zwei Arme und zwei Füße, die auf dem Torso befestigt sind. Der Torso des Roboters besitzt keine Gelenke und ist daher unflexibel. Auf dem freien Ende jeder kinematischen Kette sowie auf dem Torso befinden sich Endeffektoren (s. Abbildung 2.1). Tabelle 2.1 listet alle kinematischen Ketten des Roboters sowie die darin enthaltenen Gelenke und Endeffektoren auf.

Kinematische Kette	**Gelenke**	**Endeffektoren**
Kopf	HeadPitch, HeadYaw	Head
Arm	ShoulderRoll, ShoulderPitch, ElbowYaw, ElbowRoll, WristYaw, Hand.	Arm
Fuß	HipPitch, HipRoll, KneePitch, AnklePitch, AnkleRoll, HipYawPitch	Leg
-	-	Torso

Tabelle 2.1: Kinematische Ketten, Gelenke und Endeffektoren des Roboters NAO V5
Quelle: In Anlehnung an [SoftBank Robotics, 2017]

Eine wichtige Anforderung bei der Programmierung eines Roboters ist die Lokalisierung des Roboters sowie der relevanten Objekte in einem drei-dimensionalen Koordinatensystem (Eng.: *frame*). Der NAO-Roboter verfügt über drei Koordinatensysteme, die mit Hilfe von Software des Herstellers (s. unten das NAOqi-Framework) angesprochen werden können:

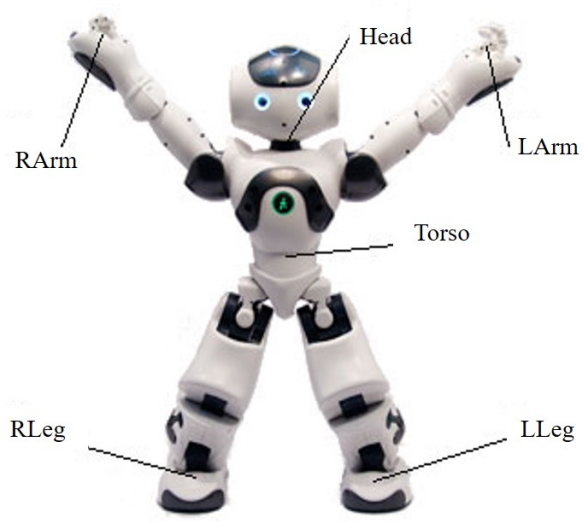


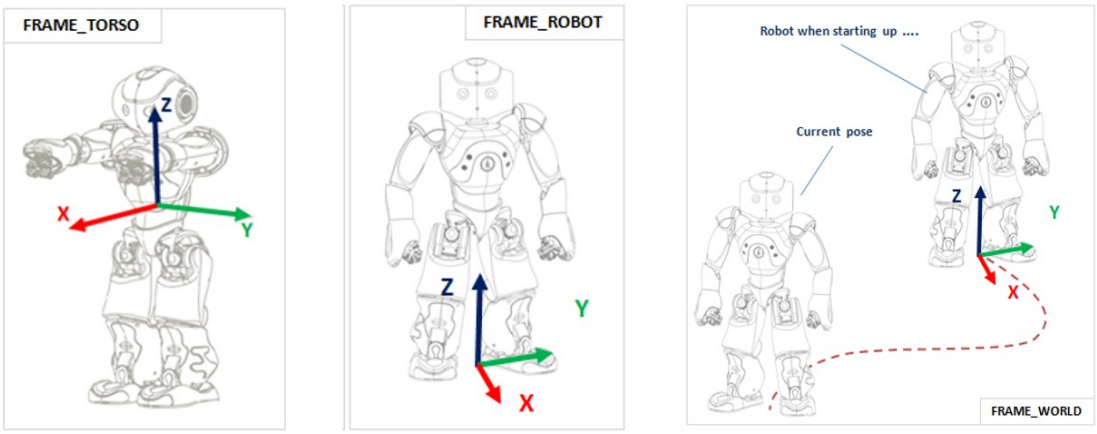
Abbildung 2.1: Endeffektoren des NAO-Roboters
Quelle: [SoftBank Robotics, 2017]

- Robotereigenes Koordinatensystem (FRAME_TORSO) mit dem Ursprung im Torso des Roboters (2.2a). Der Ursprung bewegt sich mit dem Roboter, wenn er geht und ändert die Orientierung, wenn der Roboter sich lehnt.
- Roboterzentrisches Koordinatensystem (FRAME_ROBOT), für das der Mittelwert der um die vertikale z-Achse projizierten Fußpositionen des Roboters als Ursprung gilt (Abb. 2.2b).
- Weltkoordinatensystem (FRAME_WOLD) mit einem festen Ursprung, der am Anfang der Berechnung mit dem Zentrum des Fußkoordinatensystems übereinstimmt und sich nicht ändert, wenn der Roboter sich bewegt (Abb. 2.2c).

Als Anzahl der Freiheitsgrade (Eng.: *degrees of freedom*, Abk.: DoF) eines Roboters wird die Anzahl der unabhängigen Variablen bezeichnet, die verändert werden können, um die Position des Roboters zu beeinflussen [Craig, 2005]. Die DoF-Achsen der Gelenke des NAO-Roboters sind nach der Roll-Pitch-Yaw Konvention bezeichnet. Jeder Arm und jeder Fuß des Roboters haben jeweils sechs DoF, der Kopf hat zwei DoF. Das ergibt insgesamt 26 Freiheitsgrade. Dadurch aber, dass zwei Hüftgelenke des Roboters LHipYawPitch und RHipYawPitch immer den gleichen Wert annehmen, besitzt der Roboter lediglich 25 Freiheitsgrade.

2.2 Sensorik

Sensoren ermöglichen einem Roboter, Informationen über seine Umgebung sowie seinen internen Zustand zu erfassen. Der NAO-Roboter ist mit zwei Kameras ausgestattet, die an seinem Vorderkopf platziert sind. Die Kameras bieten eine Auflösung bis zu 1280x960 Pixel und eine Bildfrequenz bis zu 30 Frames pro Sekunde an. Sie haben einen Blickwinkelgrad



a. b. c.

Abbildung 2.2: Koordinatensysteme des NAO-Roboters:
a) Robotereigenes Koordinatensystem, b) Roboterzentrisches Koordinatensystem, c) Weltkoordinatensystem
Quelle: [SoftBank Robotics, 2017]

von 47,64° auf der x-Achse und 60,97° auf der y-Achse. Es ist möglich während der Entwicklung die Parameter wie Sättigung, Auflösung, Farbraum, Kontrast, Helligkeit und weitere zu spezifizieren. Der Roboter kann zu jedem Zeitpunkt ausschließlich auf eine der beiden Kameras zugreifen [SoftBank Robotics, 2017].
In den Ohren des Roboters sind zwei Lautsprecher für die Audioausgaben eingebaut. Außerdem verfügt der Roboter über vier Mikrofone, über die er Geräusche oder Ansprache wahrnehmen kann. Die LEDs können für die Interaktion mit dem Menschen oder der Umgebung verwendet werden.
Zur Bestimmung der Gelenkpositionen ist der Roboter mit einem magnetischen 12-Bit Rotationsencoder ausgestattet. Eine Drehbewegung wird vom Sensor erfasst, der die Gelenkposition mit einer Genauigkeit von ca. 0,1° ermittelt.
Weiterhin verfügt der Roboter über neun taktile Sensoren und fünf Kontaktsensoren sowie über Sonarsensoren, die es ermöglichen, den Abstand zu Hindernissen in der Umgebung des Roboters abzuschätzen.

2.3 Software

Der Roboter ist mit dem Betriebssystem Linux NAOqi 2.1 ausgestattet. Das NAOqi-Framework, das eine Reihe von Programmierschnittstellen zur Steuerung des Roboters beinhaltet, ist die zentrale Software des Betriebssystems. Das Framework ist plattformunabhängig (Microsoft, Linux; MacOS) und sprachenübergreifend (Python, C++, Java).
Die Abbildung 2.3 veranschaulicht die Architektur des NAOqi-Framework. Der Broker ist eine Komponente, die bei der Initialisierung des Framework die Bibliotheken lädt, die in der Konfigurationsdatei autoload.ini spezifiziert sind. Jede Bibliothek besteht aus einem

oder mehreren Modulen, die wiederum eine Sammlung von Funktionen darstellen. Der Zugriff auf die Methoden eines Moduls erfolgt über ein Proxy-Objekt des entsprechenden Moduls.

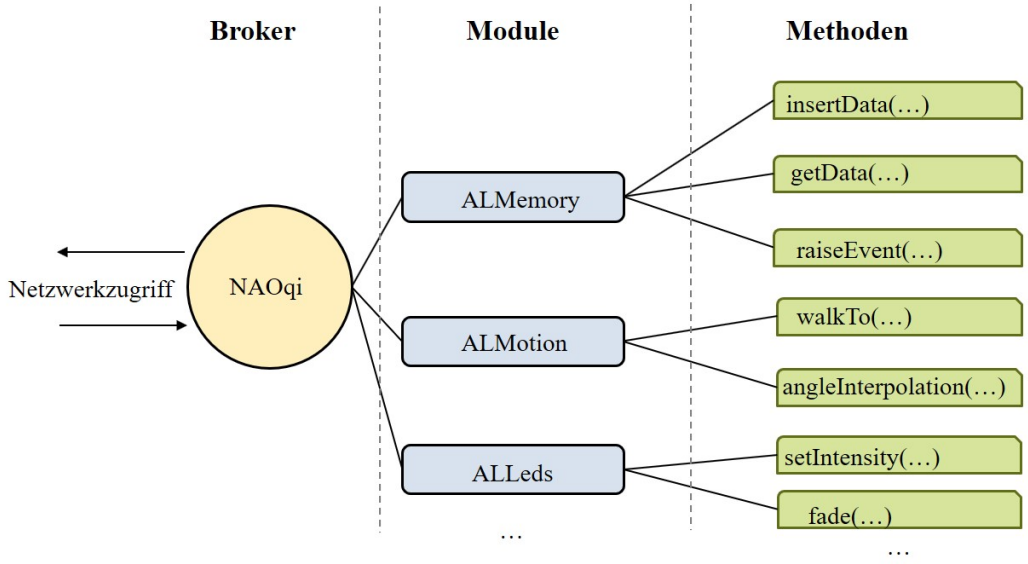


Abbildung 2.3: Architektur des NAOqi-Framework
Quelle: In Anlehnung an [SoftBank Robotics, 2017]

Im Folgenden werden einige für den praktischen Teil der Arbeit relevante Module des NAOqi-Framework vorgestellt.

ALMemory ALMemory ist ein Modul zur Datenverwaltung. Es spielt die Rolle des Gedächtnisses des Roboters und enthält drei Arten von Daten:

- die Informationen über den aktuellen Stand der Aktuatoren und Sensoren (z.B. Gelenkpositionen, Stand von LEDs oder Motoströmen).
- Events, z. B. die Erkennung eines Gesichtes oder das Hören eines bestimmten Wortes. Die Events können von den anderen Modulen des Framework abonniert werden. Falls ein Event auftritt, erhalten die Module, die das Event abonniert haben, eine Benachrichtigung und können darauf entsprechend reagieren, bspw. einen Audio-Befehl zur Begrüßung des Menschen abschicken.
- Mikro-Events, die sich von den Events nur in ihrer schnelleren Ausführungszeit unterscheiden.

Device Communication Manager (Abk: DCM) DCM ist für die Steuerung der elektronischen Geräte des Roboters mit der Ausnahme von Audio und Video zuständig. DCM stellt ein Bindeglied in der Kommunikation zwischen der „upper level“-Software (NAOqi Module) und der „lower level“-Software (Software für Leiterplatten) dar. Es erledigt in einem konstanten Takt (1 bis 10 ms) folgende zwei Aufgaben:

- leitet die Befehle, die von anderen Modulen des NAOqi-Framework an Aktuatoren gesendet werden, an elektronische Geräte weiter,
- liest die Informationen über Aktuatoren und Sensoren von elektronischen Geräten ab und aktualisiert die Daten in dem ALMemory Modul.

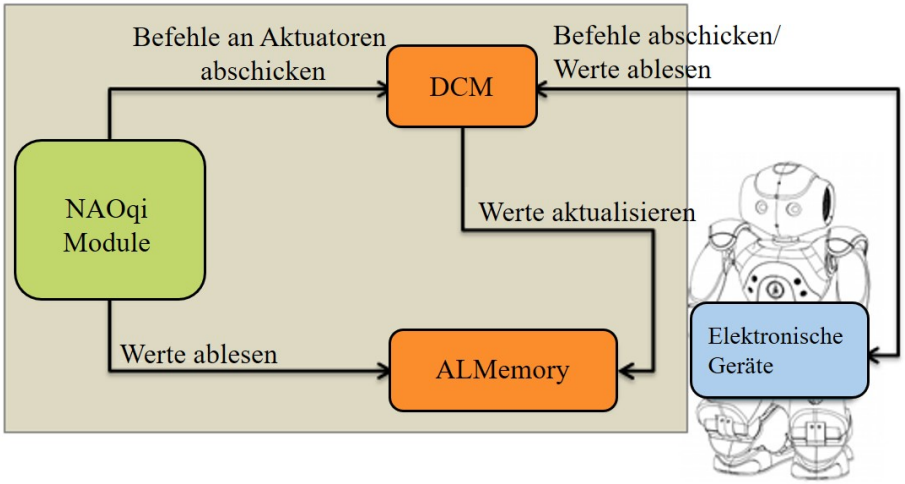


Abbildung 2.4: Die Kommunikation zwischen den Modulen des NAOqi-Framework und elektronischen Geräten des NAO-Roboters
Quelle: In Anlehnung an [SoftBank Robotics, 2017]

ALMotion Das ALMotion-Modul bietet die Methoden zur Bewegungssteuerung des Roboters, z. B. das Aus- und Einschalten von Gelenkmotoren, Änderung der Gelenkpositionen, Vermeidung von Kollisionen und weitere.
Das ALMotion-Modul bietet folgende Methoden zur Steuerung der Gelenkwinkel:

- *angleInterpolation()* - eine blockierende Methode zur Steuerung eines oder mehrerer Gelenkwinkeln in einem bestimmten Zeitintervall. Die Methode hat vier Parameter:
  - `name` (1...n): Namen der Gelenke oder kinematischer Ketten, die bewegt werden sollen
  - `angleList` (1...n): Zielwinkel in Radiant
  - `timeList` (1...n): Zeit, in der die Zielwinkel erreicht werden sollen
  - `isAbsolute` (1): wenn *True*, beschreibt der zweite Parameter die Zielwinkel relativ zur aktuellen Gelenkwinkeln
- *angleInterpolationWithSpeed()* - eine blockierende Methode zur Steuerung eines oder mehrerer Gelenkwinkel mit einer bestimmten Geschwindigkeit. Für jedes Gelenk ist nur ein Zielwinkel zulässig. Die Parameter der Methode sind:
  - `name` (1...n): Namen der Gelenke oder der kinematischen Ketten, die bewegt werden sollen
  - `targetAngles` (1...n): absolute Zielwinkel in Radiant

  - `maxSpeedFraction` (1...n): Ein Float-Wert im Intervall [0.0; 1.0], der die Geschwindikeit der Bewegung als ein Anteil der maximal möglichen Geschwindigkeit füt das Erreichen der Zielwinkel beschreibt

- *angleInterpolationBezier()* - eine blockierende Methode zur Interpolation einer Sequenz von Zielwinkeln unter Verwendung von Bezier-Kontrollpunkten. Die Methode hat folgende Parameter:
  - `jointNames` (1...n): Namen der Gelenke oder der kinematischen Ketten, die bewegt werden sollen
  - `times`: eine Matrix aus Array-Elementen, deren Zeileneinträge einem Gelenk und Spalteneinträge Kontrollpunkten entsprechen
  - `controlPoints` (1...n): Liste der Bezier-Kontrollpunkte
- *setAngles()* - eine nicht-blockierende Methode zur absoluten Änderung der Gelenkwinkel. Die Funktionsweise und die Parameter sind analog zur *angleInterpolationWithSpeed()*-Methode
- *changeAngles()* - eine nicht-blockierende Methode zur relativen Änderung der Gelenkwinkel. Die Funktionsweise und die Parameter sind analog zur *angleInterpolationWithSpeed()*-Methode

ALTracker Das `ALTracker`-Modul ermöglicht es dem Roboter, verschiedene Ziele (roter Ball, Gesicht, Landmarke) zu verfolgen. Je nach dem eingestellten Modus kann der Roboter das Zielobjekt nur mit dem Kopf oder mit dem ganzen Körper (mit oder ohne Laufen) verfolgen.
Das Modul identifiziert die Position [x, y, z] des vom Roboter gesehenen Ziels im gewünschten Koordinatensystem. Die Methode `getTargetPosition()` gibt die Position des Ziels zurück.

ALPosture Das `ALPosture`-Modul enthält vordefinierte Körperhaltungen, wie z. B. „Crouch“ oder „ LyingBack“, die der Roboter annehmen kann. Dabei ist möglich, die Geschwindigkeit der Bewegung zu spezifizieren.

2.4 Zusammenfassung

In diesem Kapitel wurden die kinematische Struktur und die Sensorik des NAO-Roboters erläutert. Außerdem wurde auf die Architektur des NAOqi-Framework, die zur Steuerung des Roboters eingesetzt wird, eingegangen. Die folgenden Module des Framework wurden näher betrachtet:

- ALMemory – Modul zur Datenverwaltung,

- Device Communication Manager – Modul zur Steuerung der elektronischen Geräte des Roboters (mit Ausnahme von Audio und Video)
- ALMotion – Modul zur Bewegungssteuerung
- ALTracker – Modul zur Verfolgung verschiedener Ziele (rotes Ball, Gesicht, Landmarke)
- ALPosturer – Modul zur Annahme einer der vordefinierten Körperhaltungen

3 Theoretische Grundlagen

Die Machine Learning-Algorithmen (Abk.: ML-Algorithmen) lassen sich nach dem zugrunde liegenden Lernstil in drei Kategorien aufteilen (vgl. [Boersch et al., 2007], S. 273 – 279):

- Supervised Learning-Algorithmen
  Beim Supervised Learning (dt.: überwachtes Lernen, Abk.: SL) liegt eine Datenmenge der Trainingsbeispiele vor, die richtige Antworten, sogenannte Labels, beinhalten. Die Aufgabe des Algorithmus besteht darin, selbständig Kriterien zu entwickeln, mit denen die Labels den möglichen Eingaben zugeordnet werden können. Das SL wird im Abschnitt 3.1 im Kontext der künstlichen neuronalen Netze näher betrachtet.
- Unsupervised Learning-Algorithmen
  Die Trainingsbeispiele beim Unsupervised Learning (dt.: unüberwachtes Lernen) enthalten dagegen keine Labels. Der Algorithmus versucht, selbstständig Strukturen und Unterschiede in den Eingabedaten zu finden.
- Reinforcement Learning-Algorithmen
  Beim Reinforcement Learning findet das Lernen nicht nur auf einer begrenzten Datenmenge statt, sondern auf den Daten, die aus der Interaktion mit der Umgebung entstehen. Das Ziel des Lernens besteht darin, eine optimale Strategie zu bestimmen. Das RL wird im Abschnitt 3.2 detailliert erläutert.

3.1 Künstliche neuronale Netze

Künstliche neuronale Netze werden in der künstlichen Intelligenz als ein leistungsstarkes Werkzeug zur Funktionsapproximation eingesetzt (vgl. [Goodfellow et al., 2016], S. 15-16). Ein künstliches neuronales Netz besteht aus Knoten, die durch gerichtete Kanten verbunden sind. Die Knoten entsprechen einer Verarbeitungseinheit und werden als künstliche Neuronen bezeichnet. Die Anordnung von Neuronen und Verbindungen zwischen denen bestimmt die Architektur des Netzes (vgl. [Boersch et al., 2007], S. 281).

Im Folgenden wird ein mathematisches Modell eines künstlichen Neurons und die Netzarchitektur Deep-Feedforward-Netze (dt.: Vorwärtsvermittlungsnetze) vorgestellt. Danach wird der Backpropagation-Algorithmus, der sich mit dem Erlernen der Wichtungen eines Deep-Feedward-Netzes befasst, erläutert sowie auf die Ziele und mögliche Probleme des

Lernprozesses eingegangen. Abschließend werden Maßnahmen zur Regularisierung und Optimierung in Deep Learning betrachtet.

3.1.1 Mathematisches Neuronenmodell

Abbildung 3.1 veranschaulicht ein mathematisches Modell eines künstlichen Neurons *j*.

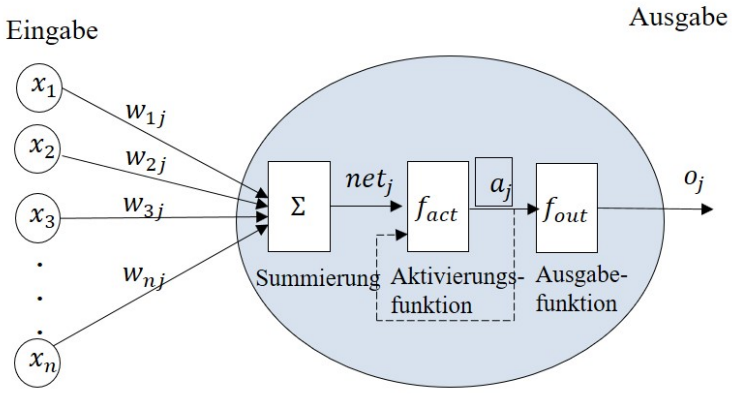


Abbildung 3.1: Mathematisches Modell eines Neurons *j*.
Quelle: In Anlehnung an [Boersch et al., 2007], S. 282

Das Neuron j nimmt die *eingehenden Signale* auf, die vom Ausgang o_i eines anderen Neurons i oder direkt von einem Eingabemuster $x = (x_1, x_2, \ldots x_n)$ kommen können. Die Signale werden durch die *Wichtungen* w_{ij} verstärkt oder geschwächt: Eine niedrige Wichtung hemmt ein Signal, eine hohe Wichtung verstärkt es. Die Wichtung vom Neuron i zum Neuron j wird mit w_{ij} bezeichnet. Die Netzaktivität net_j summiert die gewichteten Eingangssignale auf:

$$net_j = \sum_i w_{ij} o_i \text{ oder } net_j = \sum_i w_{ij} x_i \tag{3.1}$$

Die *Aktivierung* a_j eines Neurons j wird mithilfe einer *Aktivierungsfunktion* f_{act} bestimmt. Der Wert der Aktivierung eines Neurons zu einem Zeitpunkt t wird im allgemeinen Fall von der aktuellen Netzaktivität, der alten Aktivierung des Neurons und dem Schwellenwert Θ beeinflusst:

$$a_j(t) = f_{act}(net_j(t), a_j(t-1), \Theta_j) \tag{3.2}$$

Wird die alte Aktivierung bei der Berechnung der neuen Aktivierung nicht berücksichtigt, entfällt die gestrichelte Verbindung in der Abbildung 3.1 und die Abhängigkeit f_{act} von a_j*(t-1)* in der Formel 3.2. Eine der einfachsten Aktivierungsfunktionen ist die binäre Schwellenwertfunktion, die nur zwei Werte (0 oder 1) annehmen kann:

$$f_{act} = \begin{cases} 0, \text{if } net_j \leq \Theta \\ 1, \text{sonst} \end{cases} \tag{3.3}$$

Wie die Gleichung 3.3 veranschaulicht, wird ein Neuron aktiviert, wenn die Netzaktivität f_{act} größer als oder gleich einem Schwellenwert Θ ist.
In der Fachliteratur wird der Schwellenwert oft als eine zusätzliche Wichtung $w_{n+1} = -\Theta$ dargestellt, die mit einer stets aktivierenden Eingabe $x_{n+1} = 1$, dem sogenannten *ON-Neuron* oder *Bias*, multipliziert wird. Damit wird die Schwellenwertfunktion in die Netzaktivität aufgenommen und die Aktivierung eines Neurons durch den Vergleich der Netzaktivität mit 0 definiert. Die binäre Schwellenwertfuntkion wird in diesem Fall durch folgende Formel beschrieben:

$$f_{act} = \begin{cases} 0, \text{if } net_j - \Theta \leq 0 \\ 1, \text{sonst} \end{cases}$$

Die Aktivierung eines Neurons j führt dazu, dass das Neuron ein Signal an die Neuronen, die mit j verbunden sind, übermittelt. Das ausgehende Signal wird mithilfe einer Ausgabefunktion f_{out} generiert (vgl. [Boersch et al., 2007], S. 282-284).
Bei den Netzen, die im Folgenden vorgestellt und in dem praktischen Teil der Arbeit verwendet werden, wird die Ausgabefunktion Identität benutzt, sodass die Ausgabe o_j gleich seiner Aktivierung ist:

$$o_j = f_{out}(a_j) = a_j = f_{act}(net_j) \tag{3.4}$$

3.1.2 Deep-Feedforward-Netze

Das Ziel eines Feedforward-Netzes besteht darin, eine Soll-Funktion $y = f(x)$ zum Mapping zwischen der Eingabe x und der Ausgabe y zu approximieren. Dafür generiert das Netz eine Ist-Funktion $\hat{y} = f(x; W)$, deren Parameter W den Wichtungen des Netzes entsprechen. Im Laufe des Lernens verfeinert das Netz die Parameter W, um eine bessere Funktionsapproximation zu erzielen (vgl. [Goodfellow et al., 2016], S. 163).

3.1.2.1 Aufbau

Deep-Feedforward-Netze, auch als Multi-Layer Perceptrons (MLPs) bekannt, sind eine der grundlegenden Deep Learning Architekturen. Ein Deep Feedforward Netz besteht aus einer Eingabe- und einer Ausgabeschicht sowie einer oder mehreren versteckten Schichten, wobei jedes Neuron mit jedem anderen in der nächsten und der vorherigen Schicht verbunden ist. Die Verbindungen zwischen den Neuronen einer Schicht sowie die Verbindungen, die rückwärts zur Ausgaberichtung führen, sind in Deep-Feedforward-Netzen verboten. Diese Einschränkung sorgt dafür, dass die Informationen in Deep-Feedforward-Netzen vorwärts durchgereicht werden. Daher stammt die Bezeichnung „Feedforward“ (dt.: vorwärtsgerichtet). Das Wort „Deep“ weist darauf hin, dass die Netze mehrere versteckte Schichten haben. Die Abbildung 3.2 stellt ein Deep Feedforward Netz mit zwei versteckten Schichten, n Eingabeneuronen und m Ausgabeneuronen dar.

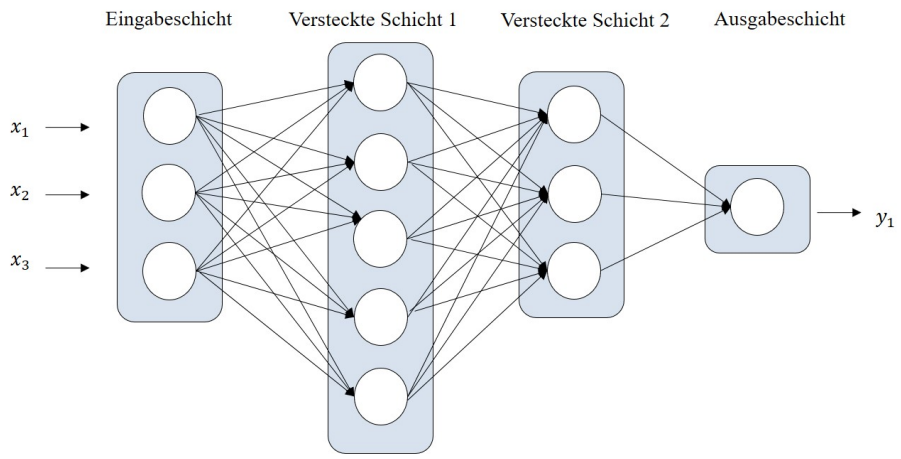


Abbildung 3.2: Deep-Feedforward-Netz mit zwei versteckten Schichten, drei Eingabeneuronen und einem Ausgabeneuron
Quelle: Eigene Darstellung

Die *Eingabeschicht* (Eng.: *input layer*) ist die erste Schicht eines Netzes. Die Neuronen der Eingabeschicht nehmen einen n-dimensionalen Eingabevektor $x = (x_1, x_2, \ldots x_n)$ auf und leiten diesen ohne die Anwendung der Aktivierungsfunktion weiter. Die Anzahl der Neuronen in der Eingabeschicht ist gleich der Dimension des Eingabevektors.
Die *Ausgabeschicht* (Eng.: *output layer*) ist die letzte Schicht eines Netzes. Die Ausgabesignale der Neuronen dieser Schicht dienen als Ausgabe des gesamten Netzes. Daher entspricht die Anzahl der Neuronen in der Ausgabeschicht der Dimension des Ausgabevektors $y = (y_1, y_2, \ldots y_m)$.
Die *versteckten Schichten* (Eng.: *hidden layers*) sind die mittleren Schichten eines Netzes. Die Ausgabesignale der Neuronen der mittleren Schichten treten nicht als Ausgänge des Netzes in Erscheinung, sondern dienen als Eingabe für die Neuronen der nächsten Schicht, sind also „versteckt“. Die Anzahl der versteckten Schichten legt die *Tiefe* eines neuronalen Netzes fest.
Laut dem Theorem der universellen Approximation ([KurtHornik et al., 1989], [Cybenko, 1989]) sind die Feedforward-Netze mit einer linearen Ausgabeschicht sowie mindestens einer versteckten Schicht mit einer „zwingenden“ Aktivierungsfunktion (z. B. der Sigmoid-Funktion) in der Lage, jede beschränkte kontinuierliche Funktion mit einem beliebigen, von Null verschiedenen Fehler zu approximieren, sofern sie genügend versteckte Neuronen haben (vgl. [Goodfellow et al., 2016], S. 192).
Es ist wichtig, zwei Punkte bzgl. des Theorems zu betonen. Erstens kann nach dem Theorem der universellen Approximation ein Feedforward Netz eine beliebige Funktion *repräsentieren*. Dies garantiert allerdings nicht, dass der Trainingsalgorithmus die Funktion *erlernen* kann. Das Lernen kann grundsätzlich aus den zwei folgenden Gründen scheitern:

- der für das Lernen verwendete Algorithmus ist nicht in der Lage, die Wichtungen zu bestimmen, die der gewünschten Funktion entsprechen, oder
- der Algorithmus wählt als Folge des Underfitting/Overfitting (s. Abschnitt 2.1.2.3) eine falsche Funktion aus.

Zweitens charakterisiert das Theorem zwar die universelle Approximationseigenschaft eines Feedforward-Netzes, macht aber keine Angaben dazu, wie viele Neuronen in einer versteckten Schicht für eine gute Approximation nötig sind. Eine theoretische Analyse von Barron [Barron, 1993] hat gezeigt, dass obwohl eine versteckte Schicht ausreichend ist, um eine beliebige Funktion darzustellen, kann diese Schicht für den praktischen Einsatz zu groß sein und am Erlernen und korrekten Generalisieren scheitern. In vielen Fällen führt eine Architektur mit mehreren versteckten Schichten zur Reduzierung der Neuronen, die zur Repräsentation der gewünschten Funktion erforderlich sind. In der Praxis hat die Auswahl der Netzarchitektur einen großen Einfluss auf die Performanz des Netzes. Daher werden Netzarchitekturen entwickelt, die bei spezifischen Aufgaben eingesetzt werden können. Beispielsweise wird im Bereich der Computer Vision die CNN-Architektur (Abk. von Eng.: convolutional neural network) bevorzugt (vgl. [Goodfellow et al., 2016], S. 192-195).

3.1.2.2 Backpropagation

Der Algorithmus Backpropagation [Rumelhart et al., 1986] wird für das überwachte Erlernen der Wichtungen eines Feedforward-Netzes eingesetzt. Der Algorithmus bezieht sich auf die Weitergabe der Fehler zwischen der gewünschten und der tatsächlichen Ausgabe des Netzes rückwärts durch alle Schichten und die Aktualisierung der Wichtungen zwischen den Neuronen entsprechend ihrem Beitrag zum Ausgabefehler(vgl. [Boersch et al., 2007], S. 293-294). Dabei wird das Ziel verfolgt, die Wichtungen W^* zu bestimmen, die in dem minimalen Wert der Kostenfunktion $J(W)$ resultieren: $W^* = argmin_W J(W)$. Für die Optimierung wird in Backpropagation das Gradientenabstiegsverfahren eingesetzt.
Gradientenverfahren werden zum Finden eines lokalen Optimums einer kontinuierlichen und differenzierbaren Funktion verwendet. Um den minimalen Wert der Funktion zu finden, wird der *Gradientenabstieg* (Eng.: gradient descent) benutzt, wobei das Maximum der Funktion mithilfe des *Gradientenaufstiegs* (Eng.: gradient ascent) bestimmt wird (vgl. [Poole & Mackworth, 2017], S. 165).
Der Grundgedanke des Gradientenverfahrens ist, sich von einem zufällig ausgewählten Startpunkt in Richtung des Abstiegs der zu minimierenden Funktion bzw. in Richtung des Aufstiegs der zu maximierenden Funktion fortzubewegen. Die Richtung des Anstiegs lässt sich durch die Berechnung des Gradienten, d. h. der partiellen Ableitung der Funktion in Bezug auf die Parameter, bestimmen. Das folgende Beispiel und die Abbildung 3.3 skizzieren die Idee des Gradientenabstiegsverfahrens.

Gegeben sei die Funktion $y = (x-1)^2 + 1$. Es soll das Minimum der Funktion y mittels des Gradientenabstiegs ermittelt werden. Ausgehend vom Startpunkt (x_0, y_0) wird in jedem Punkt der Gradient von y als $\nabla_x y = \frac{\partial y}{\partial x}$ berechnet und ein Schritt in die entgegengesetzte Richtung des Gradienten gemacht. Der neue Wert von x wird wie folgt bestimmt:

$$x_1 \;=\; x_0 - \alpha \nabla_{x_0} y, \tag{3.5}$$

α ist die Lernrate – eine Konstante, die die Schrittgröße bestimmt. Die Schritte werden solange ausgeführt, wie sich der Wert von y verkleinert. Auf der Abbildung 3.3 ist erkennbar, dass je flacher der Gradient ist, desto kleinere Schritte werden ausgeführt (vgl. [Rashid, 2017], S. 70-78). Bei multidimensionalen Optimierung macht der Gradientenabstieg einen Schritt in jede Dimension proportional zu der partiellen Ableitung in dieser Dimension (vgl. [Poole & Mackworth, 2017], S. 166).

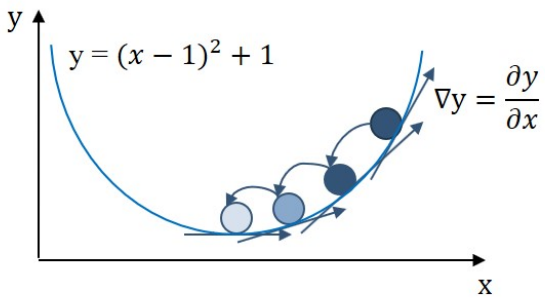


Abbildung 3.3: Darstellung des Gradientenabstiegsverfahrens für die Funktion $y = (x-1)^2 + 1$
Quelle: In Anlehnung [Rashid, 2017], S.76

Gegeben sei eine Menge von Trainingsbeispielen, die jeweils einen Eingabevektor x und den gewünschten m-dimensionalen Ausgabevektor y beinhalten. Das Netz berechnet einen Ausgabevektor $\hat{y} = (\hat{y}_1, \hat{y}_2, ...\hat{y}_m) = (o_1, o_2, ...o_m)$, wobei $o_1, o_2, ...o_m$ die Werte der Ausgabeneuronen sind. Als *Kostenfunktion J(W)*, auch oft als Fehlerfunktion bezeichnet, wird die mittlere quadratische Abweichung (Eng.: mean squared error, Abk.: MSE) zwischen der gewünschten und der vom Netz berechneten Ausgabe verwendet:

$$J = \frac{1}{m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2 = \frac{1}{m}\sum_{j=1}^{m}(y_j - o_j)^2 \tag{3.6}$$

Die Durchführung des Backpropagation-Algorithmus beinhaltet folgende Schritte (vgl. [Boersch et al., 2007], S. 293-296):

1. Initialisierung der Wichtungen mit kleinen Zufallszahlen
2. Auswahl eines Trainingsbeispiels (x, y) und die Eingabe des Vektors x im Netz
3. Forwardpropagation (dt.: Vorwärtspropagierung).
   Die Informationen aus dem Eingabevektor werden im Netz von einer Schicht zur anderen vorwärts weitergegeben und verarbeitet, bis der Ausgabevektor $\hat{y}$ berechnet wird. Die Bestimmung des Ausgabevektors wurde im Abschnitt 2.1.1 erläutert.
4. Backpropagation (dt.: Rückwärtspropagierung)
   In diesem Schritt werden die Fehler der Ausgabeneuronen berechnet und rückwärts weitergegeben. Die Fehler der Neuronen in vorderen Schichten werden von Fehlern der Ausgabeneuronen abgeleitet. Anschließend werden die neuen Wichtungen anhand der Neuronenfehler bestimmt.

Zur Berechnung der neuen Wichtungen wird das Gradientenabstiegsverfahren auf die Kostenfunktion J angewendet. Die neuen Wichtungen vom Neuron i zum Neuron j zum Zeitpunkt $(t+1)$ werden aus den alten Wichtungen zum Zeitpunkt t und dem Gradienten der Kostenfunktion in Bezug auf Wichtungen berechnet:

$$w_{ij}(t+1) = w_{ij}(t) - \frac{1}{2}\alpha\nabla_{w_{ij}(t)}J \tag{3.7}$$

Dabei erfolgt die Aktualisierung der Wichtungen entgegen der Richtung des Anstiegs der Fehlerfunktion ∇J und proportional zur Lernrate α.
Der Gradient der Kostenfunktion wird wie folgt berechnet:

$$\nabla_{w_{ij}(t)}J = -2\delta_j \cdot o_i$$

Die Formel zur Bestimmung der neuen Wichtungen (3.7) nimmt die folgende Form an:

$$w_{ij}(t+1) = w_i - \frac{1}{2}\alpha\nabla_{w_{ij}(t)}J = w_{ij}(t) + \alpha \cdot \delta_j \cdot o_i$$

Dabei steht δ_j für den Fehler des Neurons j und o_i für die Ausgabe des Neurons i. Gehört das Neuron j zur Ausgabeschicht, berechnet sich sein Fehler durch den Soll-Ist-Vergleich:

$$\delta_j = (y_j \ - \ o_j) \cdot \frac{\partial(f_{act}(net_j))}{\partial net_j}.$$

Der Term $\frac{\partial(f_{act}(net_j))}{\partial net_j}$ bezeichnet die Ableitung der Aktivierungsfunktion des Neurons j an der Stelle der aktuellen Netzaktivität net_j und ist je nach dem Typ der Aktivierungsfunktion unterschiedlich aufzulösen.
Liegt das Neuron j in einer versteckten Schicht, berechnet sich sein Fehler wie folgt:

$$\delta_j = (\sum_{k \in K} w_{jk} \cdot \ \delta_k) \cdot \frac{\partial(f_{act}(net_j))}{\partial net_j}$$

Als K wird die Menge der Neuronen bezeichnet, die vom Ausgang des Neurons j angesteuert werden. Der Fehler des versteckten Neurons j ist also proportional zur gewichteten Summe der Fehler δ_k, zu denen j beigetragen hat.

5. Wiederholung der Schritte 2-4 mit dem nächsten Trainingsbeispiel, bis der Netzfehler klein genug ist oder maximale Anzahl der Durchläufe erreicht ist.

3.1.2.3 Generalisierungsfähigkeit

Die Gesamtdaten, die für das Training eines neuronalen Netzes zur Verfügung stehen, werden in Trainingsmenge und Testmenge getrennt. Dabei gilt die Annahme, dass die Trainingsmenge und die Testmenge identisch verteilt und disjunkt sind. Der Lernprozess findet auf der Trainingsmenge statt und die abschließende Evaluation des trainierten Netzes wird auf der Testmenge durchgeführt. Der Fehler des Algorithmus auf den Daten aus der Trainingsmenge wird als *Trainingsfehler* bezeichnet. Der *Generalisierungsfehler* ist als Erwartungswert des Netzfehlers für die bisher unbeobachteten Eingabedaten definiert. Der Generalisierungsfehler kann durch die Ermittlung der Leistung des Netzes auf der Testmenge geschätzt werden (vgl. [Goodfellow et al., 2016], S.107-108).
Um die Änderung des Generalisierungsfehlers bereits während des Lernprozesses beobachten zu können, wird die Trainingsmenge in der Praxis in zwei Mengen getrennt: die Validierungsmenge (ca. 20% der ursprünglichen Trainingsmenge) und die Trainingsmenge (ca. 80% der ursprünglichen Trainingsmenge). Das Modell wird auf der neuen Trainingsmenge trainiert, wobei für die Schätzung des Generalisierungsfehlers die Validierungsmenge verwendet wird (vgl. [Goodfellow et al., 2016], S. 117-118).
Die zentrale Herausforderung beim ML besteht darin, dass der Algorithmus erstens die Muster in den Trainingsdaten genau erfassen muss und zweitens eine hohe Leistung auf neuen, bisher unbeobachteten Eingabedaten zeigen muss. Das heißt gewünscht ist ein kleiner Trainingsfehler, ein kleiner Validierungsfehler und ein kleiner Generalisierungsfehler (vgl. [Goodfellow et al., 2016], S. 107).
Die Situation, wenn der Trainigsfehler zu groß ist, wird als *Underfitting* (dt.: Unteranpassung) bezeichnet. Das Underfitting findet statt, wenn das Model nicht in der Lage ist, die wichtigen Muster in den Daten zu erkennen und folglich den Zusammenhang zwischen der Ein- und Ausgabe nicht richtig darstellt (vgl. [Ng, 2017], S. 1-2). Im Zusammenhang mit Underfitting lässt sich der Begriff Bias definieren. Als *Bias* eines Schätzers wird die Differenz zwischen dem Erwartungswert des Schätzers (bzgl. der Daten) und dem tatsächlichen Wert der Funktion bezeichnet (vgl. [Goodfellow et al., 2016], S. 121-122):

$$bias(\hat{y}) = \mathbb{E}[\hat{y}] - y$$

Wenn das gelernte Modell sich zu stark an die Trainingsdaten anpasst, sodass es auch ein mögliches Rauschen in den Daten als Muster erlernt, wird von *Overfitting* (dt.: Überanpassung) des Modells gesprochen (vgl. [Ng, 2017], S.1-2). Das Overfitting kann beim Training dadurch erkannt werden, dass der Validierungsfehler des Modells beim Lernprozess steigt, während der Trainingsfehler sinkt (vgl. [Goodfellow et al., 2016], S. 108). Das Overfitting weist auf eine hohe Varianz des Modells hin. Die *Varianz* des Schätzers $\hat{y}$ misst die Abweichung vom Erwartungswert des Schätzers, die jeder einzelne Datensatz aufweist (vgl. [Goodfellow et al., 2016], S. 123-124):

$$var(\hat{y}) = \mathbb{E}[\hat{y}^2] - \mathbb{E}[\hat{y}]^2$$

Die Abbildung 3.4 illustriert Underfitting, Overfitting und eine optimale Anpassung des Modells an die Trainingsdaten.

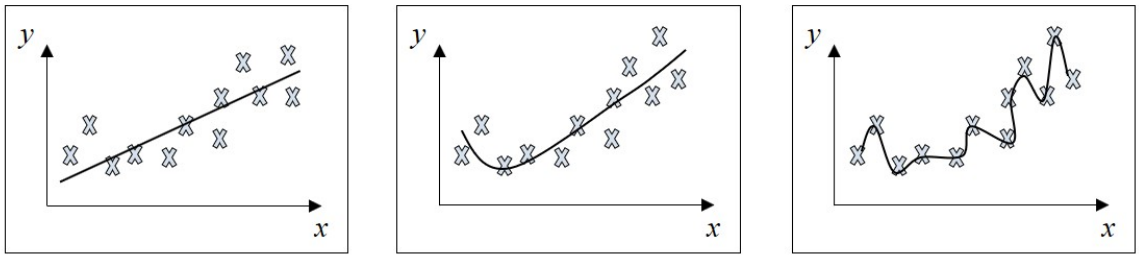


Abbildung 3.4: Evaluation eines Modells: a) Unterfitting b) Optimale Anpassung c) Overfitting
Quelle: In Anlehnung an [Ng, 2017], S. 1

Der Bias und die Varianz eines Modells können durch unterschiedliche Faktoren kontrolliert werden, z. B. durch die Steuerung der Kapazität des Modells. Als *Kapazität* wird die Fähigkeit des Modells bezeichnet, sich einer Vielzahl von Funktionen anzupassen. Modelle mit einer niedrigen Kapazität haben Probleme, komplexe Funktionen zu approximieren, wobei Modelle mit einer hohen Kapazität zur Überanpassung neigen, wenn sie sich an einer einfachen Funktion anpassen müssen. Eine Möglichkeit, die Kapazität eines Lernalgorithmus zu konfigurieren, ist den Hypothesenraum zu bestimmen, also die Menge der Funktionen, die das Modell in der Lage ist zu approximieren. Die Kapazität eines neuronalen Netzes wird durch ihre Tiefe und die Anzahl der Neuronen in jeder versteckten Schicht bestimmt. Die tatsächliche repräsentative Kapazität eines Modells kann jedoch dadurch reduziert werden, dass der zugrunde liegende Algorithmus nicht nach der besten Funktion sucht, sondern lediglich nach einer Funktion, die die Kostenfunktion relativ gut optimiert. So konvergieren beispielsweise die Gradientenverfahren gegen ein lokales Optimum.
Die Abbildung 3.5 zeigt den Zusammenhang zwischen Generalisierungsfehler (MSE-Fehlerfunktion), Kapazität, Bias, Varianz, Overfitting und Underfitting.
Die MSE-Fehlerfunktion (die y-Achse) bezieht den Bias und die Varianz ein:

$$MSE = \mathbb{E}[(\hat{y} - y)^2] = bias(\hat{y})^2 + var(\hat{y})$$

Mit zunehmender Kapazität (auf der x-Achse) sinkt der Bias des Modells, die Varianz steigt aber an, wodurch eine U-förmige Kurve für den Generalisierungsfehler entsteht. Dieses Verhalten der beiden Größen Bias und Varianz wird als *Bias-Varianz Trade-Off* bezeichnet. Die optimale Kapazität ist diese, bei der der Generalisierungsfehler minimal ist: Unterhalb des Optimums kommt es zur Underfitting, darüber zur Overfitting [Goodfellow et al., 2016].

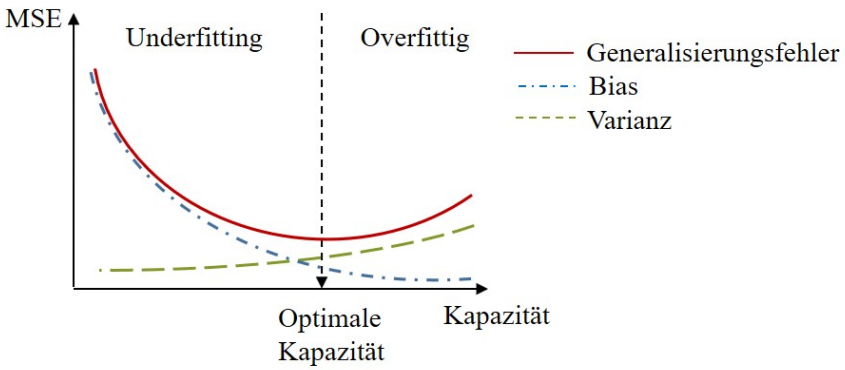


Abbildung 3.5: Zusammenhang zwischen Generalisierungsfehler (MSE-Fehlerfunktion), Kapazität, Bias, Varianz, Overfitting und Underfitting.
Quelle: In Anlehnung an [Goodfellow et al., 2016], S. 127

3.1.2.4 Regularisierung

Eine andere Möglichkeit, die Kapazität eines Modells zu steuern, ist die Festlegung einer Präferenz für eine Lösung im Hypothesenraum gegenüber einer anderen Lösung. Dafür gibt es in Maschine Learning unterschiedliche Ansätze (wie z. B. die Definition von zusätzlichen Einschränkungen für das Modell oder das Hinzufügen von zusätzlichen Termen in der Zielfunktion), die unter dem Begriff Regularisierung zusammengefasst werden. Als *Regularisierung* wird jede Modifikation des Lernalgorithmus bezeichnet, die darauf abzielt, den Generalisierungsfehler – nicht aber den Trainingsfehler - zu reduzieren. Im Kontext des Deep Learning besteht das Ziel der Regularisierung darin, das Overfitting zu vermeiden (vgl. [Goodfellow et al., 2016], S. 115).
Der folgende Abschnitt erläutert einige Strategien zur Regularisierung in Deep-Feedforward-Netzen.

Early Stopping Das Verfahren Early Stopping (dt.: Frühes Stoppen) zählt zu den weitverbreiteten Regularisierungsmaßnahmen im Deep Learning (vgl. [Goodfellow et al., 2016], S. 239). Wie bereits oben erwähnt, ist das Overfitting durch den Aufstieg des Validierungsfehlers bei weiterem Absinken des Trainingsfehlers erkennbar. Die Idee des Early Stopping Verfahrens besteht darin, diesen Zeitpunkt im Lernverlauf zu erkennen und das Training abzubrechen (vgl. [Boersch et al., 2007], S. 304).
In der Praxis wird diese Strategie dadurch realisiert, dass jedes Mal, wenn sich der Validierungsfehler im Laufe des Lernens verkleinert, eine Kopie der Netzwichtungen gespeichert wird. Wenn der Lernalgorithmus terminiert, wird die letzte Kopie und nicht die aktuellen Netzwichtungen zurückgegeben. Der Algorithmus terminiert, wenn das Lernen keine Verbesserung des Validierungsfehlers innerhalb einer vordefinierten Anzahl von Iterationen bewirkt (vgl. [Goodfellow et al., 2016], S. 239 - 246).

L^2 und L^1 Parameter Regularisierung Die Kapazität eines neuronalen Netzes kann durch das Hinzufügen eines Parameter-Straftermes Ω zu der Kostenfunktion J begrenzt werden:

$$\widetilde{J} = J + \lambda\Omega,$$

wobei als $\widetilde{J}$ die regularisierte Kostenfunktion notiert wird. Der Regularisierungsparameter $\lambda \in [0, \infty)$ bestimmt den relativen Beitrag des Straftermes: Größere λ-Werte führen zur größeren Regularisierung. Die Minimierung der regularisierten Kostenfunktion $\widetilde{J}$ führt zur Minimierung sowohl der ursprünglichen Kostenfunktion J als auch des Straftermes Ω. Das Ziel von L^2 und L^1 Straftermen besteht darin, die Wichtungen des Netzes klein zu halten, denn ein Modell mit großen Wichtungen ist komplexer als ein Modell mit kleinen Wichtungen und neigt daher zum Overfitting. Der Strafterm L^2, auch als *Weight Decay* oder *Ridge Regression* bekannt, und der Strafterm L^1 sind mathematisch wie folgt definiert:

$$\begin{aligned} L^2 &= \frac{1}{2} \sum_{j=1}^{n} W^2, \\ L^1 &= \frac{1}{2} \sum_{j=1}^{n} |W|. \end{aligned}$$

Die Iteration ab dem Parameter $j = 1$ weist darauf hin, dass die Strafterme in der Regel nicht auf Bias-Neuronen angewendet werden (vgl. [Goodfellow et al., 2016], S. 224 -230).

3.1.2.5 Optimierung

Ein gewünschtes Ergebnis des Lernprozesses eines neuronalen Netzes ist die Reduzierung der erwarteten Generalisierungsfehler über alle möglichen Eingabe- und Ausgabedaten:

$$J(W) = \mathbb{E}_{(x,y)\ \rho^{data}} L(f(x; W), y), \tag{3.8}$$

ρ^{data} steht für die datengenerierende Verteilung mit $f(x; W)$ gleich der vorhergesagten Ausgabe für die Eingabe x, y ist die gewünschte Ausgabe, $L(f(x; W), y)$ ist der Fehler zwischen der vorhergesagten und der gewünschten Ausgabe für einen Datensatz.
Da aber ρ^{data} unbekannt ist, wird stattdessen die Kostenfunktion über eine endliche Datenmenge von Trainingsbeispielen reduziert:

$$J(W) = \mathbb{E}_{(x,y)\sim\hat{\rho}^{data}} L(f(x; W), y), \tag{3.9}$$

$\hat{\rho}^{data}$ ist die empirische Verteilung. Gradientenverfahren, die für die Optimierung der Kostenfunktion verwendet werden, können je nach der Art, wie sie den Erwartungswert in (3.9) auflösen, in Stochastische, Mini-Batch und Batch Verfahren aufgeteilt werden (vgl. [Goodfellow et al., 2016], S. 268-269).

Stochastische, Mini-Batch und Batch Gradientenverfahren Gegeben sei eine Trainingsmenge mit N Datensätzen. Jeder Datensatz besteht aus einem Eingabevektor x und einem gewünschten Ausgabevektor y. *Batch Gradientenverfahren* ermitteln den Gradienten der Kostenfunktion als arithmetisches Mittel über alle Trainingsbeispiele:

$$\nabla_W J(W) = \frac{1}{N} \nabla_W \sum_{i=1}^{N} L(f(x^i; W), y^i)$$

Die Durchführung der Batch Verfahren in Deep Learning ist sehr aufwendig, weil dafür die Evaluation der aktuellen Netzparameter für jeden Datensatz der Trainingsmenge, die sehr groß sein kann, nötig ist.
In *Stochastischen Gradientenverfahren* wird der Gradient basierend auf einem einzelnen Trainingsbeispiel geschätzt:

$$\nabla_W J(W) = \nabla_W L(f(x; W), y).$$

Eine solche Schätzung des Gradienten hat eine hohe Varianz. Aus diesem Grund soll die Lernrate in Stochastischen Verfahren klein gewählt werden, um den Lernprozess zu stabilisieren. Eine kleine Lernrate resultiert jedoch im langsamen Lernen und einer hohen Gesamtlaufzeit.
Im Deep Learning werden in der Regel *Mini-Batch Gradientenverfahren* eingesetzt, in denen der Gradient basierend auf n zufälligen Trainingsbeispielen ermittelt wird:

$$\nabla_W J(W) = \frac{1}{n} \nabla_W \sum_{i=1}^{n} L(f(x^i; W), y^i), n < N$$

Der Algorithmus 1 fasst den Mini-Batch Gradientenabstieg-Algorithmus zusammen. Die Lernrate wird normalerweise während des Lernprozesses langsam verkleinert.

Algorithm 1 Mini-Batch Gradientenabstieg nach (Goodfellow et al. 2017)

1: **Parameter** : Learning rate α, initial weigths W
2: **repeat**
3: Sample a mini-batch of n examples $\{x^{(1)}, ..., x^{(n)}\}$ from the training set with corresponding targets $y^{(i)}$
4: Compute gradient estimate $\nabla_W J(W) \leftarrow \frac{1}{n} \nabla_W \sum_{i=1}^{n} L(f(x^i; W), y^i)$
5: Apply update: $W \leftarrow W - \alpha \nabla_W J(W)$
6: **until** stopping criterion not met

Probleme von Gradientenverfahren Gradientenverfahren sind erfolgversprechend, haben aber in der Praxis einige Nachteile (vgl. [Kriesel, 2007], S. 65-66):

1. Häufig konvergieren Gradientenverfahren gegen lokale Minima/Maxima.
   Wenn die zu optimierende Funktion mehrere lokale Optima besitzt, gibt es keine Garantie, dass ein Gradientenverfahren das globale Optimum findet. Die Abbildung 3.6 illustriert eine Funktion mit mehreren lokalen Minima. Die Optimierung der

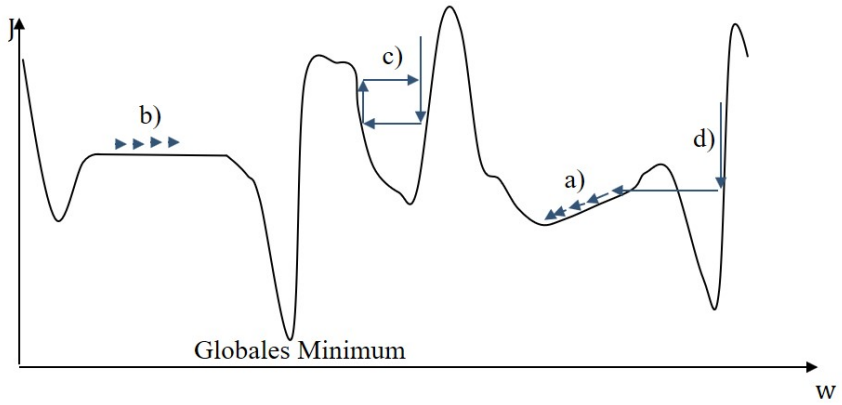


Abbildung 3.6: Mögliche Fehler während eines Gradientenabstiegs: a) Finden eines lokale Minimums b) Quasi-Stillstand bei flachen Plateaus c) Oszillation in Schluchten d) Verlassen guter Minima
Quelle: In Anlehnung an [Kriesel, 2007], S. 66

Funktion mithilfe des Gradientenabstiegs kann dazu führen, dass der Algorithmus gegen ein lokales Minimum konvergiert (Abbildung 3.6a).

2. Flache Plateaus in der Fehleroberfläche können das Training sehr verlangsamen. Beim Durchlauf eines flachen Plateaus wird der Gradient sehr klein, da es kaum Steigung gibt (Abbildung 3.6b). Ein theoretisch möglicher Gradient von Null bringt den Abstieg zum Stillstand.

3. Gute Minima/Maxima können wieder verlassen werden.
   Im Gegensatz zum Gradienten an flachen Plateaus, ist der Gradient an einem steilen Hang sehr groß, sodass ein gutes Minimum/Maximum überschritten werden kann (Abbildung 3.6c).

4. Steile Schluchten in der Fehlerfunktion können Oszillationen hervorrufen (Abbildung 3.6d).

Des Weiteren wird die Leistung der Gradientenverfahren von der Auswahl der Lernrate α stark beeinflusst. Wenn die Lernrate zu groß gewählt ist, kann der Algorithmus gute Minima überschreiten. Wenn die Lernrate zu klein ist, ist der Lernprogress zu langsam (vgl. [Boersch et al., 2007], S. 297).

Adam-Algorithmus Algorithmen mit adaptiven Lernraten ersetzten die globale Lernrate α durch adaptive, für jedes Gewicht individuelle Lernraten. Ein Beispiel dafür ist der Adam-Algorithmus [Kingma & Ba, 2014]. Adam berechnet in jedem Iterationsschritt die Schätzung des ersten und des zweiten Momentum für jede Komponente des Gradienten g. Das erste Momentum s entspricht dem gleitenden Durchschnitt der Gradienten:

$$s = p_1 s + (1 - p_1) g$$

Das zweite Momentum ist der quadrierte Gradient,

$$r = \gamma_2 r + (1 - \gamma_2) g^2,$$

wobei p_1 und p_2 die Senkraten sind. Da im ersten Iterationsschritt s und r mit Null initialisiert werden, werden im Algorithmus die Bias korrigierte Werte $\bar{s}$ und $\bar{r}$ für die Schätzungen der beiden Momenta bestimmt. Die Änderung ΔW und die Aktualisierung der Wichtungen in jedem Iterationsschritt werden wie folgt berechnet:

$$\Delta W = -\alpha \frac{\bar{s}}{\sqrt{\bar{r} + \delta}},$$
$$W \leftarrow W + \Delta W,$$

δ ist eine Konstante zur Stabilisierung des Lernprozesses. Der Algorithmus 4 im Ahnang fasst den Adam Algorithmus zusammen.

3.2 Reinforcement Learning

Reinforcement Learning Verfahren erlauben einem Agenten (dem Lerner), eine Strategie zum Erreichen eines Ziels aus eigener Erfahrung zu erlernen. Dabei ist der Agent in der Lage, die Umgebung wahrzunehmen und durch Ausführung unterschiedlicher Aktionen mit der Umgebung zu interagieren.
In diesem Abschnitt wird zunächst das Reinforcement Learning als ein endlicher Markov-Entscheidungsprozess formalisiert, dann wird eine Übersicht über die Typen der Reinforcement Learning Algorithmen gegeben und abschließend wird das Konzept des Deep Reinforcement Learning vorgestellt.

3.2.1 Markov-Entscheidungsprozess

3.2.1.1 Formalisierung

Ein RL-Problem kann als Markov-Entscheidungsprozess (Eng.: Markov Decision Process, Abk.: MDP) formalisiert werden. Ein MDP beschreibt einen sequenziellen Prozess, bei dem der Agent mit der Umgebung interagiert, um sein Ziel zu erreichen. Ein MDP besteht aus folgenden Elementen (vgl. [Sutton & Barto, 2018], S.47-49):

- Eine Menge von Zuständen $\mathcal{S}$
- Eine Menge von Aktionen $\mathcal{A}$
- Eine Rewardfunktion $R : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$, die den unmittelbaren Reward $r(s, a)$ abhängig von dem aktuellen Zustand s und der Aktion a definiert

- Eine Übergangsfunktion $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0;1]$, die die Wahrscheinlichkeit $p\,(s'\,, r \mid s, a)$ beschreibt, in den Zustand s' zu gelangen und den Reward r abhängig von dem Zustand s und der Aktion a zu bekommen

Die Abbildung 3.7 skizziert die Interaktion zwischen dem Agenten und der Umgebung. Anhand des aktuellen Zustandes $s_t \,\epsilon\, \mathcal{S}$ trifft der Agent zu jedem diskreten Zeitschritt $t = 0, 1, 2, 3, \ldots$ eine Entscheidung über seine nächste Aktion $a_t \,\epsilon\, \mathcal{A}(s)$. Einen Zeitschritt später erhält der Agent ein Feedback $r_{t+1} \,\epsilon\, \mathcal{R} \subset \mathbb{R}$ über die Aktion a_t, das als Reward (dt.: Belohnung) bezeichnet wird, und erfährt den neuen Zustand der Umgebung s_{t+1} (vgl. [Sutton & Barto, 2018], S.47-49).

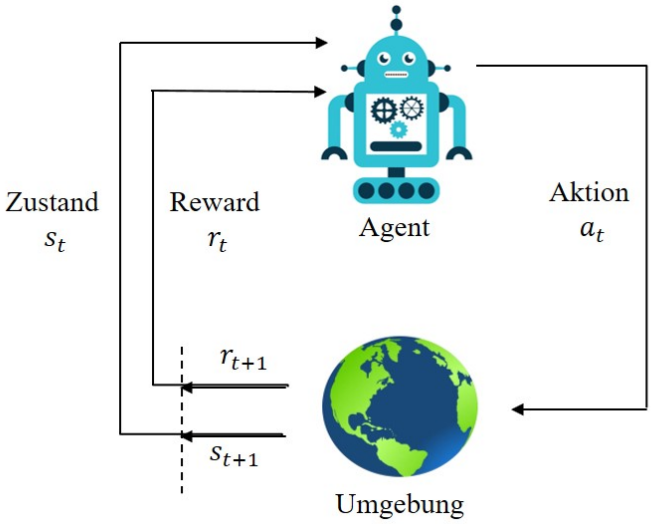


Abbildung 3.7: Interaktion zwischen dem Agenten und der Umgebung in MDP
Quelle: In Anlehnung an [Sutton & Barto, 2018], S.48

.

Durch die Interaktion zwischen dem Agenten und der Umgebung entsteht eine *Trajektorie* $\tau = (a_o, r_1, s_1, a_1, \ldots)$, die die vergangene Erfahrung des Agenten widerspiegelt. Ein MDP erfüllt die Markov-Eigenschaft, laut der „die Zukunft und die Vergangenheit bedingt unabhängig sind, wenn die Gegenwart gegeben ist“ ([Poole & Mackworth, 2017], S. 385). Für einen MDP impliziert die *Markov-Eigenschaft*, dass der Folgezustand s' und der Reward r zum Zeitpunkt $(t+1)$ nur von dem Zustand s und der Aktion a zum Zeitpunkt t abhängen:

$$p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = p(s_{t+1}|s_t, a_t) \text{ und}$$
$$p(r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ \ldots, s_0, a_0) = p(r_{t+1}|s_t, a_t)$$

Zustands- und Aktionsräume können *diskret* und *kontinuierlich* sein. Für die Vorstellung der Kernidee von MDP wird angenommen, dass $\mathcal{S}$ und $\mathcal{A}$ diskret sind, d. h. endlich viele Elemente enthalten.

Umgebung Der Agent nimmt die Umgebung über Sensoren oder symbolische Beschreibungen wahr. Es wird von einem *vollständig beobachtbaren MDP* (Eng.: fully obeservable MDP) gesprochen, wenn der Agent in der Lage ist, alle Eigenschaften der Umgebung zu erfassen. Ist dem Agenten nur ein Ausschnitt der Umgebung sichtbar, wird der MDP als *teilweise beobachtbar* (Eng.: partly obeservable MDP) bezeichnet. Im Rahmen dieser Arbeit gilt die Annahme, dass der MDP vollständig beobachtbar ist.
Die Umgebung kann *deterministisch* oder *stochastisch* sein. Wird die Aktion a in einem Zustand s zu verschiedenen Zeiten in einer deterministischen Umgebung ausgeführt, so sind der Folgezustand und der Reward zwingend identisch.
Als *Modell* der Umgebung wird alles bezeichnet, was der Agent verwenden kann, um die Reaktion der Umgebung auf seine nächste Aktion vorherzusagen. Sind ein Zustand und eine Aktion gegeben, sagt das Modell den nächsten Zustand und den nächsten Reward voraus. Das Modell kann also durch Übergangsfunktion $p\left(s', r|s, a\right)$ beschrieben werden (vgl. [Sutton & Barto, 2018], S. 159-160). In den meisten RL Problemstellungen kennt der Agent nur die Zustandsmenge $\mathcal{S}$ und Aktionsmenge $\mathcal{A}$, wobei das Modell der Umgebung dem Agenten unbekannt ist (vgl. [Poole & Mackworth, 2017], S.550; [Arulkumaran et al., 2017]).

Zustandsbeschreibung Die Zustandsbeschreibung enthält alle Informationen, die der Agent benötigt, um eine Entscheidung über die nächste Aktion zu treffen. Des Weiteren muss der Zustand aufgrund der Markov-Eigenschaft alle für die Entscheidung relevanten Ereignisse aus der früheren Erfahrung des Agenten erfassen, jedoch kompakt bleiben (vgl. [Sutton & Barto, 2018], S. 464-465).

Lernprobleme Die RL-Problemstellungen können episodisch oder kontinuierlich sein. Die *episodischen* Lernprobleme haben einen zeitlich begrenzten Horizont $t = 0, 1, 2, \ldots, T$, wobei T der letzte Zeitschritt ist. Die Sequenz der Interaktionen vom initialen Zeitschritt t_0 bis zum finalen Zeitschritt T wird *Episode* genannt. Ein Beispiel für ein episodisches Lernproblem ist ein Brettspiel. Der zeitliche Horizont kann auch in der Praxis künstlich durch die maximale Episodenlänge begrenzt werden.
Die *kontinuierlichen* Problemstellungen enthalten keinen zeitlich begrenzten Horizont, d. h. $T = \infty$. Ein Beispiel für eine kontinuierliche Problemstellung ist die Regulierung einer Klimaanlage (vgl. [Tokic, 2013], S. 8-9).

3.2.1.2 Ziel und Strategie des Agenten

In MDP spielt die Frage der zu wählenden Strategie (Policy) eine zentrale Rolle. Eine *Policy* beschreibt die Entscheidungen des Agenten über seine nächste Aktion.
Es wird zwischen einer deterministischen und einer stochastischen Policy unterschieden. Eine *deterministische* Policy μ bildet die Zustände auf die Aktionen ab: $\mu(s) = a$. Im Gegensatz dazu weist eine *stochastische* Policy π einem Zustand $s \, \epsilon \, S$ eine Wahrscheinlichkeitsverteilung über alle Aktionen $a \, \epsilon \, A(s)$ zu: $\pi(a|s) = 0...1$.

Die Summe der kumulativen Rewards ab dem Zeitpunkt t wird als *Return* G_t bezeichnet. Der Return G_t gibt den Gewinn des Agenten ab dem Zustand s_t an und kann in einem episodischen Lernproblem wie folgt berechnet werden:

$$G_t = r_{t+1} + r_{t+2} + ... + r_T \tag{3.10}$$

Bei kontinuierlichen Lernproblemen ist die Ermittlung des Returns nach (3.10) aufgrund von $t \rightarrow \infty$ nicht möglich. Um diese Schwierigkeit zu umgehen, kann der Return mittels eines *Diskontierungsfaktors* $\gamma \epsilon [0; 1)$ begrenzt werden:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.11}$$

Der Diskontierungsfaktor bestimmt, wie stark sich der Agent auf zukünftige Rewards konzentriert. Je geringer der Wert von γ ist, desto niedriger werden die zukünftigen Rewards in G_t gewichtet und umso mehr Bedeutung erhält der unmittelbare Reward.
Um eine gemeinsame Notation des Returns sowohl für episodische als auch kontinuierliche Lernprobleme zu bestimmen, können die Gleichungen (3.10) und (3.11) miteinander kombiniert werden:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.12}$$

Dabei sind die beiden Fälle $T = \infty$ und $\gamma = 1$ möglich, dürfen jedoch nicht gemeinsam auftreten (vgl. [Sutton & Barto, 2018], S. 54-56). Der Wert des Returns hängt von der Policy π ab und kann daher ebenso deterministisch oder stochastisch sein (vgl. [Lillicrap et al., 2016], S. 2).
Der Return G_t kann in zwei Komponenten zerlegt werden: den unmittelbaren Reward r_{t+1} und den Return G_{t+1} ab dem nachfolgenden Zustand s_{t+1} (vgl. [Sutton & Barto, 2018], S. 55):

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ ..) \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma^2 r_{t+3} + \ .. \ ) = \\ &= \ r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.13}$$

Das *Ziel* des RL besteht darin, den Erwartungswert des Returns bzgl. der Policy π zu maximieren. Der Agent ist daran interessiert, eine optimale Policy π^* zu finden, die das Erreichen dieses Zieles ermöglicht und dem Agenten somit den größtmöglichen Gewinn ab dem Startzustand s_0 verspricht (vgl. [Silver et al., 2014], S. 2):

$$\pi^* = argmax_\pi \mathbb{E}_\pi \left[G_0 | s_0\right] \tag{3.14}$$

Für einen MDP können mehrere optimale Policies existieren. Im Weiteren werden sie mit π^* referenziert. Die RL-Algorithmen beschreiben, wie sich die Policy des Agenten als Ergebnis seiner Erlebnisse verändert (vgl. [Sutton & Barto, 2018], S.62).

In RL definiert die Rewardfunktion das Ziel des Agenten und beeinflusst erheblich den Lernprozess. Die Rewardfunktion wird durch Entwickler spezifiziert und muss mit Bedacht gewählt werden. In [Sutton & Barto, 2018] wird darauf hingewiesen, dass der Agent durch die Rewardfunktion sein Ziel und nicht den Lösungsweg erfahren soll.
Zum Beispiel kann der Agent bei einem Schachspiel für bestimmte Spielzüge belohnt werden, obwohl das tatsächliche Ziel darin besteht, das Spiel zu gewinnen. In dieser Situation kann der Agent einen Weg finden, einen großen Gesamtreward zu erhalten ohne das Spiel zu gewinnen. Die Rewardfunktion, bei der der Agent auch für Zwischenschritte belohnt oder bestraft wird, ist im Folgenden als *Shaped-Rewardfunktion* referenziert.
Beim Schachspiel kann die Rewardfunktion auch so spezifiziert werden, dass der Agent nur am Ende des Spiels eine Belohnung für den Sieg oder eine Bestrafung für den Misserfolg erhält. Eine solche Konfiguration der Rewardfunktion ist in RL erwünscht, kann aber in der Praxis zu einem langsamen Lernprozess führen. Die Rewardfunktion, mit der der Agent nur am Ende der Episode einen Reward bekommt, wird im Folgenden *Sparse-Rewardfunktion* genannt.

3.2.1.3 Wertefunktionen

Eine Wertefunktion gibt an, *wie wertvoll* es für den Agenten ist, sich in dem Zustand s zu befinden oder eine Aktion a in s auszuführen. Das Wort „wertvoll“ bezieht sich auf den Gewinn des Agenten.
Die *Zustandswertefunktion* $v^\pi(s) : S \to \mathbb{R}$ gibt den langfristigen Wert des Zustandes s an und berechnet dafür den erwarteten Return beim Folgen von π, beginnend im Zustand s_t:

$$v^\pi(s_t) \doteq \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t\right] \tag{3.15}$$

Die *Wertefunktion für Zustand-Aktion-Paare* $q^\pi(s,a) : S \times A \to \mathbb{R}$ (Abk.: Q-Funktion) gibt den Wert eines Zustand-Aktion-Paares (s,a) bzgl. der Policy π an. Die Q-Funktion beschreibt den erwarteten Return beim Folgen von π, beginnend im Zustand s_t mit der Aktion a_t (vgl. [Sutton & Barto, 2018], S. 58):

$$q^\pi(s_t,a_t) \doteq \mathbb{E}_\pi\left[G_t|\, s_t, a_t\right] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t,a_t] \tag{3.16}$$

Die Zustandswertefunktion $v^\pi(s)$ kann aus der Q-Funktion abgeleitet werden (vgl. [Levine, 2019b], S. 27):

$$v^\pi(s) \doteq \mathbb{E}_{a\backsim\pi(a|s)}[q^\pi(s,a)] \tag{3.17}$$

Das Ziel des RL (s. 3.14) in einem episodischen MDP kann mithilfe von $v^\pi(s)$ und $q^\pi(s,a)$ wie folgt formuliert werden (vgl. [Levine, 2019b], S. 26-27):

- die Maximierung des Erwartungswertes $\mathbb{E}_{s_0\sim\rho^\pi(s_0)}\left[v^\pi(s_0)\right]$ ab dem Startzustand s_0,

der zufällig aus der Start-Zustandsverteilung $\rho^{\pi}(s_0)$ ausgewählt wurde,

- die Maximierung des Erwartungswertes $\mathbb{E}_{s_0 \sim \rho^{\pi}(s_0)} \left[\mathbb{E}_{a_0 \sim \pi(a_0|s_0)} [q^{\pi}(s_0, a_0)) \right]$ des Zustand-Aktion-Paares (s_0, a_0).

Bellman Gleichungen Ähnlich wie der Return können die Wertefunktionen rekursiv definiert werden. Der Zusammenhang zwischen dem Wert $v^{\pi}(s)$ des Zustandes s und dem Wert des Folgezustandes $v^{\pi}(s')$ ist durch die *Bellman Gleichung für* $v^{\pi}(s)$ gegeben (vgl. [Sutton & Barto, 2018], S. 59):

$$
\begin{aligned}
v^{\pi}(s) \doteq \mathbb{E}_{\pi}[G_t | s_t = s] &= && \text{nach (3.13)} \\
= \mathbb{E}_{\pi}[r_{t+1} + \gamma G_{t+1} | s_t = s] = &= && \text{nach (3.15)} \\
= \mathbb{E}_{\pi}[r_{t+1} + \gamma v^{\pi}(s_{t+1}) | s_t = s] = &= && \text{Erwartungswert auflösen} \\
= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) r + \mathbb{E}_{\pi}[G_{t+1} | s_{t+1} = s'] &= \\
= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \ [r + \gamma v^{\pi}(s')] && && (3.18)
\end{aligned}
$$

Die Abbildung 3.8a dient zur Verdeutlichung der Herleitung von (3.18): Die hellen Kreise repräsentieren die Zustände, die schwarzen Kreise stellen die Aktionen dar. Der Agent befindet sich im Zustand s und kann eine der in s möglichen Aktionen ausführen. Der Agent wählt die Aktion a nach π aus und führt sie aus, bekommt den Reward r und erfährt den neuen Zustand s'. Für die Ermittlung von $v^{\pi}(s)$ wird der Term $[r + \gamma v^{\pi}(s')]$ um die Wahrscheinlichkeit $\pi(a|s) p(s', r|s, a)$ des Tripels (a, r, s') gewichtet und über alle möglichen Werte der drei Variablen a, r, s' aufsummiert (vgl. [Sutton & Barto, 2018], S.59).

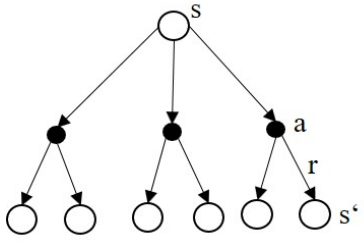


Abbildung 3.8: Backup Diagramm für $v^{\pi}(s)$.
Quelle: In Anlehnung an [Sutton & Barto, 2018], S.59.

Die *Bellman Gleichung für* $q^{\pi}(s, a)$ ist (vgl. [Sutton & Barto, 2018], S. 78):

$$
\begin{aligned}
q^{\pi}(s, a) \doteq \mathbb{E}_{\pi}[r_{t+1} + \gamma v_{\pi}(s_{t+1}) | s_t = s, \ a_t = a] = \\
= \sum_{s', r} p(s', r|s, a) \ [r + \gamma v_{\pi}(s')] && (3.19)
\end{aligned}
$$

Optimale Wertefunktionen und Bellman Optimalitätsgleichungen Die Wertefunktionen können bei der Suche nach einer optimalen Policy hilfreich sein und werden daher in

verschiedener Weise in fast allen RL-Algorithmen eingesetzt, u. a. für den Leistungsvergleich der unterschiedlichen Policies.
Eine Policy π_1 ist *besser* als eine Policy π_2 (oder *genauso gut* wie π_2), wenn der erwartete Gewinn durch π_1 größer oder gleich dem erwarteten Gewinn durch π_2 ist, d. h., unter der Policy π_1 erreicht die Zustandswertefunktion in jedem Zustand s einen größeren Wert als unter π_2 (oder den gleichen Wert wie unter π_2):

$$\pi_1 \geq \pi_2 \iff v_{\pi_1}(s) \geq v_{\pi_2}(s) \quad \forall s \, \epsilon \, \mathcal{S}$$

Die optimale Policy π^* besitzt die maximale Zustandswertefunktion $v^*(s)$ über alle möglichen Policies, die als *optimale Zustandswertefunktion* $v^*(s)$ bezeichnet wird (vgl. [Sutton & Barto, 2018], S. 49):

$$v^*(s) = max_\pi v^\pi(s)$$

Die folgende Abbildung stellt die Relation $\pi^* \geq \pi_1 \geq \pi_2$ grafisch dar.

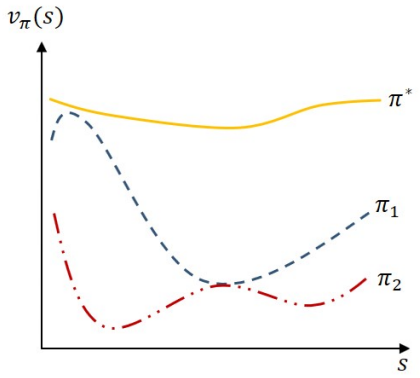


Abbildung 3.9: Darstellung der Relation $\pi^* \geq \pi_1 \geq \pi_2$ zwischen der optimalen Policy π_*, der Policy π_1 und der Policy π_2
Quelle: In Anlehnung an [White & White, 2019a]

Die maximale Q-Funktion über alle möglichen Policies wird als *optimale Wertefunktion für Zustand-Aktion-Paare* $q^*(s, a)$ bezeichnet:

$$q^*(s, a) = max_\pi q^\pi(s, a)$$

Die Bellman Gleichung für die optimale Zustandswertefunktion ist als *Bellman Optimalitätsgleichung für* $v^*(s)$ bekannt:

$$v^*(s) = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s)] \tag{3.20}$$

Im Gegensatz zu (3.18) entfällt in (3.20) die Referenz auf eine Policy. Stattdessen wird die Aktion a ausgewählt, die den Zustandswert $v^*(s)$ maximiert.

Die *Bellman Optimalitätsgleichung für* $q^*(s,a)$ ist

$$q^*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma max_{a'} q^*(s',a')] \tag{3.21}$$

Wenn der Agent sich im Zustand s befindet und die Aktion a ausführt, steht der Term $max'_a q^*(s',a')$ für den maximalen Q-Wert über den Folgepaaren (s',a'). Die Abbildung 3.10 verdeutlicht die Bellman Optimalitätsgleichungen für $v^*(s)$ und $q^*(s,a)$.

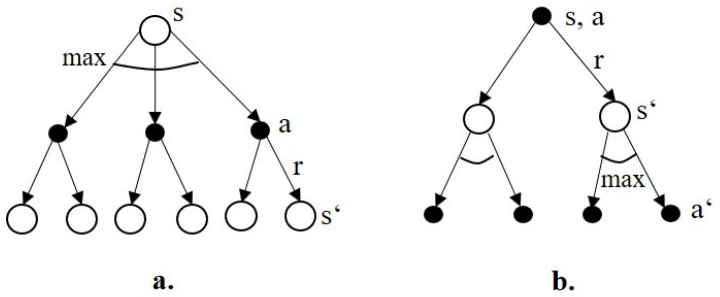


Abbildung 3.10: Backup Diagramme für a) $v^*(s)$ und b) $q^*(s,a)$
Quelle: In Anlehnung an [Sutton & Barto, 2018], S.64

Für einen endlichen MDP haben die Bellman Optimalitätsgleichungen genau eine Lösung. Wenn die Übergangsfunktion bekannt ist, kann eine Bellman Optimalitätsgleichung als ein nichtlineares Gleichungssystem mit n unbekannten, $n = |\mathcal{S}|$, mithilfe von z. B. dynamischen Programmierung gelöst werden (vgl. [Sutton & Barto, 2018], S. 64). Da das Modell der Umgebung in den meisten RL Problemstellungen unbekannt ist, werden Wertefunktionen in RL-Algorithmen aus der Erfahrung des Agenten geschätzt.
Sobald eine der optimalen Wertefunktionen bekannt ist, kann der Agent die optimale Policy bestimmen (vgl. [Sutton & Barto, 2018], S. 64-65):

- $v^*(s)$ ist bekannt: Für jeden Zustand s gibt es eine oder mehrere Aktionen a, bei denen das Maximum für $v^*(s)$ in der Bellman Optimalitätsgleichung erhalten wird. Eine Policy, die nur diesen Aktionen eine Wahrscheinlichkeit ungleich Null zuweist, ist eine optimale Policy:

  $$\pi^* = argmax_a \sum_{s',r} p(s',r|s,a)\,[r + \gamma v^*(s')] \tag{3.22}$$

  Die Gleichung (3.22) verdeutlicht, dass für die Bestimmung der optimalen Policy anhand von $v^*(s)$ u. a. die Übergangswahrscheinlichkeiten $p(s',r|s,a)$ bekannt sein müssen.

- $q^*(s,a)$ ist bekannt: Um eine optimale Policy aus $q^*(s,a)$ abzuleiten, soll der Agent in jedem Zustand s eine Aktion a auswählen, die den maximalen Wert der Q-Funktion für (s,a) ergibt:

  $$\pi^* = argmax_a q^*(s,a) \tag{3.23}$$

Aus (3.23) folgt, dass die optimale Policy ohne Kenntnisse über das Modell der Umgebung aus $q^*(s, a)$ abgeleitet werden kann. Wenn also das Modell unbekannt ist, ist das Ziel $q^*(s, a)$ zu bestimmen, äquivalent dem Ziel π^* zu finden.

3.2.2 Typen von Reinforcement Learning Algorithmen

Je nach zugrunde liegendem Kriterium können die RL-Algorithmen in unterschiedliche Klassen aufgeteilt werden.

3.2.2.1 On-policy- und off-policy-Algorithmen

In *off-Policy*-Algorithmen wird zwischen zwei folgenden Policies unterschieden: (1) Behavior-Policy β, welcher der Agent während der Interaktion mit der Umgebung folgt und (2) Target-Policy π, die anhand der Erfahrung, die durch die Interaktion mit der Umgebung nach β gesammelt wurde, verbessert wird. Dabei sind die Behavior-Policy und die Target-Policy unterschiedlich: $\beta \neq \pi$.
In *on-Policy* Algorithmen gilt $\beta = \pi$. Grundsätzlich konvergieren die on-Policy-Algorithmen schneller als off-Policy-Algorithmen (vgl. [Sutton & Barto, 2018], S. 103). Jedoch ist der Einsatz der on-Policy-Algorithmen in Anwendungsfällen, für die keine Simulation existiert, problematisch. Der Grund dafür ist die Evaluation von π, die basierend auf der Erfahrung des Agenten stattfindet, die nach π gesammelt wurde. Das heißt, jede Änderung in der Policy führt dazu, dass der Agent neue Erfahrung mit der geänderten Policy sammeln muss, was zeit- und kostenaufwendig ist (vgl. [Levine, 2019b], S. 39). Die off-Policy-Algorithmen sind dagegen in der Lage, aus Daten zu lernen, die von der Interaktion mit der Umgebung nach einer beliebigen Policy β stammen, wie z. B. von der alten Erfahrung des Agenten oder von der Erfahrung eines Experten (vgl. [Sutton & Barto, 2018], S. 103).
Zur Evaluation der Target-Policy verwenden viele off-Policy-Algorithmen die *Importance Sampling* (Abk.: IS), eine Technik zur Schätzung eines Erwartungswertes unter einer Verteilung anhand der Daten aus einer anderen Verteilung. Dabei gilt die folgende Annahme: $\pi(a|s) > 0 \implies \beta(a|s) > 0$ (*assumption of coverage*).
Wenn der Agent sich zum Zeitpunkt t im Zustand s_t befindet, kann die Policy π mittels $v_\pi(s_t) = \mathbb{E}_{\tau_t \sim \pi}[G_t]$ evaluiert werden; $\tau_t = (a_t, s_{t+1}, a_{t+1}, ..., s_T)$ ist die Trajektorie, die ab dem Zeitpunkt t entsteht, T ist der letzte Zeitschritt. Da der Agent die Daten zur Verfügung hat, die durch die Ausführung der Policy β gesammelt worden sind, kann $v_\pi(s_t)$ nicht direkt ermittelt werden.
Als *Importance Sampling Ratio* wird die relative Wahrscheinlichkeit einer Trajektorie unter der Target-Policy π und der Behavior-Policy β bezeichnet:

$$\rho_{IS,t} = \frac{p(\tau_t|\pi\ )}{p(\tau_t|\beta\ )} = \frac{\prod_{k=t}^{T-1} \pi(a_k|s_k)p(s_{k+1}|s_k, a_k)}{\prod_{k=t}^{T-1} \beta(a_k|s_k)p(s_{k+1}|s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k|s_k)}{\beta(a_k|s_k)}, \tag{3.24}$$

Mit p sind in der Gleichung (3.24) die Übergangswahrscheinlichkeiten notiert. Da die Übergangswahrscheinlichkeiten ein Teil der Umgebung und daher von der Policy unabhängig sind, können sie verkürzt werden. Die IS Ratio ρ_{IS} ermöglicht es, den erwarteten Return ab dem Zustand s_t unter der Policy π aus dem unter der Policy β erworbenen Return zu schätzen (vgl.[Sutton & Barto, 2018], S. 103-105):

$$V_{IS_\pi}(s_t) = \mathbb{E}_{\tau_t \sim \beta}[\rho_{IS,t} G_t] \tag{3.25}$$

V_{IS_π} schätzt den Return ab dem Zustand s_t unter π ohne Bias, kann aber eine große Varianz aufweisen, die exponentiell mit der Größe der Trajektorie wächst. Der Erwartungswert in (3.25) kann aus der Erfahrung des Agenten ermittelt werden, wenn der Zustand s_t N-mal besucht wird:

$$V_{IS_\pi}(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \rho_{IS,t} G_t$$

3.2.2.2 Modellfreie und modellbasierte RL-Algorithmen

Abhängig davon, ob ein RL-Algorithmus das Modell der Umgebung einbezieht, können die RL-Algorithmen in modellfreie und modellbasierte Algorithmen aufgeteilt werden.

Modellfreie Algorithmen Modellfreie Algorithmen benötigen für ihre Ausführung kein Modell und lernen eine Policy (direkt oder indirekt) aus der realen Erfahrung des Agenten. Je nachdem, wie die Suche nach einer optimalen Policy durchgeführt wird, können die modellfreien Algorithmen in drei Gruppen unterteilt werden: wertebasierte, policybasierte und Actor-Critic Verfahren. Die Abbildung 3.11 fasst die Taxonomie der modellfreien RL-Algorithmen zusammen. Die gelben Rechtecke stellen Algorithmenklassen dar und in den blauen Rechtecken sind spezifische Beispiele für Algorithmenklassen angegeben.

Wertebasierte Algorithmen

Das Kernelement der wertebasierten Algorithmen ist die Schätzung der Wertefunktionen (v^π oder q^π), aus der abschließend die optimale Policy π^* abgeleitet wird. Wie bereits oben erwähnt, kann die optimale Policy aus der optimalen Q-Funktion ohne Kenntnisse über das Modells abgeleitet werden. Aus diesem Grund wird in modellfreien wertebasierten Algorithmen die optimale Q-Funktion approximiert.
Die Grundlage für wertebasierte Verfahren stellt der *Generalized Policy Iteration (GPI)* Algorithmus, der aus zwei wechselwirkenden Prozessen besteht: (1) Policy Evaluation und (2) Policy Iteration.
Policy Evaluation befasst sich mit der Schätzung der Q-Funktion bzgl. der aktuellen Policy π. Die Aufgabe der *Policy Improvement* ist es, eine neue Policy π' zu entwickeln, die die ursprüngliche Policy π verbessert, indem sie sich gierig (Eng.: greedy) in Bezug auf

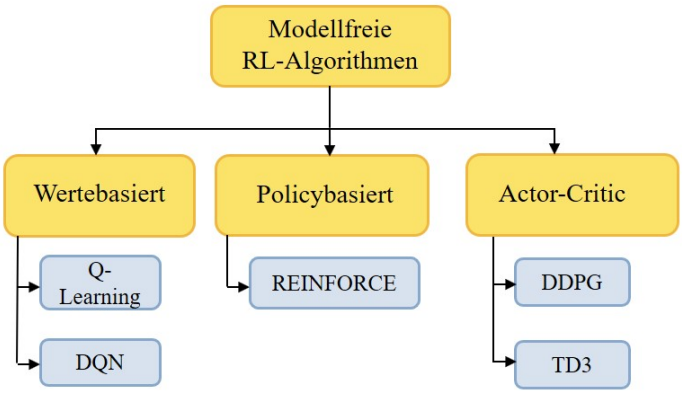


Abbildung 3.11: Taxonomie der modellfreien RL-Algorithmen
Quelle: In Anlehnung an [OpenAI Spinning Up, 2018]

die aktuelle Schätzung der Q-Funktion $Q(s, a)$ verhält. Das bedeutet, die neue Policy π' wählt in jedem Zustand s die Aktion a aus, die den maximalen Wert des Zustand-Aktion-Paares (s, a) ergibt:

$$\pi'(a|s) = \begin{cases} 1, & \text{wenn} \quad a = argmax_a Q(s, a) \\ 0, & \text{sonst} \end{cases} \tag{3.26}$$

Eine solche Policy $\pi'(a|s)$ wird *greedy-Policy* und die Aktion a entsprechend *greedy-Aktion* genannt.
Wenn die beiden Prozesse keine Veränderungen mehr aufweisen, d. h.

- die Policy Evaluation keine Veränderung der Schätzung der Wertefunktion mehr hervorruft: $Q_{t+1}(s, a) = Q_t(s, a) \pm \Theta$, wobei Θ ein kleiner Schwellenwert ist, und
- die Policy Improvement die Policy nicht verändert, $\pi'(a|s) = \pi(a|s)$,

müssen die Wertefunktion und die Policy optimal sein (vgl. [Sutton & Barto, 2018], S. 86-87).
In einem einfachen Fall, wenn die Anzahl der Zustände und Aktionen gering ist, kann der Agent die aktuelle Schätzung der Q-Funktion in einer Lookup-Tabelle verwalten. Dabei entsprechen die Zeilen der Lookup-Tabelle den möglichen Zuständen und die Spalten den möglichen Aktionen des Agenten. Die Einträge der Tabelle sind approximierte Q-Werte für jedes Zustand-Aktion-Paar (s. Tabelle 3.1). Jede Aktualisierung von Q ändert genau einen Eintrag in der Tabelle (vgl. [Sutton & Barto, 2018], S. 199). Obwohl eine Lookup-Tabelle ein einfaches und transparentes Modell zur Darstellung der aktuellen Schätzung der Q-Funktion anbietet, ist diese Methode aufgrund des Speicherplatzverbrauchs in der Praxis oft nicht realisierbar. Mit jeder zusätzlichen Dimension des Zustands- oder Aktionsraums wächst die Tabelle exponentiell. Ein weiterer Nachteil der tabellarischen Methode ist die fehlende Generalisierungsfähigkeit.
Um die Schätzung der Q-Funktion kompakt zu repräsentieren, kann statt einer Lookup

	a_1	a_2
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$

Tabelle 3.1: Beispiel einer Lookup-Tabelle für die Verwaltung der Schätzung der Q-Funktion in einem Lernproblem mit 3 Zuständen und 3 Aktionen
Quelle: Eigene Darstellung

Tabelle eine Funktion $Q^W(s, a)$ verwendet werden. Die Ausgabe von $Q^W(s, a)$ wird durch den Parametervektor W angepasst. Jede Änderung in W beeinflusst geschätzte Q-Werte von mehreren Zustand-Aktion-Paaren. Für die Bestimmung der optimalen Parameter kann ein SL-Algorithmus, wie z. B. Deep Learning oder ein Entscheidungsbaum, verwendet werden. Der SL-Algorithmus soll die folgenden Anforderungen erfüllen:

- eine gute Balance zwischen der Generalisierung und Diskriminierung der Zustände bzw. der Zustand-Aktion-Paare leisten (vgl. [White & White, 2019b]),
- eine Fähigkeit besitzen, anhand der Zielwerte zu lernen, die sich über die Zeit ändern, in GPI wird bspw. die Q-Funktion approximiert, während π sich nach jeder Iteration ändert (vgl. [Sutton & Barto, 2018], S. 198).

Aufgrund der gierigen Verbesserung der Policy, wie es GPI nach der Formel (3.26) erfordert, wächst die Rechnerkomplexität der wertebasierten Algorithmen linear mit $|\mathcal{A}|$. Diese Algorithmen sind deswegen nur in diskreten Aktionsräumen anwendbar [Dulac-Arnold et al., 2015]. Die policybasierten Algorithmen können dagegen effektiv in hochdimensionalen und kontinuierlichen Aktionsräumen eingesetzt werden (vgl. [Silver, 2015], S. 5).

Policybasierte Algorithmen

Die policybasierten Algorithmen lernen direkt eine Policy $\pi_\theta(a|s)$, die durch den Vektor θ parametrisiert ist. Das Ziel der policybasierten Algorithmen besteht darin, die Leistung $J(\theta)$ der Policy π_θ zu maximieren. Gesucht ist das globale Maximum, also der Parametervektor $\theta^* = arg\ max_\theta\ J(\theta)$ für alle möglichen θ. Zur Policyoptimierung können Gradientenverfahren oder gradientfreie Methoden, wie z. B. genetische Algorithmen oder Simplex-Verfahren, eingesetzt werden (vgl. [Silver, 2015], S. 11).
Zur Aktualisierung des Parametervektors θ werden in policybasierten Algorithmen Wertefunktionen eingesetzt. Dabei wird der Wert eines Zustand-Aktion-Paares direkt aus der Erfahrung des Agenten geschätzt, wie bspw. im REINFORCE-Algorithmus [Williams, 1992].

Actor-Critic Algorithmen

Die RL-Algorithmen, die gleichzeitig eine Policy und eine Wertefunktion erlernen, werden

als *Actor-Critic-Algorithmen* bezeichnet. Diese Algorithmen beinhalten zwei gleichbedeutende Komponenten:

- die Critic-Komponente, die sich mit der Approximation der Q-Funktion befasst und
- die Actor-Komponente, die die Parameter θ der Policy π_θ in die von der Critic-Komponente vorgeschlagene Richtung anpasst.

Modellbasierte Algorithmen Die modellbasierten Algorithmen setzen voraus, dass für die Bestimmung einer optimalen Policy ein Modell der Umgebung gegeben ist bzw. erlernt werden soll. Im Gegensatz zu den modellfreien Algorithmen, die danach streben, die optimale Policy (direkt oder indirekt) zu *erlernen*, besteht die Grundidee von modellbasierten Algorithmen darin, die optimale Policy anhand eines Modells zu *planen* (vgl. [Sutton & Barto, 2018], S. 139). In der wissenschaftlichen Literatur lassen sich eine Reihe von Methoden zur Approximation des Modells finden: künstliche neuronale Netze [Nagabandi et al., 2017], Gaußsche Prozesse [Deisenroth & Rasmussen, 2011], lineare Modelle [Costa et al., 2015] und weitere.
Modellbasierte Algorithmen sind potenziell effizienter als die modellfreien Algorithmen, da das Modell zur Simulation der Erfahrung benutzt werden kann. Die Planung von Aktionen wird jedoch für den Agenten problematisch, wenn das Modell inkorrekt ist. Empirische Versuche zeigen, dass die Fehler im Modell zu einer schlechten oder einer suboptimalen Policy führen können [Deisenroth et al., 2013].

3.2.3 Q-Learning

Der Q-Learning-Algorithmus wurde im Jahr 1989 entwickelt und zählt zu den ersten Durchbrüchen in der RL-Forschung. Der Kern des Algorithmus ist die Schätzung der optimalen Q-Funktion $q^*(s, a)$ mithilfe des *Temporal Difference Learning* (Abk.: TD-Learning). Q-Learning ist ein off-Policy modellfreier Algorithmus.

Temporal Difference Learning Gegeben ist eine Zeitreihe der Beobachtungen $x_1, ..., x_t$. Zum Zeitpunkt t soll eine Vorhersage X_t über den nächsten Wert der Zeitreihe x_{t+1} getroffen werden. TD-Learning erlaubt es, den Schätzwert X_t als geglätteter Mittelwert aus dem aktuellen Zeitreihenwert x_t und der bisherigen Schätzung des Mittelwertes X_{t-1}, zu berechnen:

$$\begin{aligned} X_t &= (1 - \alpha) \cdot X_{t-1} + \alpha \cdot x_t = \\ &= X_{t-1} + \alpha \cdot (x_t - X_{t-1}), \end{aligned} \tag{3.27}$$

$\alpha \epsilon (0; 1]$ ist ein Glättungsfaktor. Die Differenz zwischen dem tatsächlichen Zeitreihenwert x_t und dem im Zeitpunkt $(t-1)$ geschätzten Wert von x_t wird als TD-Fehler δ_t bezeichnet:

$$\delta_t = x_t - X_{t-1} \tag{3.28}$$

Um die neue Schätzung des Mittelwertes X_t zu ermitteln, wird also laut (3.27) die alte Schätzung X_{t-1} um das α-fache des TD-Fehlers aktualisiert (vgl. [Poole & Mackworth, 2017], S. 554).

1-step Q-Learning Die aktuelle Schätzung $Q(s, a)$ der optimalen Q-Funktion wird im Q-Learning-Algorithmus in einer Lookup-Tabelle verwaltet. Am Anfang des Lernens werden alle Q-Werte mit Zufallszahlen initialisiert. Der Agent startet die Interaktion mit der Umgebung im Zustand $s_t = s_0$, indem er die Aktion $a_t = a_0$ nach Policy π ausführt. Im nächsten Zeitschritt bekommt der Agent den Reward $r_{t+1} = r_1$ und findet sich im Zustand $s_{t+1} = s_1$. Das 4-Tupel $(s_t, a_t, r_{t+1}, s_{t+1})$ wird als *Transition* bezeichnet.
Im Zeitschritt $(t+1)$ aktualisiert der Agent die Schätzung des Q-Wertes $Q(s_t, a_t)$ wie folgt:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \cdot max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)], \tag{3.29}$$

das Symbol $\alpha \epsilon (0; 1]$ steht für die Lernrate (vgl. Glättungsfaktor in TD-Learning). Die Idee der Updateregel 3.29 besteht darin, $Q(s_t, a_t)$ in Richtung des Zielwertes $y_t = [r_{t+1} + + \gamma \cdot max_{a'} q^*(s_{t+1}, a')]$ (s. Bellman Optimalitätsgleichung, Formel 3.21) zu aktualisieren. Da die echte optimale Q-Funktion unbekannt ist, wird stattdessen in y_t die bisherige Schätzung der Q-Funktion verwendet (s. TD-Learning, Formel 3.27).
TD-Fehler δ_t wird für 1-step Q-Learning wie folgt berechnet:

$$\delta_t = r_{t+1} + \gamma \cdot max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \tag{3.30}$$

Wenn alle Paare (s, a) unendlich oft besucht werden, konvergiert $Q(s, a)$ mit einer Wahrscheinlichkeit 1 gegen $q^*(s, a)$. Die optimale Policy lässt sich dann aus $Q(s, a)$ ableiten, indem der Agent in jedem Zustand die greedy-Aktion $a = argmax_a Q(s, a)$ auswählt (s. Formel 3.26).
Der Algorithmus 2 stellt den Pseudocode für den 1-step Q-Learning Algorithmus dar. Q-Learning ist ein off-Policy Algorithmus. Um eine Erfahrung zu sammeln, verwendet der Agent eine Behavior-Policy (Zeile 6 des Algorithmus 2). Die Behavior-Policy unterscheidet sich von der Target-Policy des Agenten, der greedy-Policy (Zeile 8 des Algorithmus 2) (vgl. [White & White, 2019c]).

Algorithm 2 1-step Q-Learning nach [Sutton & Barto, 2018]

Parameter : Learning rate α, discount rate γ
Initialize $Q(s,a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily except $Q(terminal,-)=0$
for each episode **do**
  Initialize s
  repeat
    Choose a from s using policy derived from Q (e.g. ϵ-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  until s is terminal
end for

Ein Nachteil des 1-step Q-Learning Algorithmus besteht darin, dass der Reward r_{t+1} nur den Q-Wert des Zustand-Aktion-Paares beeinflusst, der zur Beschaffung von r_{t+1} führte, also $Q(s_t, a_t)$. Die Q-Werte der anderen Zustand-Aktion-Paare werden von r_{t+1} nur indirekt durch den aktualisierten Q-Wert des Paares (s_t, a_t) beeinflusst. Der Lernprozess kann dadurch verlangsamt werden, da viele Updates der Q-Funktion erforderlich sind, um die Belohnung r_{t+1} an die relevanten vorherigen Zustände und Aktionen zu propagieren (vgl. [Mnih et al., 2016], S. 2-3).

n-step Q-Learning In n-step Q-Learning wird die Schätzung der Q-Funktion erst nach n Transitionen aktualisiert. Als Zielwert für das Update gilt der *n-step Return* $G_{t:t+n}$, wobei statt der tatsächlichen optimalen Q-Funktion deren bisherige Schätzung verwendet wird (vgl. [Hernandez-Garcia & Sutton, 2019], S. 2-3; [Sutton & Barto, 2018], S. 142-145):

$$G_{t:t+n} = \begin{cases} r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \ max_{a_{t+n}} \gamma^n Q(s_{t+n}, a_{t+n}), & \text{falls } t+n < T \\ G_t, & \text{sonst} \end{cases} \tag{3.31}$$

Die Aktualisierung von $Q(s_t, a_t)$ wird dann wie folgt durchgeführt:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [G_{t:t+n} - Q(s_t, a_t)] \tag{3.32}$$

Trade-off zwischen Exploitation und Exploration Die Behavior-Policy beeinflusst den Lernprozess, indem sie teilweise bestimmt, welche Zustand-Aktion-Paare der Agent bei der Interaktion mit der Umgebung besucht. Der Q-Learning Algorithmus gibt jedoch keine Anweisungen dazu, wie die Behavior-Policy aussehen soll. Im Allgemeinen kann sich der Agent wie folgt verhalten:

- das Wissen über die Q-Funktion verwenden und in einem Zustand s eine greedy-Aktion in Bezug auf $Q(s,a)$ auswählen (*Exploitation*) oder

- die Umgebung erkunden und bis dahin unbekannte Zustand-Aktion-Paare (s, a) besuchen (*Exploration*).

Die Exploitation erlaubt dem Agenten, den unmittelbaren Reward r_{t+1} zu maximieren. Durch die Exploration kann der Agent aber eine Aktion entdecken, die auf lange Sicht zu einem größeren gesamten Reward führt (vgl. [Sutton & Barto, 2018], S. 26). Der Trade-off zwischen Exploitation und Exploration ist eine zentrale Frage in RL. Es gibt eine Reihe von Ansätzen, wie Exploitation und Exploration ausgeglichen werden können. Eine oft dafür verwendete Methode ist die *ε-greedy Policy*: Mit der Wahrscheinlichkeit ε wählt der Agent im Zustand s eine zufällige Aktion aus und mit der Wahrscheinlichkeit $(1 - \varepsilon)$ die greedy-Aktion. Der Wert von ε nimmt mit der Zeit ab, was dazu führt, dass am Anfang des Lernens der Agent viel exploriert und sich mit der Zeit immer häufiger für die Aktionen entscheidet, die den maximalen Q-Wert ergeben (vgl. [Poole & Mackworth, 2017], S. 558).

3.2.4 Gradientenverfahren in wertebasierten Algorithmen

Gegeben ist die Schätzung der optimalen Q-Funktion $Q(s, a; W)$ unter der Policy π, die durch eine Funktion mit den Parametern W repräsentiert ist. Es liegen Daten $\mathcal{D}$ aus der Interaktion mit der Umgebung nach der Policy β vor. Es wird angenommen, dass die Zustandsverteilung in den Daten $\rho^{\beta}(s)$ der Zustandsverteilung $\rho^{\pi}(s)$ entspricht. Gesucht ist der Parametervektor W^*, der die folgende Kostenfunktion minimiert:

$$L(W) = \mathbb{E}_{s\sim\rho^{\beta}}[(q^{\pi}(s,a) - Q(s,a;W))^2]$$

Der Term $[(q^{\pi}(s, a) - Q(s, a; W))^2]$ misst die quadrierte Abweichung des tatsächlichen Q-Wertes von dem geschätzen Q-Wert.
Der Parametervektor W kann mithilfe des SGD-Algorithmus iterativ aktualisiert werden:

$$W_{t+1} \leftarrow W_t - \nabla_{W_t} L(W_t) \tag{3.33}$$

Es gibt eine Reihe von Möglichkeiten, $q^{\pi}(s, a)$ aus der Erfahrung des Agenten $\mathcal{D}$ zu schätzen. Im bereits vorgestellten Q-Learning Algorithmus wird bspw. $q^{\pi}(s, a)$ mithilfe von TD geschätzt (*TD Policy Evaluation*). Alternativ kann als Schätzer für $q^{\pi}(s, a)$ der Return G_t verwendet werden (*Monte Carlo Policy Evaluation*) (vgl. [Sutton & Barto, 2018], S. 199-202).
TD und Monte Carlo haben unterschiedliche Zielfunktionen und je nach Problemstellung kann jeder der beiden Schätzer vorteilhaft sein. Grundsätzlich hat aber der Monte Carlo Schätzer eine größere Varianz als der TD-Schätzer (vgl. [Penedones et al., 2019], S. 3).
In 1-step Q-Learning mit einer Funktionsapproximation kann der Gradient der Kostenfunktion wie folgt bestimmt werden:

$$\nabla_{W_t} L(W_t) = -2 \cdot [r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a'; W_t) - Q(s_t, a_t; W_t)] \nabla_{W_t} Q(s_t, a_t; W_t) \tag{3.34}$$

Ein besonderes Merkmal des Gradienten nach der Gleichung (3.34) ist die Abhängigkeit des Zielwertes $[r_{t+1} + \gamma \cdot \max'_a Q(s_{t+1}, a'; W_t)]$ vom aktuellen Wert des Parametervektors W,die jedoch nicht berücksichtigt wird. Aus diesem Grund wird in manchen Quellen der Gradient nach (3.34) als *Semi-Gradient* bezeichnet.
Bei einer nicht-linearen Approximation, wie bspw. mithilfe eines Deep Neuronalen Netzes, können Semi-Gradient Verfahren unstabil sein, was zu Oszillationen und/oder zur Divergenz führen kann (vgl. [Sutton & Barto, 2018], S. 200-204). Im Verlauf dieses Kapitels werden einige Ansätze zur Stabilisierung des Lernprozesses in Semi-Gradient-Verfahren vorgestellt.

3.2.5 Gradientenverfahren in policybasierten Algorithmen

In policybasierten Algorithmen werden die Parameter θ einer stochastischen $\pi_\theta(a|s)$ oder einer deterministischen $\mu_\theta(s)$ Policy auf der Basis des geschätzten Gradienten der Policy-Performanz optimiert. Für die Vorstellung der Kernidee wird angenommen, dass ein episodisches RL- Problem vorliegt und der Diskontierungsfaktor $\gamma = 1$ ist.
Die *Policy-Performanz* $J(\theta)$ kann als Erwartungswert des Returns $\mathbb{E}_{\pi_\theta}[G_0 = \sum_{t=1}^{T} r_t]$ ab dem Startzustand s_0 definiert werden. $J(\theta)$ lässt sich bzgl. der Zustandswertefunktion $v^{\pi_\theta}(s)$ wie folgt formulieren (vgl. Abschnitt 3.2.1.3):

$$J(\theta) = \mathbb{E}_{s_0 \sim \rho^\pi(s_0)}[v^{\pi_\theta}(s_0)] \tag{3.35}$$

Gesucht ist der Parametervektor θ^*, der die Policy-Performanz $J(\theta)$ maximiert. Im Laufe des Lernens werden die Parameter θ in Richtung des positiven Gradienten aktualisiert (vgl. [Sutton & Barto, 2018], S. 321):

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_{\theta_t} J(\pi_{\theta_t}) \tag{3.36}$$

Je nachdem, ob die Parameter einer stochastischen oder einer deterministischen Policy optimiert werden, wird zwischen einem stochsatischen Policy-Gradient (Abk.: SPG) und einem deterministischen Policy-Gradient (Abk.: DPG) unterschieden. Des Weiteren kann sowohl SPG als auch DPG on-Policy oder off-Policy geschätzt werden. Die folgende Tabelle fasst die Formeln zur Bestimmung des Policy-Gradienten mit entsprechenden Publikationen zusammen. Die Tabelle wird im Anhang A.2 näher erläutert.

Policy	**Lernen**	
	On-Policy	Off-Policy
Stochastisch	$\mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta log \pi_\theta(a\|s) q^\pi(s,a)]$, [Sutton et al., 1999]	$\mathbb{E}_{s\sim\rho^\beta, a\sim\beta}[\frac{\pi(a\|s)}{\beta(a\|s)} \nabla_\theta log \pi_\theta(a\|s) q^\pi(s,a)]$, [Degris et al., 2012]
Deterministisch	$\mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s))]$, [Silver et al., 2014]	$\mathbb{E}_{s\sim\rho^\beta}[\nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s)]$, [Silver et al., 2014]

Tabelle 3.2: Zusammenfassung der Formeln zur Bestimmung des Policy-Gradienten $\nabla_\theta J$
Quelle: Eigene Darstellung

3.2.6 Deep Reinforcement Learning

Deep Reinforcement Learning (Abk.: Deep RL) kombiniert Deep Learning mit RL-Algorithmen. Deep Learning wurde seit Popularisierung des Backpropagation-Algorithmus (1986) als Approximationsmethode im Rahmen von RL eingesetzt. Ein bekanntes Beispiel dafür ist das TD-Gammon Programm von G. Tesauro (1992, 1994, 1995, 2002). Die Version 1.0 des TD-Programms, die in der Lage war, mit den menschlichen Experten zu konkurrieren, basierte auf manuell konstruierten Merkmalen der Umgebung.
Der Durchbruch von Deep Learning in den letzten Jahren führte dazu, dass es den Forschern von Google DeepMind [Mnih et al., 2013] gelang, im Jahr 2013 einen RL-Agenten ohne eine manuelle Konstruktion der Merkmale zu trainieren. Der Agent war in der Lage, aus hochdimensionalen unstrukturierten Daten (Bildern) die Strategie für 49 Atari-Videospiele zu erlernen und auf menschlichem Niveau bzw. bei einem großen Teil der Spiele über dem menschlichen Niveau zu spielen. Zu den weiteren Deep RL-Erfolgen zählen das Übertreffen der menschlichen Gegner in komplexen Spielen wie Go [Silver et al., 2017] und Dota2 [OpenAi, 2019] sowie die Anwendung der Algorithmen zur Bewegungssteuerung von realen Robotern [Haarnoja et al., 2018], [Zeng et al., 2019], [Andrychowicz et al., 2017].

3.2.6.1 Deep Q-Learning (DQN)

Deep Q-Learning Algorithmus (Abk. DQN) [Mnih et al., 2013] nutzt ein neuronales Netz $Q(s, a; W)$ mit dem Wichtungsvektor W, um die optimale Q-Funktion $q^*(s, a)$ zu approximieren. Zum Zeitpunkt t bekommt das Q-Netz den Zustand s_t als Eingabe und generiert Q-Werte $Q(s_t, a_1), Q(s_t, a_2), \ldots, Q(s_t, a_n)$ als Ausgabe, wobei $a_1, a_2, ..a_n$ die in s_t möglichen Aktionen sind. Der Agent folgt bei der Interaktion mit der Umgebung eine ε-greedy Policy. Abbildung 3.12 stellt schematisch die Eingabe und Ausgabe des Q-Netzes dar.

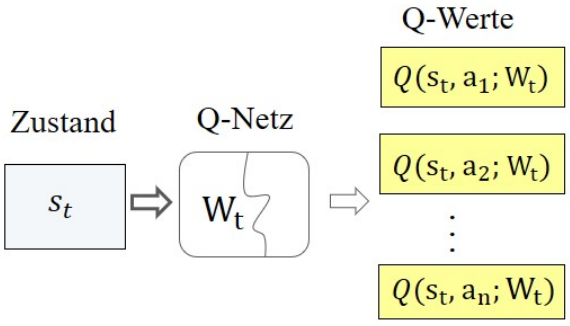


Abbildung 3.12: Eingabe und Ausgabe des Q-Netzes im DQN-Algorithmus
Quelle: Eigene Darstellung

Im klassischen Q-Learning Algorithmus nutzt der Agent für die Aktualisierung der Q-Funktion zum Zeitpunkt t die Transition $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$, die unmittelbar vor

dem Update gesammelt wurde. Dies kann aus den folgenden Gründen den Lernprozess beeinträchtigen (vgl. [Schaul et al., 2015], S. 1):

- der Agent vergisst schnell die möglicherweise nützliche, aber selten vorkommende Erfahrung
- Transitionen, die für das Update der Wertefunktion verwendet werden, sind stark zeitlich korreliert.

Diese beiden Probleme können mithilfe der *Experience Replay* Technik [Lin, 1992] behoben werden: Die Erfahrung des Agenten $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ zum Zeitpunkt t wird in einem Replay-Puffer $\mathcal{D} = e_1, ..., e_N$ gespeichert, N ist die maximale Größe des Replay-Puffers. In jedem Lernschritt wird aus dem Replay-Puffer zufällig und gleichverteilt ein Mini-Batch der Transitionen entnommen und für das Update der Wichtungen des Q-Netzes verwendet. Die entnommenen Transitionen stammen aus der Interaktion mit der Umgebung E nach einer Behavior-Policy β.
Das Q-Netz wird trainiert, indem die folgende Kostenfunktion minimiert wird (vgl. [Mnih et al., 2013], S. 3):

$$L(W_t) = \mathbb{E}_{s\sim\rho^\beta}[(y_t - Q(s,a;W))^2]$$

mit

- $y_t = \mathbb{E}_{r,s'\sim E}[r_{t+1} + \gamma \cdot max_{a'}\bar{Q}(s_{t+1}, a'; \bar{W}_t)]$ für 1-step Update
- $y_t = \mathbb{E}_{r,s'\sim E}[\sum_{i=0}^{n-1} \gamma^i r_{t+i+1} + \gamma^n max_{a'}\bar{Q}(s_{t+n}, a'; \bar{W}_t)]$ für n-step Update, falls $t+n < T$ und
  $y_t = \mathbb{E}_{r,s'\sim E}[\sum_{i=0}^{T-t-1} \gamma^i r_{t+i+1}]$ falls $t+n > T$

Zur Stabilisierung des Lernprozesses wird in DQN ein zweites neuronales Netz $\bar{Q}(s,a;\bar{W})$ mit den Wichtungen $\bar{W}$ verwendet, das im Weiteren als *Q-Zielnetz* referenziert wird. Das Q-Zielnetz stellt eine Kopie des Q-Netzes dar und wird für die Bestimmung der Update-Zielwerte verwendet.
Es wird zwischen zwei Strategien zur Aktualisierung der Wichtungen des Q-Zielnetzes unterschieden:

- *Hard-Update*: Periodisch nach k Lernschritte werden die Wichtungen des Q-Zielnetzes gleich den Wichtungen des Q-Netzes gesetzt: $\bar{W} \leftarrow W$ (vgl. [Mnih et al., 2015], S. 529). In diesem Fall soll k ausreichend groß sein, sodass die Tatsache, dass die Zielwerte nicht konstant bleiben, vernachlässigt werden kann, z. B. $k = 10^4$
- *Soft-Update*: Nach jedem Lernschritt werden die Wichtungen des Q-Zielnetzes nach folgender Regel aktualisiert (vgl. [Lillicrap et al., 2016], S. 4):

  $$\bar{W} \leftarrow \tau W - (1-\tau)\bar{W}, \tag{3.37}$$

  τ ist eine kleine Konstante, z. B. τ=0,999.

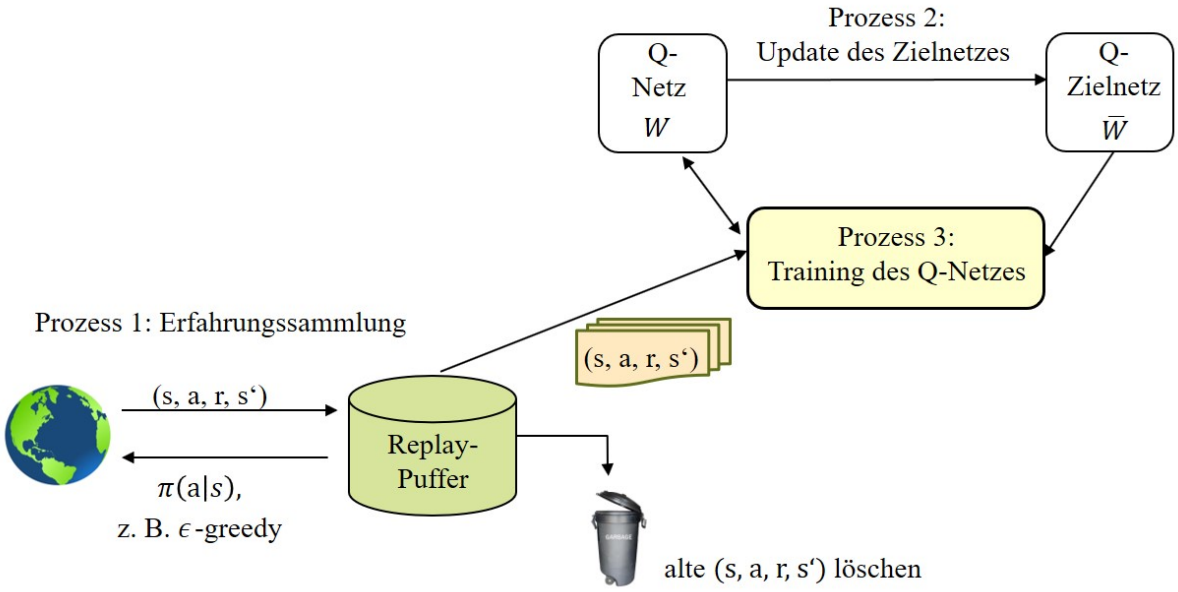


Abbildung 3.13: Visualisierung des DQN-Algorithmus
Quelle: In Anlehnung an [Levine, 2019a], S. 17

Das Zielnetz soll das bereits im Abschnitt 3.2.4 aufgeführtes Problem lösen: die Zielwerte für das Update der Q-Funktion ändern sich im Laufe des Trainings, was einen Unterschied zum üblichen SL-Training darstellt, bei dem die Zielwerte vom Beginn der Lernphase festgelegt werden und über die Zeit gleich bleiben. Da sich die Wichtungen W nach jedem Lernschritt ändern, würde die direkte Verwendung des Q-Netzes bei der Bestimmung der Update-Zielwerte die Konvergenz zur echten Q-Funktion stark erschweren. In der Praxis garantiert die Benutzung des Zielnetzes nicht die Konvergenz des Algorithmus, stabilisiert aber den Lernprozess erheblich (vgl. [Mnih et al., 2015], S. 2 des Anhangs).
Die Abbildung 3.13 fasst den DQN-Algorithmus zusammen. DQN besteht aus drei Prozessen: (1) Sammlung von Erfahrung, (2) Update des Zielnetzes und (3) Verbesserung der Approximation der Q-Funktion. Um das Lernen zu beschleunigen, kann der Prozess (3) in einer Schleife k-mal wiederholt werden, d. h. nach jeder Interaktion finden k Lernschritte statt [Levine, 2019a].

3.2.6.2 Deep Deterministic Policy Gradient (DDPG)

Der DDPG-Algorithmus gehört zur Klasse der off-Policy modellfreien Actor-Critic-Algorithmen und kann in kontinuierlichen Zustands- und Aktionsräumen eingesetzt werden.
Die *Critic-Komponente* bilden das Q-Netz $Q(s, a; W)$ und das Q-Zielnetz $\bar{Q}(s, a; \bar{W})$ mit den Wichtungen W und $\bar{W}$. Die Aufgabe der Critic-Komponente ist das Erlernen der optimalen Q-Funktion $q^*(s, a)$.
Die *Actor-Komponente* bilden das Policy-Netz $\mu(s; \theta)$ und das Policy-Zielnetz $\bar{\mu}(s; \bar{\theta})$ mit den Wichtungen θ und $\bar{\theta}$. Die Actor-Komponente lernt die optimale deterministische Policy $\mu^*(s)$.
Der Agent folgt bei der Interaktion mit der Umgebung einer explorativen Policy, die durch

das Verrauschen von $\mu(s;\theta)$ entsteht:

$$\mu(s_t) = \mu(s_t;\theta_t) + \mathcal{N},$$

$\mathcal{N}$ ist der Rauschprozess. In der Originalpublikation des DDPG-Algorithmus wurde als $\mathcal{N}$ der Ornstein-Uhlenbeck-Prozess [Uhlenbeck & Ornstein, 1930] verwendet. In der Praxis wird oft stattdessen das Gaußsche Rauschen benutzt.
Die Erfahrung $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ wird in einem Replay-Puffer $\mathcal{D}$ gespeichert. In jedem Lernschritt wird aus dem Replay-Puffer zufällig und gleichverteilt ein Mini-Batch der Transitionen entnommen und für die Aktualisierung der Actor- und der Critic-Komponente verwendet.

Training der Actor-Komponente Das Policy-Netz bekommt einen Zustand s als Eingabe und generiert die Aktion a, die den maximalen Q-Wert des Zustand-Aktion-Paares (s, a) ergibt. Der deterministische Policy-Gradient wird wie folgt berechnet (vgl. Tabelle 3.2):

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t\sim\rho^\beta}[\nabla_a Q(s_t, \mu(s_t;\theta); W)\nabla_\theta \mu(s_t;\theta)]$$

Nach jedem Lernschritt werden:

- die Wichtungen des Policy-Netzes θ in Richtung des positiven Gradienten angepasst (s. Gleichung 3.36),
- die Wichtungen des Policy-Zielnetzes $\bar{\theta}$ nach der „Soft-Update“ Strategie aktualisiert (s. Gleichung 3.37)

Training der Critic-Komponente Da die Actor-Komponente darauf zielt, eine Aktion zu generieren, die den größtmöglichen Q-Wert ergibt, wird das Policy-Zielnetz $\bar{\mu}(s;\bar{\theta})$ bei der Bestimmung der Zielwerte für das Update der Wichtungen des Q-Netzes verwendet. Beim Training der Critic-Komponente wird die folgende Kostenfunktion minimiert:

$$L(W_t) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_{t+1}, s_{t+1}\sim E}[(r_{t+1} + \gamma \cdot \bar{Q}(s_{t+1}, \bar{\mu}(s_{t+1};\bar{\theta}); \bar{W}_t) - Q(s_t, a_t; W_t))^2] \quad (3.38)$$

Nach jedem Lernschritt werden:

- die Wichtungen des Q-Netzes W in Richtung des negativen Gradienten der Kostenfunktion angepasst (s. Gleichung 3.33),
- die Wichtungen des Q-Zielnetzes $\bar{W}$ nach der „Soft-Update“ Strategie aktualisiert (s. Gleichung 3.37).

Die Abbildung 3.14 stellt die Kommunikation zwischen den Elementen des DDPG-Algorithmus dar und der Pseudocode 5 im Anhang fasst den DDPG-Algorithmus zusammen.

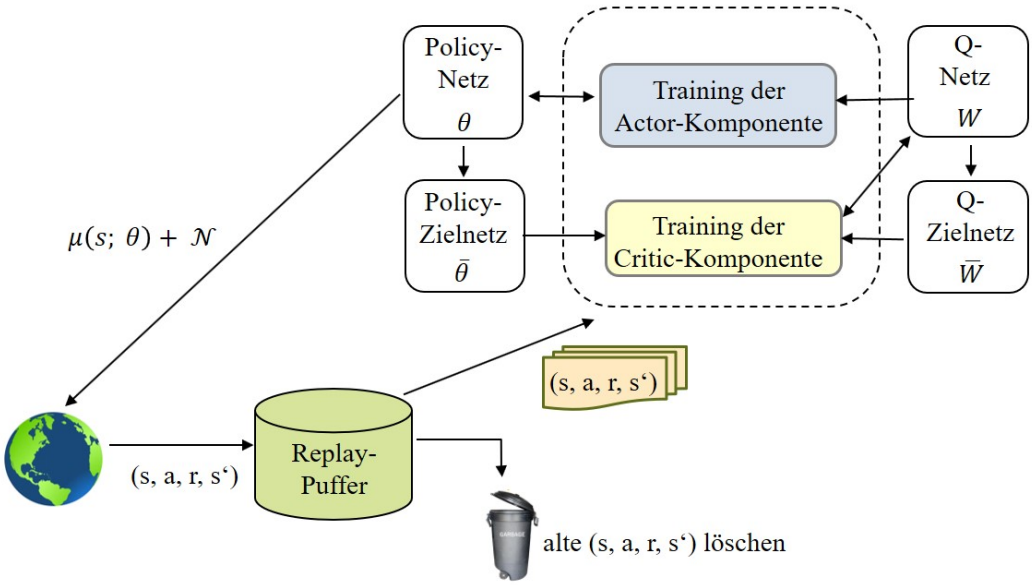


Abbildung 3.14: Visualisierung des DDPG-Algorithmus
Quelle: Eigene Darstellung

DDPG wird in vielen Anwendungen, z. B zur Bewegungssteuerung eines Roboters [Kumar et al., 2018], [Sampedro et al., 2019] oder zum autonomen Fahren [Choi et al., 2019], erfolgreich eingesetzt. Der Lernprozess kann jedoch nach dem Algorithmus instabil verlaufen und hängt stark von der Auswahl der Hyperparameter ab [Achiam, 2019].

3.2.6.3 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Im Jahr 2018 erweiterten [Fujimoto et al., 2018] den DDPG-Algorithmus auf den TD3-Algorithmus. In der Originalpublikation wurden TD3 und DDPG bei dem Erlernen der Bewegungssteuerung eines virtuellen Roboters in der Mujoco Umgebung [Todorov et al., 2012] verglichen. Die Ergebnisse zeigen, dass TD3-Algorithmus in der Lage war, eine signifikant bessere Policy als DDPG zu erzielen.
Die Entwickler des TD3-Algorithmus setzten sich mit den Faktoren auseinander, die den Lernprozess in Actor-Critic-Algorithmen stören können. Auf der Basis der empirischen Tests und theoretischen Überlegungen, kamen Fujimoto et al. zu folgenden Schlüssen: (1) die Critic-Komponente divergiert aufgrund der Überschätzung der Q-Funktion (Eng.: over-estimation bias), wenn die Policy der Actor-Komponente schlecht ist; (2) Die Actor-Komponente generiert eine schlechte Policy, wenn die Schätzung der Q-Funktion inakkurat ist. Um diese zwei Probleme zu umgehen, werden in TD3 folgende Modifikationen des DDPG-Algorithmus vorgenommen:

- Clipped-Double Q-Learning [Hasselt, 2010] als Lösung für den Überschätzungsbias,
- Glättung der Target-Policy und
- Verspätetes Update des Policy-Netzes und der Zielnetze als Maßnahme zur Reduktion der Varianz

Clipped Double Q-Learning Wenn die aktuelle Approximation der Q-Funktion den Fehler ε jeglicher Art enthält, besteht bei der Bestimmung des Update-Zielwertes $y = r + \gamma \cdot max_{a'} Q(s', a')$ das folgende Problem: Das Maximum über die Q-Werte und deren Fehler ergibt einen größeren Wert als das tatsächliche Maximum ist: $\mathbb{E}_\pi[max_{a'} Q(s', a') + \varepsilon] \geq max_{a'} Q(s', a')$. Im DQN-Algorithmus und in Actor-Critic Verfahren, die auf der Funktionsapproximation der Q-Funktion basieren, sind Schätzfehler aufgrund der Ungenauigkeit der Approximation unvermeidlich (vgl. [Fujimoto et al., 2018], S. 3). Da der Q-Learning Algorithmus die Rekursion beinhaltet, wird die Überschätzung weiter propagiert. In [Thrun & Schwartz, 1993], [Hasselt, 2010] und [Fujimoto et al., 2018] wird darauf hingewiesen, dass die Überschätzung der Q-Funktion zu einer suboptimalen Policy und/oder zur Divergenz führen kann.
Der TD3-Algorithmus geht die Problematik des Überschätzungsbias mithilfe der Erweiterung der Critic-Komponente an. In TD3 besteht die Critic-Komponente aus zwei Subkomponenten, die unabhängig voneinander versuchen, die optimale Q-Funktion zu approximieren:

- Critic1: Q-Netz $Q^1(s; W^1)$ sowie das Q-Zielnetz $\bar{Q}^1(s; \bar{W}^1)$ und
- Critic2: Q-Netz $Q^2(s; W^2)$ sowie das Q-Zielnetz $\bar{Q}^2\left(s; \bar{W}^2\right)$,

Während des Trainings minimieren die beiden Subkomponenten die folgende Kostenfunktion:

$$L(W^k{}_t) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_{t+1}, s_{t+1} \sim E}[(y_t - Q^k(s_t, a_t; W^k{}_t))^2], \quad k = 1, 2, \tag{3.39}$$

wobei bei der Bestimmung des Zielwertes y_t das Minimum der beiden Subkomponenten verwendet wird:

$$y_t = r_{t+1} + \gamma \cdot \min_{k=1,2} \bar{Q}^k(s_{t+1}, \bar{\mu}(s_{t+1}; \bar{\theta}); \bar{W}_t^k) \tag{3.40}$$

Zugegeben fördert ein solches Zielwert die Unterschätzung der Q-Funktion. Das heißt, die Aktionen, die das Policy-Netz vorschlägt, können von der Critic-Komponente schlechter als sie tatsächlich sind, beurteilt werden. Nichtsdestotrotz wird die Unterschätzung während des Lernens nicht weiter propagiert, da die Aktionen mit geringen Q-Werten vom Policy-Netz vermieden werden (vgl. [Fujimoto et al., 2018], S. 4).

Verspätetes Update Die Stabilität des Policy-Netzes hängt von der Schätzung der Q-Funktion durch das Q-Netz ab. In TD3 wird das Policy-Netz sowie die Zielnetze (die Q-Zielnetze und das Policy-Zielnetz) seltener als das Q-Netz aktualisiert. Die Idee ist, die ausreichende Verbesserung des Q-Netzes zu erzielen, bevor es für das Update der anderen Netze eingesetzt wird.

Glättung der Target-Policy Bei der Bestimmung des Zielwertes y_t verwendet der TD3-Algorithmus eine verrauschte Ausgabe des Policy-Zielnetzes. Diese Maßnahme fordert,

dass die ähnlichen Zustand-Aktion-Paare schneller ähnliche Q-Werte erhalten. Dies glättet die Policy und hilft dabei, das Overfitting des Policy-Netzes zu vermeiden. Mit dieser Modifikation werden die Zielwerte für das Update des Q-Netzes wie folgt berechnet:

$$y_t = r_{t+1} + \gamma \cdot \min_{k=1,2} \bar{Q}^k(s_{t+1}, \bar{\mu}(s_{t+1}; \bar{\theta}) + \varphi); \bar{W^k}_t), \varphi \sim clip((\mathcal{N}, 0, \sigma), -c, c)$$

Als φ wird das Gaußsche Rauschen notiert, das die Grenzen des Intervalls $[-c; c]$ nicht überschreiten darf, um die verrauschten Aktionen den ursprünglichen Aktionen ähnlich zu halten.
Der Algorithmus 6 im Anhang fasst den TD3-Algorithmus zusammen.

3.2.6.4 Prioritized Experience Replay(PER)

In den oben vorgestellten RL-Algorithmen besteht der Replay-Puffer $\mathcal{D}$ aus Transitionen, die die gleiche Wahrscheinlichkeit haben, in jedem Lernschritt in dem Mini-Batch entnommen zu werden. Die Idee von *Prioritized Experience Replay* Technik (dt: priorisierte Erfahrungswiederholung, Abk.: PER) [Schaul et al., 2015] basiert auf der Tatsache, dass der Agent von einigen Transitionen effektiver lernen kann als von anderen. Ein Beispiel dafür ist die Situation, bei der der Agent sehr selten verschiedene Rewards bekommt, z. B. der Reward „+1“ für das Gewinnen eines Spiels und „0“ in allen anderen Fällen. Der Agent erlebt zufällig einen Erfolg durch die Exploration. Diese Transition ist viel wichtiger für den Agenten als die anderen Transitionen im Puffer, weil nur sie das gewünschte Verhalten des Agenten beschreibt. Angenommen, nach einer Weile lernt der Agent, in einem bestimmten Zustand immer die Aktion auszuführen, die zum Erfolg führte. Ab diesem Zeitpunkt wird die Transition weniger wichtig für den Lernprozess, da sie keine neuen Informationen mehr enthält. Allgemein haben also die Transitionen, die neue und überraschende Informationen für den Agenten enthalten, einen größeren Wert für den Lernprozess. Dieser Wert kann durch den TD-Fehler gemessen werden (vgl. [Schaul et al., 2015], S. 2-3).
In PER werden die Transitionen in einem Replay-Puffer $\mathcal{D}$ der Größe N gespeichert. Jeder neuen Transition wird die maximale Priorität zugewiesen (die bis dahin höchste Priorität), die mit in den Replay-Puffer gespeichert und im Laufe des Lernprozesses aktualisiert wird. Je nach der Berechnung der Priorität einer Transition, unterscheiden die Autoren von PER zwischen einer rankbasierten und proportionalen Priorisierung. Bei der *proportionalen Priorisierung* wird die Priorität $p(i)$ einer Transition e_i proportional zu ihrem absoluten TD-Fehler δ_i gesetzt, der misst, wie überraschend die Transition für den Agenten ist:

$$p(i) = |\delta_i| + \epsilon, \tag{3.41}$$

ϵ ist eine kleine positive Konstante, die sicherstellt, dass alle Transitionen mit einer gewissen Wahrscheinlichkeit aus $\mathcal{D}$ entnommen werden können. Aus der Definition der Priorität (3.41) folgt, dass $p(i)$ sich nach jeder Aktualisierung der Wichtungen des Q-Netzes ändern soll. Da es rechenintensiv ist, nach jedem Lernschritt den TD-Fehler für alle im

Replay-Puffer gespeicherten Transitionen neu zu berechnen, müssen in der Praxis einige Vereinfachungen vorgenommen werden. Eine mögliche Option ist, die Priorität nur für die Transitionen aus dem aktuellen Mini-Batch zu aktualisieren.
Die Wahrscheinlichkeit $P(i)$ der Entnahme einer Transition e_i aus $\mathcal{D}$ ist proportional zu deren Priorität $p(i)$:

$$P(i) = \frac{{p_k}^\alpha}{\sum_k {p_k}^\alpha} \tag{3.42}$$

Der Exponent α bestimmt, wie stark die Priorisierung ist. Der Fall $\alpha = 0$ entspricht einer gleichmäßigen Entnahme der Transitionen. Bei $\alpha = 1$ wird eine gierige Priorisierung durchgeführt, bei der nur die Transitionen mit einem hohen Fehler ausgewählt werden. Die gierige Priorisierung ist jedoch nicht immer erwünscht, da die Transitionen, die beim ersten Mal einen kleinen TD-Fehler hatten, im Weiteren fast ignoriert werden. Das führt dazu, dass für das Lernen nur einen Teil der Erfahrung verwendet wird.
Das priorisierte Sampling von Transitionen ändert die Verteilung und führt zu einem Bias bei der Schätzung des Gradienten. Um eine Schätzung ohne Bias zu erhalten, muss der TD-Fehler bei dem Update der Netzparameter mit der Importance-Sampling-Wichtung w_i (Abk.: IS-Wichtung) korrigiert werden. Für die Stabilisierung des Lernprozesses werden die Wichtungen mit $\frac{1}{\max_i w_i}$ normalisiert:

$$w_i = \frac{N \cdot P(i)^{-\beta}}{\max_i w_i} \tag{3.43}$$

Das Bias wird komplett entfernt bei $\beta = 1$.
Die Autoren von PER [Schaul et al., 2015] schlagen eine effiziente Umsetzung eines priorisierten Replay-Puffers mithilfe der Datenstruktur Summenbaum (Eng.: sum tree) vor. Der Summenbaum ist ein spezifischer Fall eines binären Baums, bei dem die Elternknoten der Summe ihrer Kinderknoten entsprechen.
Bei PER speichern die Blätter des Summenbaums die Prioritäten der Transitionen, die internen Knoten speichern deren Zwischensummen und die Wurzel entspricht der Summe über alle Prioritäten $p_{total} = \sum_k {p_k}^\alpha$. Die Blätter sind von links nach rechts von 0 bis $(N-1)$ nummeriert. Die Indizes der Blätter entsprechen den Indizes der Transitionen im Replay-Puffer. Ist der Eintrag mit dem Index i im Replay-Puffer leer, hat das entsprechende Blatt den Wert 0 (d. h. die Priorität der Transition ist 0). Wird eine neue Transition in den Replay-Puffer hinzugefügt, bekommt das entsprechende Blatt die maximale Priorität, die in den Blättern des Baums vorkommt.
Um einen Mini-Batch der Größe m aus dem Replay-Puffer zu entnehmen, wird das Intervall $[0; p_{total}]$ in m äquivalenten Segmenten aufgeteilt. Aus jedem Segment wird gleichmäßig ein Wert x_i ausgewählt. Für jeden Wert x_i wird der Baum abgefragt, um den höchsten Index i zu finden, sodass die Summe der vorherigen Blätter kleiner oder gleich x_i ist:

$$sum(arr[0] + arr[1] + ... + arr[i - i]) <= x_i$$

Abschließend werden aus dem Replay-Puffer die Transitionen extrahiert, die die gleichen Indizes aufweisen wie die ausgewählten Blätter.

3.3 Imitation Learning in der Robotik

Das Ziel des *Imitation Learning* (Abk.: IL) ist es, einem System zu ermöglichen, ein gewünschtes Verhalten aus der Beobachtung eines Expertenverhaltens zu erlernen [Osa et al., 2018]. Für diese Arbeit ist der Anwendungsfall relevant, in dem ein Mensch oder ein anderer KI-Agent als Experte auftritt und ein realer oder virtueller Roboter als Lerner.
Die Forschungsrichtung Imitation Learning wurde dadurch motiviert, dass für die Programmierung der autonomen Roboter eine Reihe von fachlichen Kenntnissen benötigt wird, die der Endnutzer in der Regel nicht besitzt. Eine natürliche und intuitive Möglichkeit, das Wissen zu vermitteln, ist die Demonstration (vgl. [Hussein et al., 2017], S. 2). In IL demonstriert der Endnutzer dem Roboter, wie er selbst die Aufgabe erledigt. Der Roboter lernt eine Strategie zur Ausführung der gegebenen Aufgabe, indem er die gezeigte Strategie imitiert (vgl. [Billard & Grollman, 2013]). Außer der einfachen Wissensvermittlung bietet die Verwendung des IL einige weitere Vorteile. Erstens erlaubt IL den Suchraum für die Lernalgorithmen zu reduzieren. Zweitens kann IL die Akzeptanz der KI-Anwendung beim Endnutzer erhöhen (vgl. [Billard et al., 2008], S. 59). Viele Entwicklungen in ML beruhen sich auf der Fähigkeit eines KI-Agenten, sich wie ein Mensch zu verhalten. Beispiele dafür sind selbstfahrende Fahrzeuge, Assistenzroboter und Chatbots. In diesen Bereichen ist es notwendig, dass die Strategie des Agenten für Menschen glaubwürdig und natürlich wirkt (vgl. [Hussein et al., 2017], S. 3).

3.3.1 Arten von Demonstrationen

Demonstrationsmethoden, die in IL verwendet werden, können in folgende Klassen aufgeteilt werden:

- *Kinästhetische Demonstrationen* Während der Demonstrationen bewegt der Experte Gelenke des Roboters. Die Gelenkmotoren werden dabei ausgeschaltet, um eine einfache Steuerung zu ermöglichen. Die Aktionen des Experten werden durch die Bewegungssensoren des Roboters erfasst. Ein Vorteil von kinästhetischen Demonstrationen besteht darin, dass die Aufgabe direkt mit dem Roboter ausgeführt wird. Dadurch entfällt die Notwendigkeit, die Unterschiede in der Kinematik des Experten und des Roboters zu berücksichtigen.
  Die kinästhetischen Demonstrationen können jedoch die Kräfte und Drehmomente beeinflussen, sodass aufgenommene Daten inakkurat werden (vgl. [Zhu & Hu, 2018], S. 6). Des Weiteren können kinästhetische Demonstrationen zu unsynchronisierten Bewegungen führen, da der Lehrer die Motoren nacheinander bewegt, anstatt natürliche koordinierte Bewegungen zu zeigen (vgl. [Ott et al., 2013], S. 1).

- *Bewegungssensorbasierte Demonstrationen* Die Bewegung des Experten kann mithilfe eines Tracking-Systems, die auf einem Exoskelett oder einem anderen tragbaren Bewegungssensor basiert, direkt erfasst werden. Der Vorteil von Motor-Sensor Demonstrationen besteht darin, dass der Experte sich frei bewegen kann. Jedoch muss die vorgezeigte Bewegung in die Gelenkkonfiguration des Roboters transformiert und an die Limitationen des Körperbaus des Roboters angepasst werden (vgl. [Zhu & Hu, 2018], S. 7).
- *Ferngesteuerte Demonstrationen* Während der ferngesteuerten Demonstrationen steuert der Experte den Roboter mithilfe eines Kontrollers und der Roboter erfasst die Aktionen durch seine eigenen Sensoren. Die Bewegungen des Experten sind vom Körperbau des Roboters beschränkt und benötigen keine zusätzlichen Transformationen. Des Weiteren werden bei ferngesteuerten Demonstrationen nur die kinematischen Informationen (Position, Geschwindigkeit) an den Roboter geschickt, wobei die Steuerung der Kräfte unbeeinflusst bleibt.
  Der Hauptnachteil der ferngesteuerten Demonstrationen besteht darin, dass der Experte oft eine Schulung benötigt, um die Bedienung des Kontrollers zu erlernen. Außerdem kann eine Fernsteuerung eines Roboters, der eine hohe Anzahl der Freiheitsgrade besitzt, sehr komplex sein (vgl. [Billard & Grollman, 2013]).

3.3.2 Imitation Learning Methoden

In diesem Abschnitt werden zwei weitverbreitete Klassen von Imitation Learning Algorithmen vorgestellt.

3.3.2.1 Behavioral-Cloning

In Behavioral-Cloning werden die Demonstrationen als Zustand-Aktion-Paare (s_t, a_t) in einer geeigneten Datenstruktur $\mathcal{D}$ (Datenbank, Puffer, etc.) gespeichert.
Gegeben ist ein Trainingsbeispiel $\mathcal{D}^i = (s_0^i, a_0^i, \ldots, s_T^i, a_T^i)$ aus $\mathcal{D}$, wobei T der finale Zeitschritt ist. Der Agent kann mit einem SL-Algorithmus trainiert werden. Dabei wird die folgende Kostenfunktion minimiert:

$$\mathcal{L}_{BC}(\theta) = \frac{1}{2}(\pi(s_t^i|\theta) - a_t^i)^2 \tag{3.44}$$

$\mathcal{L}_{BC}$ wird als *Behavioral-Cloning-Kostenfunktion* bezeichnet und beschreibt quadrierte Abweichung zwischen der Aktion, die der Agent nach seiner Policy $\pi(s_t|\theta)$ in s_t auswählt, und der Aktion a_t, die der Experte im Zustand s_t ausgeführt hat. Der Agent sucht nach der optimalen Parameter $\theta^* = argmin_\theta\ \mathcal{L}_{BC}(\theta)$ für alle mögliche θ. Die durch Minimierung von $\mathcal{L}_{BC}$ gelernte Policy erlaubt dem Agenten vorherzusagen, welche Aktion der Experte in dem Zustand s_t auswählen würde (vgl. [Goecks et al., 2019], S. 3).
Zu den erfolgreichen Beispielen der Anwendung von Behavior Cloning Methode zählen das

autonomen Fahren [Bojarski et al., 2016] und die Steuerung einer Drohne [Giusti et al., 2016].
Für die Anwendung von Behavioral-Cloning sollen die Aktionen des Experten (die Zielwerte für den SL-Algorithmus) bekannt sein (*Learning from Demonstration*, Abk.: LfD). Jedoch kann diese Voraussetzung nicht immer erfüllt werden. In vielen Problemstellungen kann der Agent die Aktionen des Experten zwar beobachten, kennt aber die genauen nummerischen Werte der Aktion nicht(*Learning from Observation*, Abk.: LfO). Ein Bespiel dafür ist der Anwendungsfall, bei dem der Roboter eine Strategie aus einem Video erlernen soll (vgl. [Goo & Niekum, 2018], S. 1). Ein weiterer Nachteil von Behavioral-Cloning besteht darin, dass der Agent nur die Policy des Experten nachmacht, ohne die Schlussfolgerungen über seine Aktionen zu ziehen (vgl. [Levine, 2019c], S. 5). Das heißt, wenn der Agent sich in einem Zustand befindet, der keinem Zustand aus der Trainingsmenge ähnelt, ist der Agent nicht in der Lage sich ähnlich wie der Experte zu verhalten, und braucht weitere Demonstrationen.

3.3.2.2 Inverse Reinforcement Learning

Der Lernprozess im Inverse Reinforcement Learning wird in zwei folgenden Phasen aufgeteilt:

1. Das Lernproblem wird als ein unvollständiger MDP formalisiert, in dem die Rewardfunktion $\mathcal{R}$ unbekannt ist. Die Policy des Experten gilt als optimale Policy π^* für den MDP. Dem Agenten stehen Demonstrationen zur Verfügung, also die Agent-Umgebung Interaktion nach π^*. Das Ziel in diesem Schritt besteht darin, die Rewardfunktion $\mathcal{R}$ zu erlernen, die der Experte versucht hat zu maximieren.
   Dieser Schritt hat folgende technische Herausforderungen (vgl. [Levine, 2019c], S. 6):
   - die gelernte Rewardfunktion ist schwierig zu evaluieren,
   - die Demonstrationen können suboptimal sein,
   - die Spezifizierung einer Kostenfunktion stellt eine nicht triviale Aufgabe dar, denn zu einer Policy können mehrere passende Rewardfunktionen existieren.
2. Wenn die Rewardfunktion erlernt ist, kann das Lernproblem als ein vollständiger MDP formalisiert und mit einem RL-Algorithmus gelöst werden.

In den letzten 20 Jahren wurde eine Reihe von IRL-Algorithmen entwickelt, u.a.

- Feature matching IRL [Abbeel & Ng, 2004], 2004
- Bayesian IRL [Ramachandran & Amir, 2007], 2007
- Maximum Entropy IR [Ziebart et al., 2008], 2008
- Maximum Causal Entropy IR [Ziebart et al., 2010], 2010
- Maximum Entropy Deep IRL [Wulfmeier et al., 2015], 2016

3.4 Zusammenfassung

Dieses Kapitel besteht aus drei Themenblöcken:

- **Künstliche neuronale Netze (mit Fokus auf Deep-Feedforward-Netze)**
  Künstliche neuronale Netze stellen ein wichtiges Werkzeug zur Funktionsapproximation dar. Im Lernprozess werden die Wichtungen des Netzes iterativ angepasst, und zwar basierend darauf, wie falsch die Ausgabe des Netzes für eine bestimmte Eingabe ist, verglichen mit dem bekannten wahren Beispiel.
  In diesem Abschnitt wurde das mathematische Modell eines künstlichen Neurons und der Lernalgorithmus Backpropagation vorgestellt. Außerdem wurde ein Überblick über mögliche Probleme beim Training eines neuronalen Netzes gegeben und einige Maßnahmen zur Verbesserung des Lernprozesses betrachtet (Anpassung der Lernrate und der Tiefe des Netzes sowie Regularisierungsmaßnahmen Early Stopping und L^1 und L^2 Strafterme). Der Abschnitt endete mit der Erläuterung von folgenden Optimierungsalgorithmen: Batch, Mini-Batch und stochastischer Gradientenabstiegsverfahren, Gradientenabstieg mit Momentum sowie der Adam-Algorithmus.
- **Reinforcement Learning (mit Fokus auf Deep Reiforcement Learning)**
  Reinforcement Learning ermöglicht dem Agenten aus eigener Erfahrung zu lernen. In diesem Abschnitt wurde ein RL-Lernproblem als ein MDP formalisiert und Konzepte, die benötigt werden, um ein MDP zu lösen, vorgestellt (Wertefunktionen, Return, Policy). Dann wurde eine Taxonomie von RL-Algorithmen präsentiert, die diese Algorithmen in folgende Klassen aufteilt:
  - on-Policy und off-Policy,
  - modellbasiert und modellfrei. Die modellfreien Algorithmen wurden zusätzlich in wertebasierte, policybasierte und Actor-Critic-Verfahren aufgeteilt.

  Als Grundlage zum Deep RL wurden Gradientenverfahren in wertebasierten und policybasierten Algorithmen vorgestellt.
  Folgende RL-Algorithmen wurden im Rahmen dieses Abschnittes näher betrachtet:
  - Q-Learning: 1-step Q-Learning, n-step Q-Learning,
  - Deep Q-Learning (DQN): 1-step DQN, n-step DQN Learning,
  - Deep Deterministic Policy Gradient (DDPG),
  - Twin Delayed Deep Deterministic Policy Gradient (TD3).

  Abschließend wurde die Prioritized Replay Experience (PER), eine Technik zum priorisierten Sampling von Transitionen, aus denen der Agent lernt, erläutert.
- **Imitation Learning**
  Imitation Learning erlaubt dem Agenten ein gewünschtes Verhalten aus der Beobachtung einer Expertenstrategie zu erlernen. In diesem Abschnitt wurden verschiedene Demonstrationsarten (kinästhetische, bewegungssensorbasierte und ferngesteuerte

Demonstrationen) mit ihren Nachteilen und Vorteilen vorgestellt. Außerdem wurde ein Überblick über zwei Klassen von IL-Algorithmen gegeben:

- Behavioral Cloning und
- Inverse Imitation Learning.

4 Methodik

4.1 Anforderungen an den Lernalgorithmus

Das Lernverfahren soll folgende Anforderungen erfüllen:

1. Der Lernenalgorithmus soll das Lernen anhand einer begrenzten Anzahl der Interaktionen mit der Umgebung unterstützen (Sample-Effizienz).
   Das Erlernen von bestimmten Fähigkeiten in der realen Welt beansprucht viel Zeit. Die Formalisierung einer neuen Aufgabe erfordert mehrere Versuche, dazu kommt die Zeit, die der Agent benötigt, um die neue Fähigkeit zu beherrschen [Haarnoja et al., 2018]. In der Robotik wird des Weiteren die Anzahl der Interaktionen zwischen dem Roboter und der Umgebung durch die Limitationen des Roboters, wie z. B. die Gelenküberhitzung, eingeschränkt. Aus diesem Grund ist die Sample-Effizienz eine der wichtigsten Anforderungen an den Lernalgorithmus.
2. Das Lernen anhand von Demonstrationen und der alten Erfahrung des Agenten soll möglich sein (off-Policy Learning).
   Die Konfiguration von Hyperparametern für Deep RL-Algorithmen wird vom Entwickler durch den Vergleich der Leistung des Algorithmus unter verschiedenen Einstellungen bestimmt. Außer den Hyperparamatern wird auch die Rewardfunktion mehrmals angepasst. Ein off-Policy-Algorithmus ermöglicht es, die bereits erfasste Erfahrung des Agenten für die neuen Einstellungen zu verwenden sowie anhand von Daten aus Demonstrationen zu lernen.
3. Der Lernenalgorithmus soll in hochdimensionalen kontinuierlichen Zustands- und Aktionsräumen anwendbar sein.
4. Das Lernverfahren soll robust gegen Sensorfehler und Verzögerungen in der Bildverarbeitung sein.

4.2 Ähnliche Arbeiten

Der in dieser Arbeit vorgeschlagene Lösungsansatz kann in die Kategorie *RL mit Expertendemonstrationen* (Abk.: RLED), die eine Kombination von RL und IL (spezifisch LfD) darstellt, eingeordnet werden. Das Ziel von RLED besteht darin, die optimale Policy in einem MDP anhand der eigenen Erfahrung des Agenten sowie der Erfahrung des Experten

zu bestimmen (vgl. [Piot et al., 2014], S. 1).
Die erste Inspiration für den Lösungsansatz stellte die Arbeit [Hester et al., 2017], in der der Algorithmus **Deep Q Learning from Demonstrations** (Abk.: DQfD) vorgestellt wurde. In DQfD besteht der Replay-Puffer aus Demonstrationen und eigener Erfahrung des Agenten. Beim Erreichen der maximalen Größe des Replay-Puffers wird nur die eigene Erfahrung mit neuen Transitionen überschrieben. Bevor der Agent die Interaktion mit der Umgebung anfängt, wird er mit den Demonstrationen vortrainiert. Die zu minimierende Kostenfunktion besteht aus vier Summanden: 1-step Double-Q-Learning-Fehler, n-step Double-Q-Learning-Fehler, Large-Margin-Fehler (aus SL, erhöht die Q-Werte der Aktionen des Experten) und L^2-Regularisierung (vermeidet das Overfitting auf Demonstrationen). Die Auswahl der Transitionen für die Aktualisierung der Netzwichtungen wird mithilfe der PER-Technik durchgeführt. Die Transitionen des Experten bekommen dabei einen Bonus zu ihrer Priorität in Höhe von einer Konstante ϵ_d. DQfD wurde in der Arcade Umgebung [Bellemare et al., 2013] an 6 Atari-Videospielen evaluiert. Der Lernprozess konnte durch Demonstrationen beschleunigt werden und erlaubte dem Agenten eine hohe Leistung in schwierigen Problemstellungen zu erzielen.
In [Pohlen et al., 2018] wird der DQfD Algorithmus um die Ape-X-Architektur [Horgan et al., 2018], auch als verteilte PER-Technik bekannt, erweitert. Die Interaktion mit der Umgebung, die Aktualisierung der neuronalen Netze und die Speicherung der Transitionen in dem Replay-Puffer werden in der Ape-X-Architektur auf verschiedene Prozesse verteilt. Der resultierende Algorithmus wird **Ape-X DQfD** genannt. Die Abbildung 4.1 visualisiert die Idee des Algorithmus.

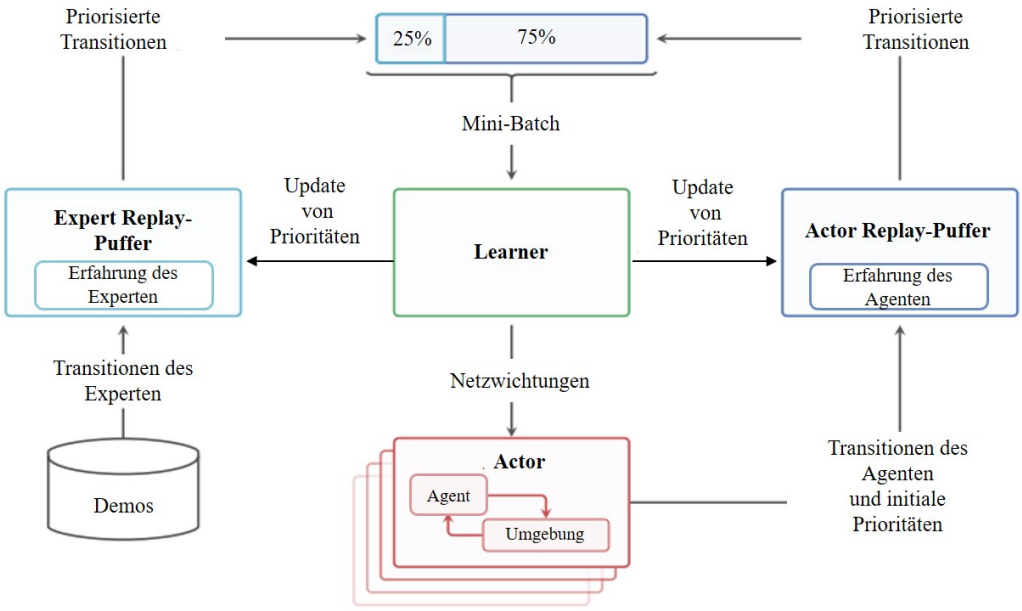


Abbildung 4.1: Ape-X DQfD
Quelle: In Anlehnung an [Pohlen et al., 2018]

Außer der verteilten Architektur unterscheidet sich Ape-X DQfD von DQfD-Algorithmus

in drei Aspekten. Erstens wird der Agent vor der Interaktion mit der Umgebung nicht mit den Demonstrationen vortrainiert, sondern lernt direkt anhand einer Mischung von eigener Erfahrung und der Erfahrung des Experten. Zweitens besteht ein Mini-Batch in jedem Lernschritt aus 75% eigener Transitionen des Agenten und 25% Transitionen des Experten. Dieses Verhältnis bleibt im Laufe des Trainings konstant. Drittens wird der Large-Margin-Fehler nur für die besten Episoden des Experten berechnet.
In [Vecerík et al., 2017] wird der Algorithmus **Deep Deterministic Policy Gradient from Demonstrations** (Abk.: DDPGfD) vorgestellt und zum Erlernen einer Strategie zur Bewegungssteuerung eines 7-DOF Roboterarmes angewendet. DDPGfD kombiniert RL und IL, indem er DDPG-Algorithmus auf die Nutzung von Demonstrationen erweitert. Außerdem modifiziert DDPGfD den Lernprozess, sodass ein effizientes Lernen in den Problemstellungen mit einer Sparse-Rewardfunktion möglich ist. In DDPGfD wird der Agent ähnlich wie in DQfD mit Demonstrationen des Experten vortrainiert. Im Gegensatz zur DQfD wird jedoch in DDPGfD der Large-Margin-Fehler beim Training des Agenten nicht berücksichtigt.
Die Autoren evaluierten DDPGfD sowohl in der Simulation als auch mit dem realen Roboter in folgenden Aufgabenstellungen (Abbildung 4.2):

a) Stift-Platzierung (Simulation)

b) Festplatte-Platzierung (Simulation)
   Dieses Problem unterscheidet sich von a) dadurch, dass für die Lösung die Orientierung der Festplatte beachtet werden muss.

c) Platzierung eines Clips (Simulation, realer Roboter)
   Der Clip besteht aus drei separaten Teilen.

d) Platzierung eines Kabels (Simulation)
   Als Kabel wurde eine stark unteraktuierte 20-gliedrige Kette verwendet.

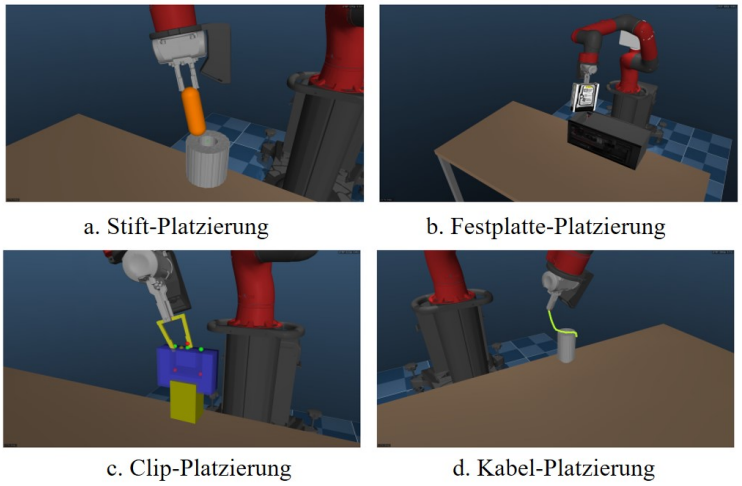


Abbildung 4.2: Aufgabenstellungen für die ursprüngliche Evaluation des DDPGfD-Algorithmus
Quelle: [Vecerík et al., 2017]

Die Policy des Experten wurde durch das kinästhetische Führen vorgezeigt. Um sicherzustellen, dass der Experte die Dynamik der Umgebung nicht beeinflusst, wurde der lernende Roboterarm nicht direkt, sondern über einen anderen Roboterarm gesteuert. Alle Bewegungen wurden während der Demonstration von dem lernenden Roboterarm wiederholt.
Die Forscher haben die experimentellen Versuche sowohl mit einer Shaped-Rewardfunktion als auch mit einer Sparse-Rewardfunktion durchgeführt. Die Ergebnisse zeigen, dass DDPGfD in allen Konfigurationen der Umgebung (Aufgabenstellungen *a*-*d* und unterschiedliche Rewardfunktionen) DDPG[3] und einen puren Behavioral-Cloning-Algorithmus übertrifft. Der Lernprozess verläuft in DDPGfD schneller und zuverlässiger und erlaubt dem Agenten einen höheren Gesamtreward zu erlangen. In den Aufgabenstellungen *a*-*c* war der Agent in der Lage, einen größeren durchschnittlichen Gesamtreward pro Episode als der Demonstrator zu erreichen.
[Nair et al., 2017] setzten sich mit der Aufgabenstellung zur Bewegungssteuerung auseinander, die ein mehrstufiges Vorgehen und die Generalisierung auf unterschiedlichen Zielzustände erfordert. Sie schlagen eine Erweiterung von DDPG vor, bei der die Actor-Komponente neben der Maximierung der Policy-Performanz die Behavioral-Cloning-Kostenfunktion L_{BC} (s. Formel 3.44) minimiert. Um das Problem der suboptimalen Demonstrationen zu behandeln, wird L_{BC} nur bei den Zuständen berücksichtigt, bei denen die Critic-Komponente die Aktion des Experten besser als die Aktion des Agenten findet (Q-Filter). Des Weiteren wird in diesem Ansatz die Technik *Hindsight Experience Replay* (Abk.: HER) [Andrychowicz et al., 2017] benutzt, die dem Agenten erlaubt auch aus fehlgeschlagenen Episoden zu lernen. Bei HER enthalten die Transitionen die Information über das Ziel des Agenten und werden im Replay-Puffer nach jeder Episode zwei Mal gespeichert: einmal mit dem echten Ziel und dem tatsächlich erlebten Reward und einmal mit einem imaginären Ziel (das Ziel, das der Agent tatsächlich erreicht hat) und einem positiven Reward. Beim Training werden die Transitionen gleichmäßig behandelt.
Der Lösungsansatz wurde in der Publikation in einer Aufgabenstellung evaluiert, bei der ein 7-DOF Roboterarm 6 Blöcke nacheinander stapeln musste. Für die Evaluation wurde eine Simulation (die MuJoCo Umgebung) verwendet. Der vorgeschlagene Algorithmus erzielte im Vergleich mit DDPGfD eine bessere Leistung in Hinsicht auf die Geschwindigkeit des Lernens und den Gesamtreward.

4.3 Auswahl des Lernalgorithmus

Das Spiel Ball-in-a-Cup besitzt als Lernszenario folgende Eigenschaften:

- der Zustandsraum und der Aktionsraum sind kontinuierlich,
- ein episodisches Lernproblem,

[3] Der DDPG wurde in den Experimenten auf die PER-Technik, n-step Zielwerte und L^2-Regularisierung erweitert.

- die Umgebung ist stochastisch,
- da die Position und die Orientierung des Bechers sich mit jeder Bewegung des Roboters ändert, ändert sich auch die Zielposition für den Ball,
- die Informationen, die der Roboter über den Zustand der Umwelt bekommt, können aufgrund der Sensorfehler oder Verzögerungen in der Bildverarbeitung unvollständig oder verzerrt sein.

Die Algorithmen DQfD und Ape-X DQfD sind nur in diskreten Zustands- und Aktionsräumen anwendbar, da sie auf dem DQN Algorithmus basieren. Der Lösungsansatz in [Nair et al., 2017] und der DDPGfD-Algorithmus können dagegen in kontinuierlichen Zustands- und Aktionsräumen angewendet werden und wurden in den Originalpublikationen an Lernaufgaben zur Bewegungssteuerung eines Roboters evaluiert. Jedoch verwenden [Nair et al., 2017] die HER-Technik, die sich als eine effektive Lösung für Anwendungsfälle mit einer Sparse-Rewardfunktion, aber mit einem statischen Ziel erwiesen hat. Für das Ball-in-a-Cup-Spiel muss der Ansatz auf HER mit dynamischen Zielen erweitert werden, z. B. mithilfe von Dynamic HER [Fang et al., 2019].
Zur Umsetzung wird der **DDPGfD-Algorithmus** ausgewählt und modifiziert. Die Grundlage für DDPGfD (2017) stellt DDPG dar. **TD-Algorithmus** (2018) schlägt die im Abschnitt 3.2.6.3 beschriebenen Modifikationen des DDPG-Algorithmus vor, die das Potenzial haben, den Lernprozess zu stabilisieren und zu beschleunigen.
In dieser Arbeit wird DDPGfD auf folgenden in TD3 umgesetzten Modifikationen erweitert:

- Clipped-Double Q-Learning,
- Glättung der Target-Policy.

Da in der vorliegenden Arbeit das Interesse darin besteht, eine gute Policy schnell zu erlernen, wird die Modifikation „verspätetes Update“ aus TD3 nicht übernommen.
Die Abbildung 4.3 zeigt die Verfahren, auf denen sich der umgesetzte Lernalgorithmus aufbaut. Die Abbildung enthält die Angaben über die Abschnitte dieser Arbeit, in denen die relevanten Algorithmen erläutert werden und die Referenzen auf die Originalpublikationen. Außerdem sind die Verfahren je nach der Zugehörigkeit zu den folgenden Themenbereichen durch die gestrichelte Linie getrennt:

1. Policybasierte Algorithmen
2. Wertebasierte Algorithmen
3. Actor-Critic Algorithmen
4. Experience Replay
5. Kombination von RL und IL

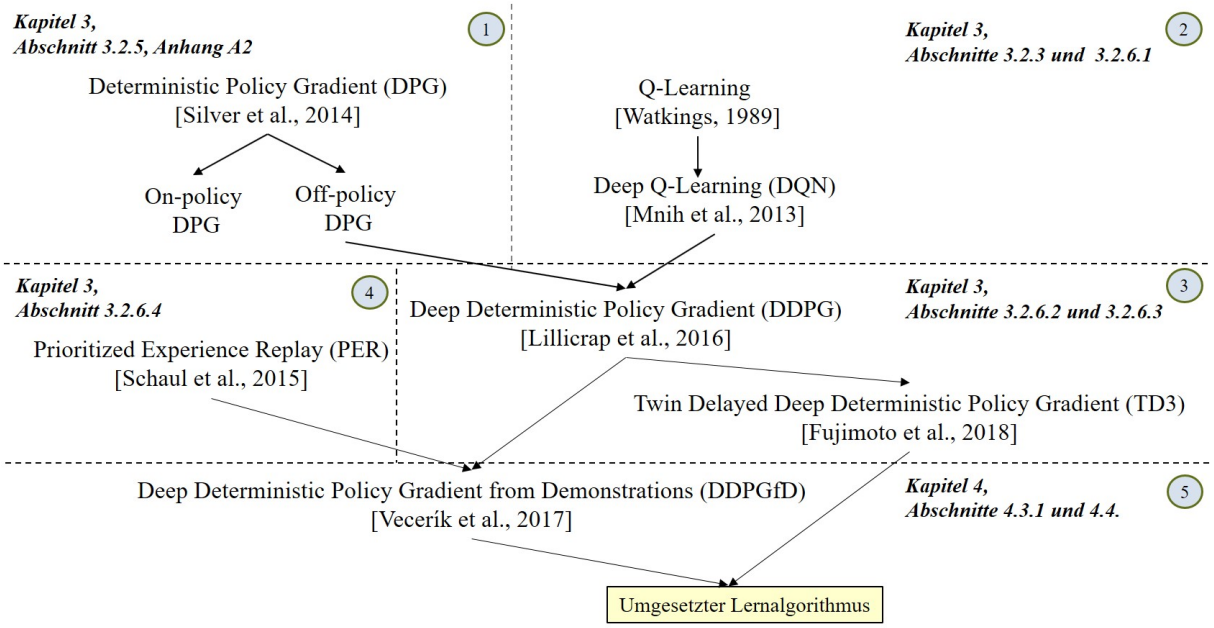


Abbildung 4.3: Einordnung des umgesetzten Algorithmus
Quelle: Eigene Darstellung

4.3.1 Deep Deterministic Policy Gradient from Demonstrations (DDPGfD)

DDPGfD unterscheidet sich von DDPG in folgenden Aspekten:

1. Der Replay-Puffer besteht aus den demonstrierten Transitionen und den Transitionen, die der Agent selbst erlebt hat.
2. Statt einer zufälligen Entnahme der Transitionen aus dem Replay-Puffer, wird in DDPGfD die PER-Technik verwendet. Die Formel (3.41) wird angepasst, sodass die Priorität einer Transition aus vier Summanden besteht:
   - quadrierter TD-Fehler,
   - quadrierter Fehler der Actor-Komponente, die mit einer Konstante λ_3 gewichtet ist,
   - die Konstante ϵ,
   - die Konstante ϵ_d, die für die Transitionen, die aus eigener Erfahrung des Agenten stammen, den Wert 0 annimmt.
3. Die Critic-Komponente minimiert die L_{Critic}-Kostenfunktion, die sich aus drei Summanden zusammensetzt:

$$L_{Critic}(W) = L_1(W) + \lambda_1 L_n(W) + \lambda_2 L^C_{reg}(W)$$

Für die Schätzung von $L_1(W)$ werden 1-step Zielwerte verwendet (s. Formel 3.38). $L_n(W)$ wird mit dem gleichen Vorgehen, aber auf Basis der n-step Zielwerten geschätzt. $L^C_{reg}(W)$ ist der L^2-Strafterm, λ_2 und λ_1 sind Konstanten.

4. Das Policy-Netz wird ebenso regularisiert:

$$L_{Actor}(\theta) = J(\theta) + \lambda_2 L^A_{reg}(\theta)$$

5. Im DDPG-Algorithmus wird nach jeder Episode genau ein Lernschritt ausgeführt. Im Gegensatz dazu, aktualisiert der DDPGfD-Algorithmus die beiden Komponenten mehrmals nach jeder Episode.
   Die Autoren von DDPGfD weisen darauf hin, dass die Auswahl der Trainingsdauer nach einer Episode auf ein Trade-off zwischen der Sammlung neuer Erfahrung und dem Lernen zurückgeht (vgl. [Vecerík et al., 2017], S. 3). Obwohl die off-Policy Deep RL-Algorithmen in der Lage sind, eine Policy anhand der Erfahrung zu lernen, die nach einer anderen Policy gesammelt wurde, nutzen sie zur Erkennung der Zusammenhänge in den Trainingsdaten neuronale Netze, welche in der Praxis eine endliche Kapazität besitzen. Wenn eine plötzliche Änderung der Policy die Verteilung der besuchten Staaten verändert, kann der Lernprozess schnell destabilisiert werden. Das Problem kann sich besonders in den Lernaufgaben bemerkbar machen, in denen die Existenz mehrerer Policies möglich ist, die den gleichen Gesamtreward erzielen, jedoch unterschiedliche Zustandsraum-Trajektorien haben. Ein Beispiel dafür ist das Schwingen eines Pendels im Uhrzeigersinn oder gegen den Uhrzeigersinn (vgl. [de Bruin et al., 2018], S. 39).

6. Für die Exploration wird in DDPGfD das Gaußsche Rauschen statt des Ornstein-Uhlenbeck-Prozesses verwendet.

7. Die Wichtungen der Zielnetze werden nach der „Hard-Update“ Strategie aktualisiert.

Der Algorithmus 3 präsentiert den Pseudocode von DDPGfD.

Algorithm 3 DDPGfD nach [Vecerík et al., 2017]

Input: Env Environment; θ initial policy parameters; $\bar{\theta}$ initial policy target parameters

Input: W initial action-value parameters; $\bar{W}$ initial action-value target parameters; N' target network replacement frequency; ϵ action noise

Input : $\mathcal{D}$ replay buffer initialized with demonstrations; k number of pre-training gradient updates

Output: $Q(.|W)$ action-value function (critic) and $\mu(.|\theta)$ the policy (actor)

/* Learning via interaction with the environment

for episode $e \in \{1, ..., M\}$ **do**

  Initialise state $s_0 \sim Env$

  for steps $t \in \{1, ..., EpisodeLength\}$ **do**

    Sample noise from Gaussian $n_t = \mathcal{N}(0, \epsilon)$

    Select an action $a_t = \mu(s_{t-1}, \theta) + n_t$

    Get next state and reward $s_t, r_t = T(s_{t-1}, a_t), R(s_t)$

    Add 1-step transition $(s_{t-1}, a_t, r_t, \gamma, s_t)$ to the replay buffer

    Add n-step transition $(s_{t-n}, a_{t-n}, \sum_{i=0}^{n-1} r_{t-n+1+i}\gamma^i, \gamma^n, s_t)$ to the replay puffer

  end for

  for steps $l \in \{1, ..., EpisodeLength \cdot LearningSteps\}$ **do**

    Sample a minibatch of transitions with prioritization from $\mathcal{D}$ and calculate $L_1(W)$ and $L_n(W)$ as appropriate for a given transition

    Update the critic using a gradient step with loss: $L_{Critic}(W)$

    Update the actor: $\nabla_\theta L_{Actor}(\theta)$

    if $step \equiv 0 \ (mod \ N')$ **then**

      Update the target networks: $\bar{W} \leftarrow W, \bar{\theta} \leftarrow \theta$

    end if

  end for

end for

4.4 Konzeption des umgesetzten Lernprozesses

Im Folgenden werden die Schritte des umgesetzten Lernprozesses in der Reihenfolge der Ausführung erläutert.

1. Initialisierung

- Critic-Komponente: Initialisierung der Wichtungen für zwei Q-Netze $Q^1(s_t, a_t; W^1)$, $Q^2(s_t, a_t; W^2)$ und zwei Q-Zielnetze $\bar{Q}^1(s_t, a_t; \bar{W}^1), \bar{Q}^2(s_t, a_t; \bar{W}^2)$
- Actor-Komponente: Initialisierung der Wichtungen für ein Policy-Netz $\mu(s_t; \theta)$ und ein Policy-Zielnetz $\bar{\mu}(s_t; \bar{\theta})$
- Konfiguration der Hyperparameter
- Erstellung eines 1-step Replay-Puffers rb^{1-step} und eines n-step Replay-Puffers rb^{n-step}. Jeder Replay-Puffer wird als eine leere Queue-Liste der maximalen Größe N initialisiert.

- Initialisierung der Umgebung

2. Speichern der Demonstrationen in Replay-Puffern
Demonstrationen werden in Form von Transitionen in beiden Replay-Puffern gespeichert. Um die Berechnung der Netzfehler anhand von Transitionen aus den Replay-Puffern zu erleichtern, werden sie nicht als ein 4-Tupel, sondern als ein 5-Tupel (s, a, r, s', d) gespeichert. Die Variable d enthält die Informationen darüber, ob die Episode in s' endet. Sie nimmt den Wert 1 an, falls die Episode in s' endet und den Wert 0 sonst.

3. Interaktion mit der Umgebung
Der Agent startet die Interaktion mit der Umgebung. In jedem Schritt entscheidet er sich mithilfe des Policy-Netzes für eine Aktion a_t, die mit dem Gaußschen Rauschen verrauscht wird: $\widetilde{a}_t = a_t + \mathcal{N}$. Der Agent führt $\widetilde{a}_t$ aus und erfährt seinen Reward und den neuen Zustand.

3.1. Hinzufügen der Transitionen in Replay-Puffer
In jedem Zeitschritt t wird die 1-step Transition $(s_{t-1}, a_{t-1}, r_t, s_t, d_t)$ in einen Zwischenpuffer rb^{helper}, eine Queue-Liste der maximalen Länge n, hinzugefügt. Wenn nach dem Hinzufügen der Transition der Zwischenpuffer bereits die Länge n hat, wird

- das erste Element von rb^{helper}, also $(s_{t-n}, a_{t-n}, r_{t-n+1}, s_{t-n+1}, d_{t-n+1})$ in rb^{1-step} hinzugefügt,
- die n-step Transition $(s_{t-n}, a_{t-n}, \sum_{i=0}^{n-1} \gamma^i r_{t-n+1+i}, s_t, d_t)$ erstellt und in rb^{n-step} hinzugefügt.

Auf diese Weise wird sichergestellt, dass die Replay-Puffer die gleiche Länge N haben und dass die Einträge mit dem gleichen Index in den beiden Replay-Puffern die Transitionen ab dem gleichen Zustand-Aktion-Paar enthalten. Beim Erreichen der maximalen Größe der Replay-Puffer wird die eigene Erfahrung des Agenten mit neuen Transitionen überschrieben.
Der Anhang A.3 enthält eine detaillierte Beschreibung der Befüllung des 1-step und des n-step Replay-Puffer.
Jeder Eintrag in rb^{1-step} erhält eine Priorität. Die neuen 1-step Transitionen werden mit der maximalen Priorität gespeichert.

3.2. Ausführung von $k * length$ Lernschritten
Die Länge der Episode *length* wird als Gesamtanzahl der Schritte innerhalb einer Episode definiert. Nach jeder Episode führt der Agent $k * length$ Lernschritte aus, wobei k ein durch den Entwickler festgelegter Parameter ist. Die folgenden Unterpunkte 3.2.1- 3.2.2 beschreiben die Ausführung *eines* Lernschrittes.

3.2.1. Sampling von 1-step und n-step Mini-Batches
a) Sampling eines Mini-Batches aus dem 1-step Replay-Puffer
Ein 1-step Mini-Batch enthält m Einträge mit jeweils folgenden Informationen:

- 1-step Transition,
- Index der Transition im Replay-Puffer,
- IS-Wichtung der Transition.

Die Transitionen werden nach dem im Abschnitt 3.2.6.4 beschriebenen Vorgehen aus dem 1-step Replay-Puffer ausgewählt. Für jede Transition wird eine IS-Wichtung nach der Formel (3.43) mit $\beta = 1$ berechnet.
b) Sampling eines Mini-Batches aus dem n-step Replay-Puffer
Aus dem n-step Replay-Puffer werden die Transitionen extrahiert, die die gleichen Indizes wie die entnommenen 1-step Transitionen haben.

3.2.2. Aktualisierung der Critic-Komponente
$Q^1(s_t, a_t; W^1)$ und $Q^2(s_t, a_t; W^2)$ minimieren die in dem DDPGfD-Algorithmus vorgeschlagene Kostenfunktion $L_{Critic}(W^k)$:

$$L_{Critic}(W^k) = L_1(W^k) + \lambda_1 L_n(W^k) + \lambda_2 L_{reg}^C(W^k), \quad k = 1, 2, \tag{4.1}$$

$L_1(W^k)$ und $L_n(W^k)$ werden auf die Modifikationen „Clipped-Double Q-Learning“ und „Glättung der Policy“ des TD3-Algorithmus sowie die PER-Technik angepasst und berechnen sich über einen Mini-Batch der Größe m wie folgt:

$$L_1(W^k) = \frac{1}{m} \sum_{i=0}^{m-1} w_i (\underbrace{y_i^1 - Q^k(s_i, a_i; W^k)}_{1-step\ \delta_i})^2, \tag{4.2}$$

$$L_n(W^k) = \frac{1}{m} \sum_{i=0}^{m-1} w_i \underbrace{(y_i^n - Q^k(s_{min(i+n;\ T)}, a_{min(i+n;\ T)}; W^k))}_{n-step\ \delta_i})^2, \tag{4.3}$$

w_i ist die IS-Wichtung einer Transition, δ_i ist der TD-Fehler, T ist der letzte Zeitschritt. Die entsprechenden 1-step-Zielwerte y_i^1 sind:

$$y_i^1 = r_{i+1} + (1 - d_{i+1}) \cdot \gamma \cdot \min_{k=1,2} \bar{Q}^k(s_{i+1}, \bar{\mu}(s_{i+1}; \bar{\theta}) + \varphi; \bar{W}^k), \tag{4.4}$$
$$\varphi \sim clip((\mathcal{N}, 0, \sigma), -c, c)$$

Die n-step-Zielwerte y_i^n sind:

$$y_i^n = \sum_{j=0}^{min(n-1;\ T-i-1)} \gamma^j r_{i+1+j} + (1 - d_{\bar{t}}) \cdot \gamma^n \cdot \min_{k=1,2} \bar{Q}^k(s_{\bar{t}}, \bar{\mu}(s_{\bar{t}}; \bar{\theta} + \varphi); \bar{W}^k),$$
$$\bar{t} = min(i + n;\ T),$$
$$\varphi \sim clip((\mathcal{N}, 0, \sigma), -c, c) \tag{4.5}$$

3.2.3. Aktualisierung der Actor-Komponente

Die Kostenfunktion des Policy-Netzes besteht aus zwei Summanden, wobei der erster Summand $J(\theta)$ maximiert und der zweite Summand $\lambda_2 L^A_{reg}(\theta)$ minimiert werden muss:

$$L_{Actor}(\theta) = J(\theta) + \lambda_2 L^A_{reg}(\theta) \tag{4.6}$$

Der Gradient von L_{Actor} bzgl. θ ist

$$\nabla_\theta L_{Actor}(\theta) = \nabla_\theta J(\theta) + \lambda_2 \nabla_\theta L^A_{reg}(\theta) =$$
$$= \nabla_a Q^1(s_t, \mu(s_t;\theta); W^1) \cdot \nabla_\theta \mu(s_t;\theta) + \lambda_2 \nabla_\theta L^A_{reg}(\theta)$$

Da die Durchführung des Gradientenschrittes mithilfe eines Framework (s. Abschnitt 5.2) erfolgt, das eine automatische Differenzierung bereitstellt, kann der Gradient direkt bzgl. der Parameter θ berechnet werden. Des Weiteren kann der erste Summand negiert werden, sodass für die Aktualisierung des Policy-Netzes ein Schritt des Gradientenabstiegs durchgeführt werden muss. Für einen Mini-Batch der Größe m wird also $\nabla_\theta L_{Actor}(\theta)$ wie folgt berechnet:

$$L_{Actor}(\theta) = \frac{1}{m} \sum_{i=0}^{m-1} -\nabla_\theta Q^1(s_i, \mu(s_i;\theta); W^1) + \lambda_2 \nabla_\theta L^A_{reg}(\theta) \tag{4.7}$$

3.2.4. Aktualisierung der Zielnetze

Die Zielnetze werden nach jedem Lernschritt nach der „Soft-Update“ Strategie aktualisiert (s. Formel 3.37).

3.2.5. Aktualisierung von Prioritäten

Für die Transitionen des 1-step Mini-Batches werden die Prioritäten neu berechnet. Für die Priorisierung wird die PER-Technik (proportionale stochastische Priorisierung, $\alpha < 1$) eingesetzt, wobei die Berechnung einer Priorität modifiziert wird.

Die Priorität einer Transition setzt sich aus dem quadrierten 1-step TD-Fehler von $Q^1(s, a; W^1)$ und von $Q^2(s, a; W^2)$; dem mit λ_1 gewichteten quadrierten n-step TD-Fehler von $Q^1(s, a; W^1)$ und von $Q^2(s, a; W^2)$; dem mit λ_3 gewichteten Actor-Fehler L_{Actor} sowie der Konstanten ϵ und ϵ_d.

4. Logging, Speicherung der Netzwichtungen

Während des Lernprozesses werden folgende Daten in getrennten .csv-Dateien gespeichert:

- Gesamtreward für jede Trainings- und Testepisode,
- L_{Critic} und L_{Actor} (Durchschnitt über 100 Lernschnitte),
- Anzahl der Demonstrationen, die für einen Lernschritt verwendet werden (Durchschnitt über 100 Lernschnitte),
- Erfahrung des Agenten (nur in dem Ball-in-a-Cup-Szenario)

5. Testen der bis dahin gelernten Policy
In einem bestimmten Takt wird zwischen den Trainingsepisoden eine festgelegte Anzahl von Testepisoden ausgeführt. Dabei wird die Policy des Agenten nicht verrauscht. Die Transitionen, die während eines Tests entstehen, werden nicht für das Lernen verwendet und daher nicht in den Replay-Puffern gespeichert.
Die Schritte 3-5 werden solange wiederholt, bis die maximale Anzahl der Trainingsepisoden erreicht ist. Die Abbildung 4.4 fasst einige Informationsflüsse im oben beschriebenen Lernprozess zusammen.

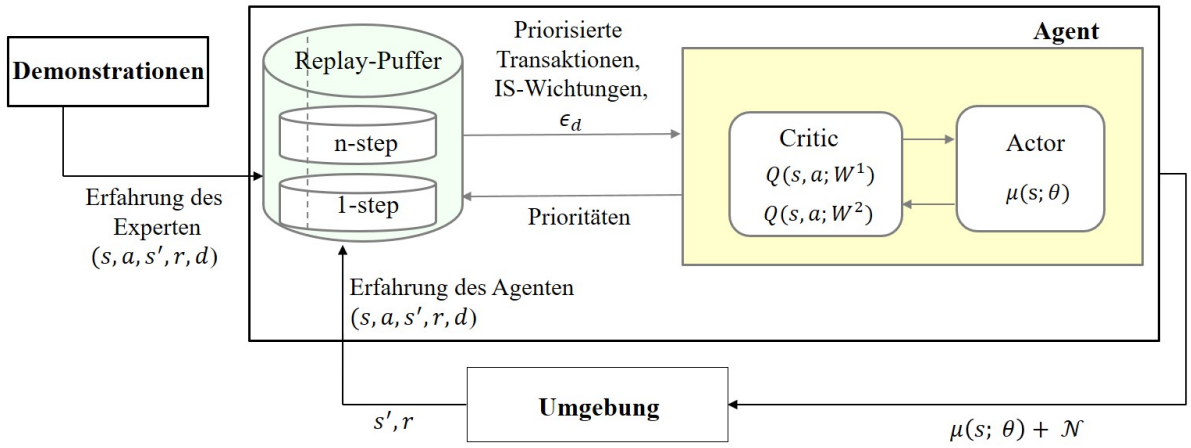


Abbildung 4.4: Lösungsansatz
Quelle: Eigene Darstellung

4.5 Formalisierung des Spiels Ball-in-a-Cup

Die Formalisierung des MDPs und die Auswahl der Netzarchitekturen für das Spiel hängt u. a. von folgenden Entscheidungen ab:

1. Soll der Agent anhand von Bildern oder manuell ausgewählten Merkmalen der Umgebung lernen?
   Obwohl das Lernen aus unstrukturierten Daten wie Bilder in ML erwünscht ist, kann der Lernprozess in diesen Einstellungen viel Zeit in Anspruch nehmen. In [Schwab et al., 2019] wurde gezeigt, dass der Lernprozess bei Ball-in-a-Cup schneller verläuft, wenn die Zustandsbeschreibung bereits extrahierte Merkmale enthält (vgl. [Schwab et al., 2019], S. 1). Um das Lernproblem relativ einfach zu halten, wurde die Entscheidung getroffen, mit manuell ausgewählten Merkmalen zu arbeiten.
2. Darf der Roboter für die Lösung der Aufgabe alle Armgelenke verwenden?
   Während der Aufnahme von Demonstrationen wurde festgestellt, dass für die Lösung der Aufgabe die Steuerung von lediglich drei Armgelenken ausreichend ist: Die Bewegung der rechten Schulter nach rechts und nach links (`RShoulderRoll`)

sowie nach oben und nach unten (`RShoulderPitch`) und die Bewegung des rechten Handgelenkes nach rechts und nach links (`RWristYaw`).

3. Welche Methode des NAOqi Framework wird zur Bewegungssteuerung benutzt?
   Zur Änderung der Gelenkwinkel wird die *angleInterpolation()*-Methode des ALMotion-Moduls verwendet.

Zustandsbeschreibung Zu jedem Zeitpunkt beobachtet der Roboter einen 27-dimensionalen Zustand, der aus folgenden Elementen besteht:

- 0-2: absolute Winkel der Gelenke `RShoulderRoll`, `RShoulderPitch`, `RWristYaw` in Grad: $\varphi^{RShoulderRoll}$, $\varphi^{RShoulderPitch}$, $\varphi^{RWristYaw}$
- 3-5: aktuelle Position des RArm-Effektors im Koordinatensystem `FRAME_ROBOT`
- 6-14: die Position des Balls in den letzten drei Zeitschritten im Koordinatensystem `FRAME_ROBOT`
- 15-17: Differenz zwischen Gelenkwinkeln zum Zeitpunkt *(t-1)* und dem aktuellen Zeitpunkt t
- 18-26: Differenz zwischen Positionen des Balls zum Zeitpunkt *(t-1)* und dem aktuellen Zeitpunkt t sowie zwischen *(t-2)* und *(t-1)*, zwischen *(t-3)* und *(t-2)*
- 27: Verzögerung
  Bei der Aufnahme von Demonstrationen wurde der Zustand des Agenten in einem zeitlichen Abstand von 0.15s erfasst. Wenn der Agent selbst mit der Umgebung interagiert, dauert ein Schritt aufgrund des Rechenaufwands für die Aktionsauswahl oder der blockierenden Methode zur Bewegungsausführung länger als 0.15s.

Aktionen Der Roboter ist in der Lage, in jedem Schritt eine 3-dimensionale Aktion a auszuführen. Jede Dimension entspricht der Änderung eines Gelenkwinkels relativ zum aktuellen Winkel:

$$a = [\Delta\varphi^{RShoulderRoll}, \Delta\varphi^{RShoulderPitch}, \Delta\varphi^{RWrist}]$$

Für jedes Gelenk bestimmt Softbank Robotics den Bewegungsbereich $range_{global}$ (Tabelle 4.1, Spalte 2 und Abbildung 3). Aus den Demonstrationen wurde der Bewegungsbereich $range_{demo}$ ermittelt, den der Experte für die Lösung der Aufgabe benutzt hat (s. Tabelle 4.1, Spalte 3). Anhand dieser Daten und einigen weiteren Aspekten, wie z. B. der Blickwinkel des Roboters, der den Ball beobachtet, oder die Konstruktion der Umgebung (die Position des Tisches, die Pose des Roboters), konnte ein Bereich $range_{interaction}$ bestimmt werden, in dem der Roboter sich bei der Interaktion mit der Umgebung bewegen darf (Tabelle 4.1, Spalte 4). Dieser wurde kleiner als $range_{global}$ jedoch ein wenig größer als $range_{demo}$ gehalten.

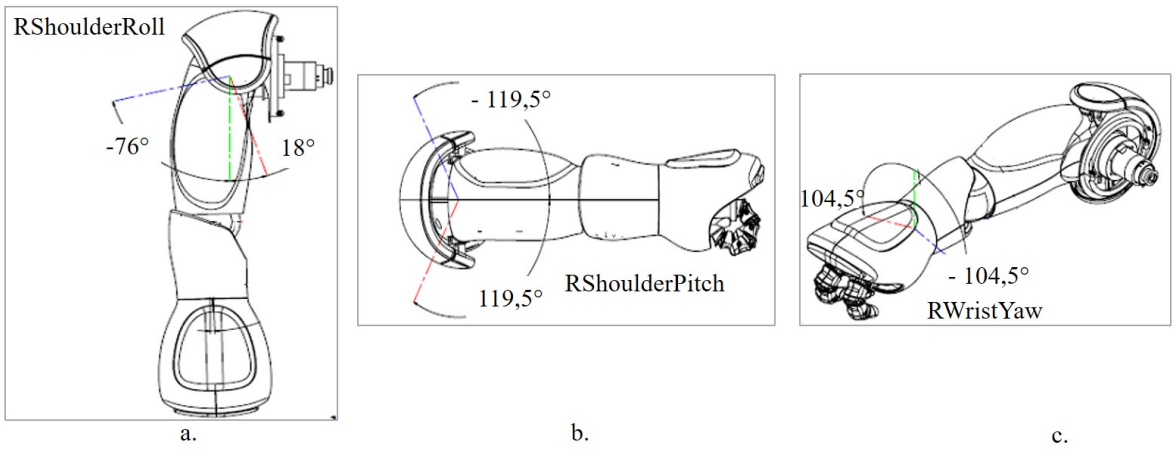


Abbildung 4.5: Visualisierung der Gelenkwinkel
Quelle: [SoftBank Robotics, 2017]

Gelenk	$range_{global}$	$range_{demo}$	$range_{interaction}$
RShoulderRoll	[-76, 18]	[-10.2; 18]	[-31; 18]
RShoulderPitch	[-119.5, 119.5]	[-0.17; 21.98]	[-1; 22]
RWristYaw	[-104.5, 104.5]	[-19.87; 65.21]	[-36; 95]

Tabelle 4.1: Bewegungsbereiche der Gelenke in Grad
Quelle: [SoftBank Robotics, 2017]

Für die Ausführung einer Aktion ist die Zeit 0.15s vorgesehen. Um die Sicherheit des Roboters zu gewährleisten, wurde für jedes Gelenk die maximale Winkeländerung festgelegt (Tabelle 4.2, Spalte 3).

Gelenk	Max. Winkeländerung pro Schritt in Demos (Grad)	Max. erlaubte Winkeländerung pro Schritt bei der Agent-Umgebung Interaktion (Grad)
RShoulderRoll	15.4	20
RShoulderPitch	6.5	20
RWristYaw	29.97	30

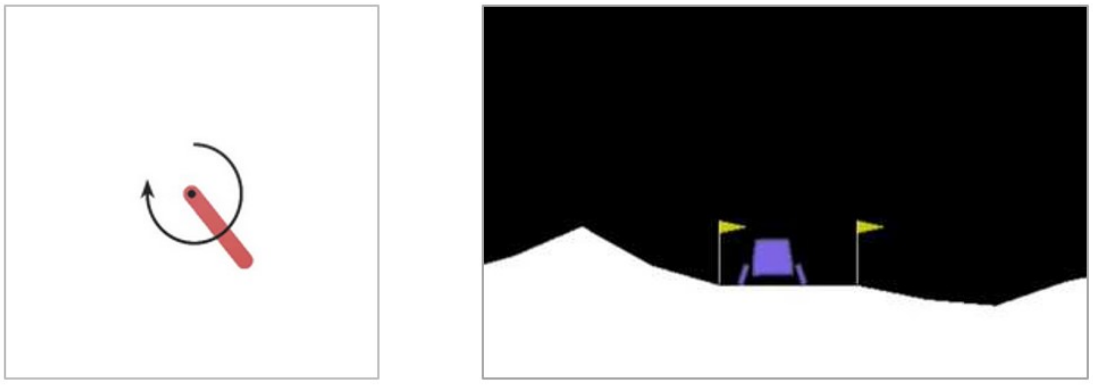
Tabelle 4.2: Maximale Winkeländerungen der Gelenke pro Schritt
Quelle: Eigene Darstellung

Reward In jedem Zeitschritt bekommt der Roboter den Reward „+5“, falls der Ball sich im Zielbereich befindet, ansonsten „0“. Wenn der Reward „+5“ in zwei nacheinander folgenden Zeitschritten auftritt, gilt das Spiel als gewonnen. Die Episode wird gestoppt und der Roboter erhält den zusätzlichen Reward „+45“. Wenn diese Bedingung nicht erfüllt ist, endet die Episode im Zeitschritt $T = 100$ mit dem Reward „-50“, wenn der Ball sich *nicht im* Zielbereich befindet und „+50“, wenn der Ball sich *im* Zielbereich befindet.

Der Zielbereich für den Ball wird anhand der Position des Endeffektors `RArm` und des Winkels des Gelenkes `RWristYaw` berechnet. Der Roboter hält den Becher in der Hand, wo der Endeffektor `RArm` liegt (vgl. Abb. 2.1), zudem sind die Parameter des Bechers bekannt. Die Zielposition für den Ball kann relativ zur Position von `RArm` definiert werden. Wenn die Öffnung des Bechers sich höher auf der z-Achse befindet als der Endeffektor, liegt der Zielbereich weiter auf der x-Achse (mind. +4 cm max. +10 cm), links auf der y-Achse (mind. +1 cm, max. +9 cm) und höher auf der z-Achse (mind. 0 cm, max. +10 cm). Wenn die Öffnung des Bechers sich niedriger auf der z-Achse befindet als der Endeffektor, wird der Zielbereich auf der z-Achse nach unten (-3 cm) erweitert. Um zu schätzen, ob die Öffnung oben oder unten ist, wird der Winkel des `RWristYaw`-Gelenkes benutzt. Wenn `RWristYaw` > 0 ist, ist die Öffnung oben, sonst unten.

4.6 Virtuelle Umgebungen

Das Lernen mit dem vorgestellten Lösungsansatz kann grundsätzlich aufgrund einer fehlerhaften Implementierung, einer unpassenden Kombination von Hyperparametern oder einer ungünstigen Formalisierung des MDPs (insbesondere Rewardfunktion) scheitern. Auch wenn der Algorithmus nicht in der Lage ist, die Komplexität der Aufgabe zu bewältigen, wird das Lernen nicht erfolgreich sein.
Um sicherzustellen, dass die Implementierung der Lernschritte korrekt ist, und um die Intuition für die Bestimmung der Hyperparameter für die Aufgabe zu bekommen, wird der Algorithmus in zwei virtuellen Umgebungen von OpenAI [Brockman et al., 2016] getestet: „Pendulum-v0“ und „LunarLanderContinuous-v2“ (s. Abbildung 4.6).



a. Pendulum-v0 b. LunarLanderContinuous-v2

Abbildung 4.6: Virtuelle Umgebungen Pendulum-v0 und LunarLanderContinuous-v2
Quelle: [SoftBank Robotics, 2017]

Bei der Auswahl der Umgebungen wurden Eigenschaften des Lernszenarios (s. Abschnitt 4.4) berücksichtigt. In ausgewählten Lernumgebungen soll ein virtueller Roboter

eine Strategie zur Steuerung erlernen. Aktions- und Zustandsmengen sind in den beiden Lernproblemen kontinuierlich.
OpenAI stellt eine API zur Interaktion mit der Umgebung bereit:

- `env.reset()` - setzt die Umgebung zurück und gibt den Startzustand als Rückgabewert zurück
- `env.step()` – nimmt die gewünschte Aktion des Agenten als Parameter an und gibt das 4-Tupel `(observation, reward, done, info)` zurück. `observation` und `reward` beschreiben den Folgezustand und den Erfolg des Agenten, `done` beinhaltet die Information darüber, ob die Episode in dem Folgezustand endet, `info` enthält zusätzliche Daten, die vom Nutzer spezifiziert werden
- `env.observation_space()`/`env.action_space()` – geben den Typ (diskret/kontinuierlich) und die Dimensionalität des Zustands-/Aktionsraums zurück

Im Folgenden wird ein Überblick über die verwendeten Umgebungen gegeben.

MDP für Pendulum-v0 „Pendulum“ ist eine klassische Kontrollaufgabe, die darin besteht, ein Pendel in Schwingung zu versetzten und aufrecht zu halten. Die Episode beginnt in einer beliebigen Position mit einer zufälligen Geschwindigkeit und endet in 200 Schritten. Der *Zustand* des Pendels wird durch einen 3-dimensionalen Vektor beschrieben. Die ersten zwei Elemente geben den Winkel des Pendels an und können die Werte im Bereich $[-1.0;\ 1.0]$ annehmen. Das dritte Element ist die Winkelgeschwindigkeit, die durch den Bereich $[-8.0;\ 8.0]$ eingeschränkt wird.
Aktionen werden durch 1-dimensionale Vektoren im Wertebereich $[-2.0;\ 2.0]$ repräsentiert. Eine Aktion gibt die Höhe der linken oder der rechten Krafteinwirkung auf das Pendel an. Die *Rewardfunktion* spezifiziert das Ziel des Agenten als das Erreichen des Winkels 0°(also der vertikalen Position) mit der geringsten Drehgeschwindigkeit und der niedrigsten Krafteinwirkung. Die Rewards sind von $-\pi$ bis π normalisiert, sodass ein einzelner Reward einen Wert im Bereich [-16.2736044; 0] annehmen kann.

MDP für LunarLanderContinuous-v2 Der Landeplatz befindet sich auf den Koordinaten (0, 0). Die Koordinaten sind die ersten zwei Elemente des Zustandsvektors. Der *Zustand* wird durch einen 8-dimensionalen Vektor beschrieben.
Aktionen sind 2-dimensionale Vektoren. Mit dem ersten Element, der im Bereich $[-1;\ 1]$ liegt, reguliert der Agent die Motoren ([-1..0] aus, $[0 \ldots 1]$ an). Mit dem zweiten Element, dass einen Wert im Bereich $[-1.0;\ 1.0]$ annehmen kann, steuert der Agent das Fahrzeug nach links, nach rechts oder nach unten.
Der *Reward* für den Wechsel vom oberen Bildschirmrand zum Landepunkt und zur Geschwindigkeit 0 beträgt etwa „+100“ – „+140“ Punkte. Wenn sich das Raumschiff vom Landeplatz entfernt, bekommt der Agent einen negativen Reward. Die Episode endet, wenn das Raumschiff abstürzt oder landet, wofür der Agent zusätzlich „-100“ oder „+100“

Punkte erhält, oder wenn die maximale Länge der Episode (1000 Schritte) erreicht ist. Für jeden Bodenkontakt mit dem Fuß wird der Agent mit „+10“ belohnt, jede Erhöhung des Antriebs wird mit „-0.3“ für jeden Frame bestraft. Der Treibstoff ist unendlich.
Im Rahmen dieser Arbeit wird neben der vorgestellten Rewardfunktion von OpenAI eine Sparse-Rewardfunktion spezifiziert: der Agent bekommt den Reward „+100“, wenn die Maschine landet, und „-100“ bei einem Absturz. Die maximale Länge der Episode beträgt in dieser Einstellung 350 Schritte.

4.7 Zusammenfassung

Zu Beginn dieses Abschnittes wurden die Anforderungen an den Lernalgorithmus definiert. Danach wurden vier Ansätze zur Kombination von RL und IL vorgestellt: DQfD [Hester et al., 2017], Ape-X DQfD [Pohlen et al., 2018], DDPGfD [Vecerík et al., 2017] und die Arbeit von Nair et al. [Nair et al., 2017]. Aufbauend auf der Analyse der Eigenschaften des Ball-in-a-Cup-Spiels, wurde der **DDPGfD** Algorithmus zur Lösung der Aufgabe ausgewählt und auf das Clipped Double Q-Learning und die Glättung der Target-Policy (inspiriert von **TD3**) erweitert. Als Nächstes wurden die Schritte des umgesetzten Lernprozesses erläutert. Zum Schluss wurde das Ball-in-a-Cup-Spiel mit dem NAO-Roboter als ein MDP formalisiert und die virtuellen Umgebungen „Pendulum-v0“ und „LunarLanderContinuous-v2“ vorgestellt, in denen der verwendete Algorithmus neben der Aufgabe mit dem realen Roboter untersucht wird.

5 Implementierung

5.1 Entwicklungsumgebung

Die Entwicklung findet auf einem PC mit **Windows 10** statt, der über den **PyCharm Professional Editor** mit einem virtuellen Python-Interpreter konfiguriert ist. Die lokalen Änderungen im Code werden vom Editor automatisch erkannt und auf einen festgelegten Entwicklungsendpunkt auf der **Linux**-Maschine übertragen.
Die für die Aktualisierung der Netzwichtungen benötigten Berechnungen werden auf der **NVIDIA Titan RTX** Grafikkarte mit **CUDA 10.1** ausgeführt. Für die Implementierung wird die Programmiersprache **Python** ausgewählt. Die Entscheidung wurde anhand von folgenden Überlegungen getroffen:

- Python wird von NAOqi und OpenAI unterstützt
- die meisten Implementierungsbeispiele, Blog-Posts und Lehrmaterialien für RL und IL sind in Python geschrieben

In dem Lernszenario mit dem NAO-Roboter wird **Python 2.7** verwendet, da die höheren Python-Versionen von NAOqi 2.14 nicht unterstützt werden.
Für das Lernen in den virtuellen Umgebungen wird **Python 3.6** benutzt, um die spätere Erweiterung der Anwendung auf die OpenAI Robotics Umgebungen zu ermöglichen.
Um die Konflikte zwischen unterschiedlichen Python-Versionen zu vermeiden, werden alle notwendigen Frameworks und Bibliotheken in zwei unabhängigen Python-Umgebungen mithilfe vom `virtualenv`-Paket[7] installiert.
Der Code wird mit **Git** versioniert und auf *gitlab.com* verwaltet. Als Ausgangspunkt für die Entwicklung wurde eine lokale Kopie von folgenden Repositories erstellt:

- OpenAI Baselines [Dhariwal et al., 2017],
- Keras RL [Plappert, 2016],
- Medipixel RL Algorithms [Medipixel Inc., 2019],
- TD3 [Fujimoto et al., 2019].

[7] https://wiki.ubuntuusers.de/virtualenv/

5.2 Bibliotheken und Frameworks

In der Arbeit werden folgende Python-Bibliotheken verwendet:

- **Seaborn und Matplotlib**
  Seaborn und Matplotlib werden zur Datenvisualisierung eingesetzt.
- **Pandas (Python Data Analysis Library)**
  Pandas ist eine Open-Source-Bibliothek, die die Datenanalyse für Python durch die Bereitstellung von benutzerdefinierten Datenstrukturen wie DataFrames und Series ermöglicht. Die Bibliothek wird zum Auslesen und zur Erstellung der Demonstrations- und Log-Daten verwendet.
- **NumPy**
  Numpy wird für die Operationen mit N-dimensionalen Arrays auf der CPU benutzt.

Die Steuerung des NAO-Roboters erfolgt mithilfe des **NAOqi 2.14** Framework (s. Abschnitt 2.3).
Für die Erstellung und das Training der neuronalen Netze wird das **PyTorch** Framework verwendet. Die Daten werden in dem Framework in Form von Tensoren (spezifischen multidimensionalen Arrays) repräsentiert. PyTorch stellt eine API zur Ausführung von mathematischen Operationen mit Ternsoren auf NVIDIA GPU mithilfe von CUDA-Toolkit bereit. Für die Erstellung eines neuronalen Netzes bietet sich das Paket `torch.nn` an. Mithilfe des Pakets `autogad` kann die automatische Differenzierung der Tensoren durchgeführt werden.

5.3 Aufnahme von Demonstrationen

In *virtuellen Umgebungen* wurde der Agent zu Beginn ohne Demonstrationen trainiert, bis er in der Lage war, eine gewünschte Leistung zu erzielen. Im „Pendulum-v0“ wurde das Training gestoppt, wenn der Agent in 100 Testepisoden einen durchschnittlichen Gesamtreward größer als „-130“ bekam. In „LunarLanderContinuous-v2“ musste der Agent in 100 Testepisoden einen durchschnittlichen Gesamtreward im Wert von mehr als „200“ erhalten. Die Netzwichtungen wurden gespeichert und eine benötigte Anzahl von Demonstrationen aufgenommen.
Im *Ball-in-a-Cup Spiel* wurde die Strategie des Experten durch kinästhetische Demonstrationen erfasst. Die Motoren der Gelenke `RShoulderRoll`, `RShoulderPitch` und `RWrist` mussten ausgeschaltet werden, während die Motoren der Gelenke `RElbowRoll`, `RHipPitch` und `LHipPitch` an blieben, um die Bewegung des Torsos und des rechten Ellenbogengelenks auszuschließen.
Die erfassten Demonstrationen wurden analysiert und auf Duplikate, fehlende Werte und Anomalien geprüft. Es wurde festgestellt, dass die Sensoren des Roboters Gelenkwinkel nicht genau ermitteln. So kamen beispielsweise in Demonstrationen die Winkelangaben

vor, die die globalen Bewegungsbereiche $range_{global}$ (s. Tabelle 4.1) überschreiten. Diese Erkenntnis sollte bei der Skalierung der Aktionen (s. Abschnitt 5.4.4) berücksichtigt werden.

5.4 Module der Softwareanwendung

Die Softwareanwendung beinhaltet folgende Komponenten:

- `Agent` - steuert den Lernprozess und spielt die Rolle eines Bindeglieds zwischen allen anderen Komponenten.
- `Models` – implementiert die neuronalen Netze für die Actor- und die Critic-Komponente.
- `Environment` – enthält die Methoden zur Steuerung des Agenten während der Interaktion mit der Umgebung.
- `Replay-Buffer` – ist für die Verwaltung des 1-step- und des n-step Replay-Puffers zuständig.
- `Logger` – speichert die vom Entwickler festgelegten Daten, z. B. die Erfahrung des Agenten, in einer externen Datei.
- `Visualizer` – enthält die Methoden zur Visualisierung der Daten.

In der Datei `run.py` werden folgende Größen definiert, die beim Starten der Anwendung an die relevanten Klassen weitergegeben werden:

- die Hyperparameter des Lernalgorithmus,
- die Dimensionalität der Zustände und Aktionen,
- die maximale Winkeländerung pro Schritt,
- numerische Werte der Rewardfunktion (z. B. `reward_win = 50`, `reward_lose = -50`, `reward_step = 0`, `reward_middle = 5`).

Im Folgenden werden einige Module und Klassen der aufgelisteten Komponenten näher erläutert.

5.4.1 Actor und Critic

Das Policy-Netz wird mithilfe der Klasse `Actor` und zwei Q-Netze mit der Klasse `Critic` implementiert. Die Zielnetze werden als tiefe Kopien der entsprechenden lokalen Netze initialisiert. `Critic` und `Actor` sind Unterklassen von `torch.nn.module.Module` des PyTorch-Framework. Die Abbildung 5.1 zeigt das Klassendiagramm für die `Actor`- und `Critic`-Komponente.

Forwardpropagierung Die Methode `forward()` der `Actor`-Klasse erwartet einen Zustand s als Eigabe und gibt die vom Netz vorhergergesagte bestmögliche Aktion zurück. Die Methode `forward()` der `Critic`-Klasse berechnet die Ausgabe der beiden Q-Netze für ein Zustand-Aktion-Paar (s, a). Die Methode `forward_q1()` führt die Forwardpropagation nur für das erste Q-Netz durch.

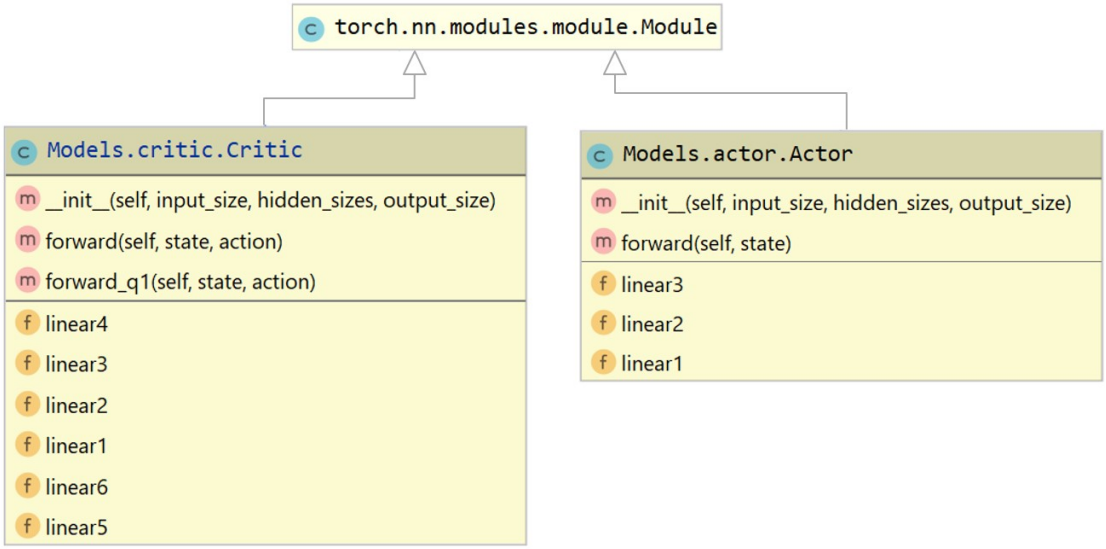


Abbildung 5.1: Hierarchie der Klassen `Actor` und `Critic`. Methoden und Attribute der Klasse `torch.nn.module.Module` wurden ausgeblendet.
Quelle: Eigene Darstellung

Optimierung und Hyperparameter Für die Optimierung der Netzwichtungen wird der Adam Optimierer (Eng.: optimizer, s. Pseudocode A.1.1) benutzt. Die Implementierung des Optimierers wird durch die Klasse `torch.optimizer.Adam` des PyTorch-Framework bereitgestellt.
In [Vecerík et al., 2017] werden keine Angaben zu eingesetzten Netzarchitekturen gemacht. Aus diesem Grund ähnelt sich die Konfiguration der netzbezogenen Hyperparameter in der vorliegenden Arbeit der in [Lillicrap et al., 2016] verwendeten Konfiguration. In den Standardeinstellungen haben die Netze die folgenden Architekturen:

- Policy-Netz
  - Deep-Feedforward-Netz mit zwei versteckten Schichten: [128; 128]
  - Aktivierungsfunktionen: LeakyReLU in den versteckten Schichten, Tanh in der Ausgabeschicht
  - Optimierung: Adam Algorithmus mit der Lernrate 0.0003 und dem L^2-Strafterm 0.01
  - $\tau = 0.005$ für das Update des Zielnetzes
  - Wichtungen und Bias-Neuronen der letzten Schicht werden mit Zufallszahlen im Bereich $[-3 \times 10^{-3}, -3 \times 10^{-3}]$ initialisiert

- Q-Netz 1, Q-Netz 2
  - Deep-Feedforward-Netz mit zwei versteckten Schichten: [128; 128]
  - Aktivierungsfunktionen: LeakyReLU in den versteckten Schichten, keine Aktivierungsfunktion in der Ausgabeschicht
  - Optimierung: Adam Algorithmus mit der Lernrate 0.003 und dem L^2-Strafterm 0.01
  - $\tau = 0.005$ für das Update der Zielnetze
  - Wichtungen und Bias-Neuronen der letzten Schicht werden mit Zufallszahlen im Bereich $[-3 \times 10^{-4}, -3 \times 10^{-4}]$ initialisiert

5.4.2 Replay-Puffer

Das Klassendiagramm in der Abbildung 5.2 verdeutlicht die Umsetzung des 1-step und des n-step Replay-Puffers. Eine gerichtete Assoziation zwischen zwei Klassen weist darauf hin, dass in der Quellklasse ein Objekt der Zielklasse verwendet wird. Der Assoziationsname entspricht dabei dem Objektnamen.

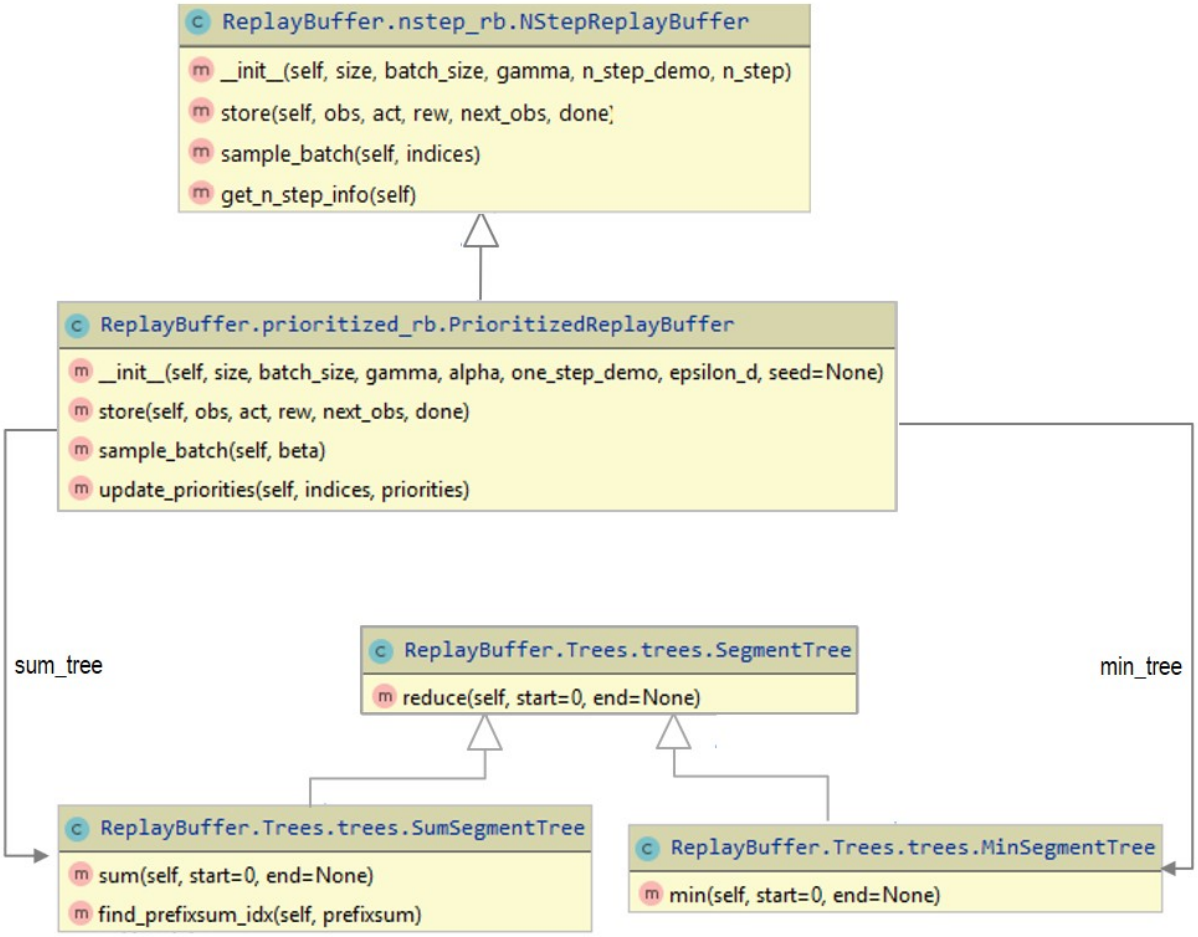


Abbildung 5.2: Klassendiagramm zur Veranschaulichung der Implementierung von Replay-Puffern. Attribute und Methoden, die nicht zum Verständnis beitragen, werden hier vernachlässigt .
Quelle: Eigene Darstellung

Der 1-step Replay-Puffer wird mit der Klasse `PrioritizedReplayBuffer` und der n-step Replay-Puffer mit der Klasse `NStepRelayBuffer` implementiert, wobei `PrioritizedReplay Buffer` eine Unterklasse von `NStepRelayBuffer` mit `n_step=1` ist.
Die Klasse `PrioritizedReplayBuffer` verwaltet die Priorisierung der Transitionen. Um ein 1-step Mini-Batch für die Aktualisierung der Netzwichtungen zu entnehmen, wird die Methode `sampe_batch()` der Klasse `PrioritizedReplayBuffer` aufgerufen, die u. a. die Prioritäten der ausgewählten Transitionen, deren Indices und IS-Wichtungen zurückliefert. Der entsprechende n-step Mini-Batch wird mithilfe der Methode `sample_batch()` der Klasse `NStepReplayBuffer` entnommen. Nach jeder Aktualisierung der Netzwichtungen werden die Prioritäten der Transitionen aus dem Mini-Batch über den Aufruf der Methode `update_priorities()` von `PrioritizedReplayBuffer` neu gesetzt.

```
1 states, actions, rewards, next_states, dones, weights, indices, eps_d =
      self.replay_buffer_demos_1_step.sample_batch(self.beta)
```

```
2 states_n, actions_n, rewards_n, next_states_n, dones_n = self.
    replay_buffer_demos_n_step.sample_batch(indices)
3 # TO-DO: Update Critic and Actor components
4 new_priorities = self.update_priorities(critic_loss_element_wise,
    actor_loss_element_wise, eps_d)
5 self.replay_buffer_demos_1_step.update_priorities(indices, new_priorities)
```

Listing 5.1: Sampling des 1-step und des n-step Mini-Batches

5.4.3 Agent

Die Klasse `Agent` steuert den Lernprozess. Sie enthält die Methoden für

- die Ausführung einer oder mehrerer Trainingepisoden `train()`,
- die Ausführung einer oder mehrerer Testepisoden `run_test()`,
- die Befüllung der Replay-Puffer mit Demonstrationen `fill_demos()`,
- das Vortrainieren des Agenten, bevor er die Interation mit der Umgebung anfängt `pretrain()`,
- die Befüllung der Replay-Puffern mit der Erfahrung des Agenten, die bei einem anderen Testversuch gesammelt wurde `pretrain_from_own_exp()`,
- die Aktualisierung der Q-Netze und des Policy-Netzes `update_models()`,
- die Aktualisierung der Prioritäten `update_priorities`(),
- die Aktualisierung der Zielnetze `soft_update()`,
- die Speicherung der Netzwichtungen `save_weights()`,
- das Laden der Netzwichtungen `load_weights()`.

In `update_models()` werden die Fehler der Q-Netze nach der Gleichung (4.1) und der Fehler des Policy-Netzes nach der Gleichung (4.6) berechnet, jedoch ohne den letzten Summanden $\lambda_2 L^C_{reg}$ und $\lambda_2 L^A_{reg}$. Die L^2-Regularisierung wird von dem Adam-Optimierer bei der Aktualisierung der Netzwichtungen automatisch durchgeführt.

Aktualisierung der Q-Netze Zu Beginn werden die 1-step Zielwerte für das Update der Q-Netze nach der Formel (4.4) ausgerechnet. Das Listing 5.2 schildert die Implementierung der Formel. Die Variable `target_Qs` bezeichnet die benötigen 1-step Zielwerte y^1_i.

```
1         noise = (torch.randn_like(actions) * policy_noise).clamp(-
              noise_clip, noise_clip)
2         next_actions = (self.actor_target(next_states) + noise).clamp(-1,1)
3         target_Qs1, target_Qs2 = self.critic_target(next_states,
              next_actions)
4         target_Qs = torch.min(target_Qs1, target_Qs2)
5         target_Qs = rewards + (gamma * target_Qs * (1-dones)).detach()
```

Listing 5.2: Berechnung der 1-step Zielwerte für das Update der Q-Netze

Als nächstes wird die Forwardpropagation in den beiden Q-Netzen durchgeführt, die Fehler zwischen den vorhersagten Ausgaben und den Zielwerten für jede Transition berechnet und anschließend $L_1(W^1)$ und $L_1(W^2)$ bestimmt. Aufgrund der Architektur der `Critic`-Klasse werden $L_1(W^1)$ und $L_1(W^2)$ in `critic_loss` kombiniert (s. das folgende Listing).

```
        current_Qs1, current_Qs2 = self.critic(states, actions)
        critic_loss_element_wise1 = (current_Qs1 - target_Qs).pow(2)
        critic_loss_element_wise2 = (current_Qs2 - target_Qs).pow(2)
        critic1_loss_l1 = torch.mean(critic1_loss_element_wise * weights)
        critic2_loss_l1 = torch.mean(critic2_loss_element_wise * weights)
        critic_loss = critic1_loss_l1 + critic2_loss_l1
```

Listing 5.3: Berechnung der Fehler $L_1(W^1)$ und $L_1(W^2)$

Mit dem gleichen Vorgehen werden für den n-step Mini-Batch $L_n(W^1)$ und $L_n(W^2)$ bestimmt, die zu `critic_loss_n` komnbiniert werden. `critic_loss_n` wird mit λ_1 gewichtet und zu `critic_loss` addiert. Abschließend werden die Wichtungen der beiden Q-Netze aktualisiert:

```
        critic_loss += lambda1 * critic_loss_n
        # train critic
        self.critic_optim.zero_grad()
        critic_loss.backward()
        self.critic_optim.step()
```

Listing 5.4: Aktualisierung der Q-Netze

Aktualisierung des Policy-Netzes Der Fehler der Actor-Komponente wird anhand des aktualisierten ersten Q-Netzes $Q^1(s_t, a_t; W^1)$ berechnet (vgl. Formel 4.7). Basierend darauf werden die Wichtungen des Policy-Netzes angepasst:

```
        # Compute actor loss
        actor_loss_element_wise = -self.critic.forward_q1(states, self.
            actor(states))
        actor_loss = torch.mean(actor_loss_element_wise * weights)

        # Optimize the actor
        self.actor_optim.zero_grad()
        actor_loss.backward()
        self.actor_optim.step()
```

Listing 5.5: Berechnung von L_{Actor} und Aktualisierung der Wichtungen des Policy-Netzes

Aktualisierung der Zielnetze Die Wichtungen der Zielnetze werden nach jedem Update der Q-Netze und des Policy-Netzes mithilfe der Methode `target_soft_update()` neu gesetzt.

```
1     def target_soft_update(self):
2         """Soft-update: target = tau*local + (1-tau)*target."""
3         tau = self.tau
4
5         for t_param, l_param in zip(
6                 self.actor_target.parameters(), self.actor.parameters()
7         ):
8             t_param.data.copy_(tau * l_param.data + (1.0 - tau) * t_param.
                  data)
9
10        for t_param, l_param in zip(
11                self.critic_target.parameters(), self.critic.parameters()
12        ):
13            t_param.data.copy_(tau * l_param.data + (1.0 - tau) * t_param.
                  data)
```

Listing 5.6: Aktualisierung der Zielnetze nach der Soft-Update Strategie

Der Trainingsprozess verläuft unabhängig vom lokalen Rechner und kann daher pausiert und nach einiger Zeit fortgesetzt werden. Wenn der Prozess jedoch an der Remote-Maschine abgebrochen wird, geht die in dem 1-step Replay-Puffer verwaltete Priorisierung verloren.

5.4.4 NAO

Die Klasse `NAO` enthält die Methoden zur Steuerung der beiden NAO-Roboter. Bei der Initialisierung der Klasse wird ein Zugriff auf die benötigten Module des NAOqi-Framework über das Objekt `ALProxy` eingerichtet. In der Anwendung wird das roboterzentrische Koordinatensystem (`FRAME_ROBOT`) benutzt. Im Folgenden wird auf einigen Funktionalitäten, die mit der Klasse implementiert werden, eingegangen.

Bewegungssteuerung Die Bewegungssteuerung des Roboters wird mit den Modulen `ALMotion` und `ALRobotPosture` des NAOqi-Framework durchgeführt.
Für die Ausführung einer Aktion wird die *angleInterpolation()*-Methode des ALMotion-Moduls mit den folgenden Parameterwerten aufgerufen:

- `name =[„RShoulderRoll",„RShoulderPitch",„RWristYaw" ]`,
- `angleList` - eine 3-dimensionale Liste mit Float-Werten für jeden Winkel in der `name`-Liste,
- `timeList = [0.15,0.15,0.15]`,
- `isAbsolute = True`.

Skalierung und Ausführung einer Aktion Da in der letzten Schicht des Policy-Netzes die Aktivierungsfunktion *Tanh* verwendet wird, die einen Definitionsbereich [-1; 1] hat, müssen die Aktionen zwischen der maximalen Winkeländerung nach links und der maximalen Winkeländerung nach rechts skaliert werden. Dabei sind die erlaubten Bewegungsbereiche der Gelenke (s. Tabelle 4.1) sowie die maximale Winkeländerung pro Schritt für jedes Gelenk (s. Tabelle 4.2) zu berücksichtigen.
Nach dem sich der Agent für eine Aktion $a_t = [\Delta\varphi^{RShoulderRoll}, \Delta\varphi^{RShoulderPitch}, \Delta\varphi^{RWrist}]$ entschieden hat, wird sie wie folgt bearbeitet:

1. Die Aktion wird mithilfe der Methode `choose_noised_action()` (Klasse `Agent`) verrauscht. Wenn ein Element der verrauschten Aktion $\widetilde{a}_t$ größer als 1 oder kleiner als -1 ist, wird diesem Element den Wert 1 oder -1 zugewiesen, um $\widetilde{a}_t$ im Bereich [-1; 1] zu halten.

```
1 def choose_noised_action(self, cur_state_tensor):
2         action = self.actor(cur_state_tensor).cpu().data.numpy().
              flatten()
3         noise = np.random.normal(0, self.exp_noise, size=act_dim)
4         action = action + noise
5         for i in xrange(0, act_dim):
6             action[i] = self.helper.clip_number(action[i], -1, 1)
7         return action
```

Listing 5.7: Verrauschen einer Aktion des Agenten

2. Anhand der aktuellen Gelenkwinkel wird die größtmögliche Winkeländerung nach rechts Δh^k_{high} und nach links Δh^k_{low} für jedes relevante Gelenk unter der Berücksichtigung der Tabelle 4.1 berechnet.
3. Wenn Δh^k_{high} größer als die erlaubte positive Winkeländerung $\Delta h^k_{high\_per\_step}$ ist (s. Tabelle 4.2), wird $\Delta h^k_{high} = \Delta h^k_{high\_per\_step}$ gesetzt.
4. Wenn Δh^k_{low} kleiner als die erlaubte negative Winkeländerung $\Delta h^k_{low\_per\_step}$ ist, wird $\Delta h^k_{low} = \Delta h^k_{low\_per\_step}$ gesetzt.
5. Abschließend wird die verrauschte Aktion $\widetilde{a}_t$ im Bereich $[\Delta h^k_{low}; \Delta h^k_{high}]$ mithilfe der Methode `scale_action()` skaliert. (s. das folgende Listing)

```
1     def scale_action(self, action, low, high):
2         """Change the range (-1, 1) to (low, high)."""
3         scale_factor, reloc_factor = self.get_scale_and_reloc_factors(
              bounds=[low, high])
4         action = action * scale_factor + reloc_factor
5         return action
6
7     @staticmethod
8     def get_scale_and_reloc_factors(bounds):
9         high = bounds[1]
10        low = bounds[0]
11        scale_factor = (high - low) / 2
```

```
12          reloc_factor = high - scale_factor
13          return scale_factor, reloc_factor
```

Listing 5.8: Skalierung der Aktionen

An dieser Stelle ist anzumerken, dass Aktionen im Bereich [-1; 1] zurück skaliert werden, bevor sie für die Aktualisierung der Netze verwendet werden.

Die Aktion $\widetilde{a}_t$ wird durch den Aufruf der Methode `make_action()` der Klasse `NAO` ausgeführt, die als Parameter ein Array mit Float-Werten für die relativen Winkeländerungen erwartet und die Elemente [0:14] des nächsten Zustandes zurückgibt.

```
1           def make_action(self, action):
2                   """
3                   Parameters
4                   ----------
5                   action: float array of relative angles
6                   Returns
7                   -------
8                   elements [0:14] of the next state as np.array
9                   """
10                  names = ["RShoulderRoll", "RShoulderPitch", "RWristYaw"]
11                  angle_list = list()
12                  time_list = list()
13                  for i in xrange(0, len(action)):
14                      angle_list.append(action[i] * almath.TO_RAD)
15                      time_list.append(0.15)
16                  is_absolute = False
17                  self.motion_active.angleInterpolation(names, angle_list,
                        time_list, is_absolute)
18                  return self.get_state()
```

Listing 5.9: Skalierung der Aktionen

Ballverfolgung Zur Erkennung des Balls und zur Ermittlung der Ballposition wurden folgende Möglichkeiten in Betracht gezogen:

- das OpenCV-Framework für die Bildverarbeitung und maschinelles Sehen,
- das `ALTracker`-Modul des NAOqi-Framework.

In dieser Arbeit wird das `ALTracker`-Modul dem OpenCV-Framework vorgezogen, da es direkt die gewünschten Daten (die 3D-Position des Balls im Koordinatensystem `FRAME_ROBOT`) zurückliefert.
In der Praxis hat sich jedoch herausgestellt, dass die Ermittlung der Ballposition mithilfe des Moduls `ALTracker` einige Probleme mit sich bringt:

1. Ein anderes rotes Objekt im Blickwinkel des Roboters kann als Spielball wahrgenommen werden.

2. Wenn der Roboter den Ball mit dem Kopf verfolgt und der Ball sich schnell bewegt, verliert der Roboter den Ball aus dem Blickfeld.

Das Problem 1 kann reduziert werden, wenn das rote Objekt, den der Roboter als Ball wahrnimmt, sich am Anfang der Episode nicht bewegt. Wenn der Roboter die Startposition annimmt (s. Abb. 6.12), muss der Ball sich bewegen. Falls die vom Beobachter ermittelte Position des Balls sich jedoch nicht ändert, erkennt der Roboter als Ball ein anderes Objekt (s. den Thread `track_if_ball_moves_thread()` der Klasse `NAO`).
Das Problem 2 wird dadurch gelöst, dass der Ball von einem zweiten NAO-Roboter (im Weiteren als Beobachter referenziert) verfolgt wird. Dabei steht der Beobachter in einer Entfernung, die genügt, um den Ball im Auge zu behalten ohne den Kopf dafür bewegen zu müssen. Das Modul `ALTracker` des Beobachters wird mit dem Modus „Head“ initialisiert, die Gelenkmotoren des Roboters werden aber ausgeschaltet, sodass sich der Kopf sich während des Trackings nicht bewegt.

Der Beobachter erfährt die Position des Balls $ball_{observer} = [x_{observer}, y_{observer}, z_{observer}]$ im Koordinatensystem `FRAME_ROBOT` durch den Aufruf der Methode `getTargetPosition()` des Moduls `ALTracker`. Die Ballposition $ball_{observer}$ wird in das Koordinatensystem `FRAME_ROBOT` des Lerners umgerechnet, indem die x- und y-Achsen gespiegelt werden sowie der Ursprung um den Abstand zwischen den Robotern auf der `x`-Achse verschoben wird. Dabei wird darauf geachtet, dass die Roboter parallel zueinanderstehen.

Sicherheit des Roboters Während der Ausführung einer Episode werden u. a. die Motoren der Gelenke `RHipPitch` und `LHipPitch` angeschaltet, damit der Roboter nicht umfällt.
Nach jeder Episode nimmt der Roboter die vordefinierte sichere Pose „Crouch“ an und die Motoren aller Gelenke werden ausgeschaltet. Es wurde festgestellt, dass insbesondere das Gelenk `RShoulderPitch` sich schnell überhitzt. Aus diesem Grund wird nach jeder Episode die Methode `sleep()` aufgerufen, welche die Temperaturen der Robotergelenke überprüft und wartet, bis sie sich abkühlen (s. Listing 5.10).

```
1   def sleep(self):
2       sleep = True
3       while sleep:
4           count = 0
5           for key in ALMEMORY_KEY_NAMES_TEMPERATURE:
6               value = self.nao.memory_nao_active.getData(key)
7               if value > 66:
8                   count += 1
9           if count > 0:
10              time.sleep(60)
11          else:
12              sleep = False
```

Listing 5.10: Vermeidung der Überhitzung der Robotergelenke

Das NAOqi-Framework führt eine automatische Überprüfung der Temperaturen der Robotergelenke durch. Falls ein Gelenk überhitzt ist, wird die Stiffness des Gelenkes reduziert. Dieses Verhalten soll verhindert werden, da bei einer niedrigen Stiffness die gewünschte Aktion des Roboters unpräzise ausgeführt wird. In der Dokumentation des NAOqi-Framework Version 2.14 konnten keine Angaben dazu gefunden werden, ab welcher Temperatur die automatische Reduzierung der Stiffness stattfindet. In der Dokumentation des NAOqi-Framework Version 1.4 wird jedoch erwähnt, dass die Temperaturgrenze bei 75 °C liegt [SoftBank Robotics, 2013]. Da während einer Episode die Temperatur der verwendeten Gelenke um maximal 4-5 °C ansteigt, würde die Überprüfung, dass alle Gelenktemperaturen am Anfang der Episode kleiner als 70 °C sind, ausreichend sein. Bei der Durchführung der Trainingsversuche wurde jedoch festgestellt, dass die Bewegungen des Roboters bereits ab $>$ 71-72 °C anders aussehen können als bei niedrigen Temperaturen. Aus diesem Grund wurde in der `sleep()`- Methode die Temperaturgrenze 66 °C festgelegt.

5.5 Zusammenfassung

Im Rahmen dieses Kapitels wurde die Implementierung der im Abschnitt 4.4 beschriebenen Lösungsansatzes für das Spiel Ball-in-a-Cup betrachtet. Es wurde die Umsetzung der Actor- und Critic-Komponenten mithilfe von PyTorch-Framework beschrieben und die für die neuronalen Netze verwendeten Hyperparameter angegeben. Des Weiteren wurde auf die Skalierung der Aktionen, Ballverfolgung und Sicherheit des Roboters eingegangen.

6 Evaluation

6.1 Vorstellung der Ergebnisse

Im Folgenden werden die Ergebnisse der Experimente in virtuellen Umgebungen und mit dem realen NAO-Roboter vorgestellt. Bei der Auswertung der Lernversuche steht die Reproduzierbarkeit und Analyse der Ergebnisse im Vordergrund. Ein Deep RL-Algorithmus kann unterschiedliche Leistung erzielen, beeinflusst u. a. von folgenden Faktoren (vgl. [Henderson et al., 2017], S. 2):

- stochastische Umgebung,
- Random-Seeds (Initialisierung der Netzwichtungen, Generierung von Zufallszahlen für das Rauschen, PER),
- Kombination der Hyperparameter,
- Implementierung des Algorithmus.

Die Abbildung 6.1 verdeutlicht die Varianz der Leistung des umgesetzten Algorithmus. Das Programm wird dreimal mit den gleichen Hyperparametern aber mit unterschiedlichen Random-Seeds in einer deterministischen Umgebung gestartet. Die Kurven zeigen den Gesamtreward des Agenten über 200 Episoden. Um die Ergebnisse übersichtlicher zu präsentieren, werden die Kurven in einer geglätteten Form dargestellt (gleitender Durchschnitt mit Fenster 20).

In wissenschaftlichen Publikationen wird das Problem der Varianz der Ergebnisse aufgrund der unterschiedlichen Random-Seeds dadurch minimiert, dass die Experimente mehrmals mit einer großen Anzahl von verschiedenen Random-Seeds durchgeführt werden und die Kurve der Gesamtbelohnung als geglätteter Mittelwert über alle Versuche mit Standardabweichungsgrenzen dargestellt wird (vgl. [Lynnerup et al., 2019], S. 3).

Da die Durchführung der Experimente mit dem realen Roboter zeitintensiv ist, kann eine solche Evaluation im Lernszenario mit dem Roboter nicht vorgenommen werden. Für die virtuellen Umgebung werden jedoch alle Versuche mit den drei in der Tabelle 6.1 notierten Random-Seeds durchgeführt und als geglätteter Mittelwert über die drei Random-Seeds dargestellt. Die Abbildung 6.2 zeigt eine solche Kurvenrepräsentation und stellt die gleichen Daten wie die Abbildung 6.1 dar.

Da bei dem Spiel Ball-in-a-Cup keine Simulation vorliegt und das Gelenk `RShoulderPitch` des NAO-Roboters sich schnell überhitzt, wodurch das Training sehr zeitaufwendig wird,

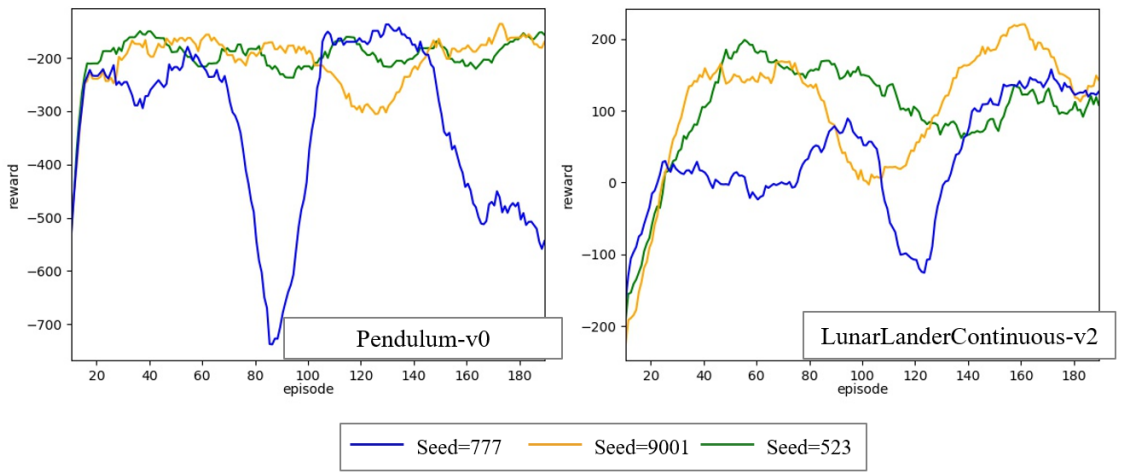


Abbildung 6.1: Varianz der Ergebnisse in einer deterministischen Umgebung
Quelle: Eigene Darstellung

werden die Experimente auf 200 Episoden eingeschränkt. Die Analyse der Hyperparameter in virtuellen Umgebung befasst sich auch mit diesem Abschnitt des Trainings. Des Weiteren ist zu erwähnen, dass die Episodenlänge nur in „Pendulum-v0“ konstant ist. In „LunarLanderContinuous-v2“ und dem Ball-in-a-Cup Spiel variiert die Länge von Episode zur Episode, was zu einer unterschiedlichen Anzahl von Lernschritten zwischen den Episoden führt.

Seed 1	Seed 2	Seed 3
9001	523	777

Tabelle 6.1: Random-Seeds für Experimente in virtuellen Umgebungen
Quelle: Eigene Darstellung

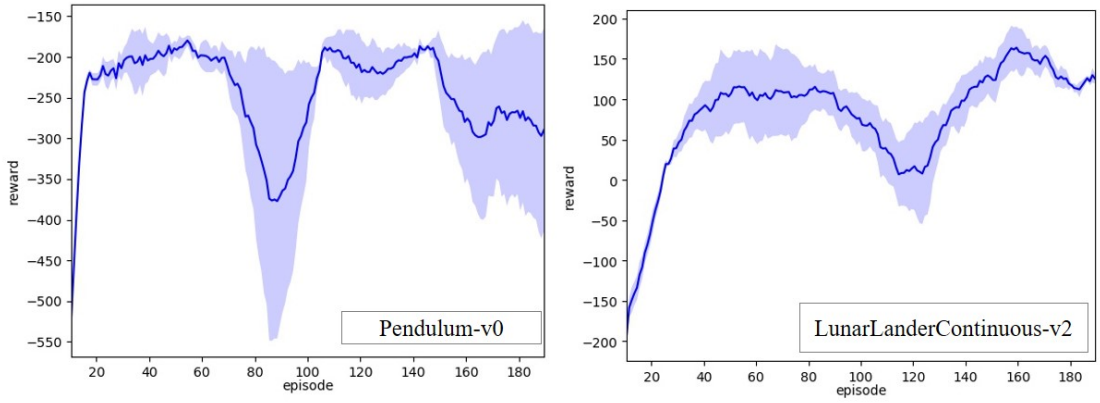


Abbildung 6.2: Darstellung der Lernkurve über drei Random-Seeds
Quelle: Eigene Darstellung

6.1.1 Virtuelle Umgebungen

Training ohne Demonstrationen und mit Demonstrationen Die Abbildung 6.3 verdeutlicht die Wirkung der Demonstrationen auf den Trainingsprozess.

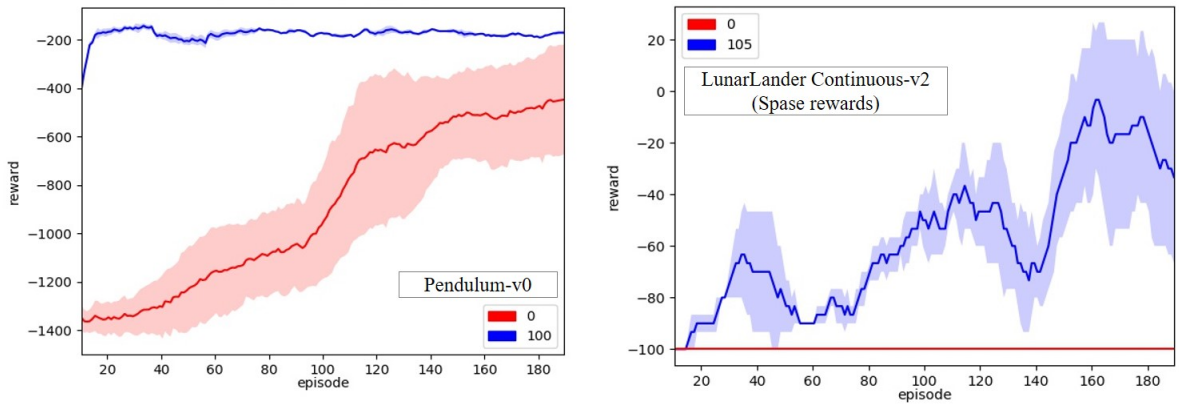


Abbildung 6.3: Wirkung der Demonstrationen auf das Training.
Hyperparameter für „Pendulum-v0“: A.4.4,
Hyperparameter für „LunarLander Continuous-v2“: A.4.3
Quelle: Eigene Darstellung

Der Lernprogress verläuft mit den Demonstrationen in den beiden Umgebungen schneller als ohne Demonstrationen. In der Umgebung „LunarLander Continuous-v2“ mit einer Sparse-Rewardfunktion erlebt der Agent während der 200 Trainingsepisoden keine positive Erfahrung und kann aus diesem Grund seine Policy nicht verbessern. Im Gegensatz dazu ist der Lernprogress beim Training mit den Demonstrationen bereits ab den ersten Episoden sichtbar.

Die Anzahl von Demonstrationen Die Aufnahme von Demonstrationen beansprucht Zeit des Experten. Die Abbildung 6.4 verdeutlicht, welche Auswirkung die Anzahl von Demonstrationen auf das Training hat. Für die Experimente wurden die Lernschritte vor der Interaktion mit der Umgebung an die Größe der Demonstrationen angepasst (ca. 70 Lernschritte pro Demonstration).
Wenn dem Agenten wenig Demonstrationen bereitstehen, weist die Kurve des Gesamtrewards in den beiden Umgebungen eine merkliche Varianz auf. In der Umgebung „Pendulum-v0“ erreicht der Agent trotz der wenigen Vorführungen nach ca. 150 Trainingsepisoden eine hohe Leistung. In der Umgebung „LunarLander Continuous-v2“ mit der Sparse-Rewardfunktion reichen jedoch 3 Demonstrationen nicht aus, um in dem untersuchten Trainingsabschnitt eine gute Policy zu erlernen.
In den beiden Umgebungen ist die Leistung des Agenten hoch und stabil bei der Verwendung von 50 Demonstrationen. Bei 100 Demonstrationen verschlechtert sich das Lernen. Das ist besonders in der „LunarLander Continuous-v2“-Umgebung deutlich zu erkennen.

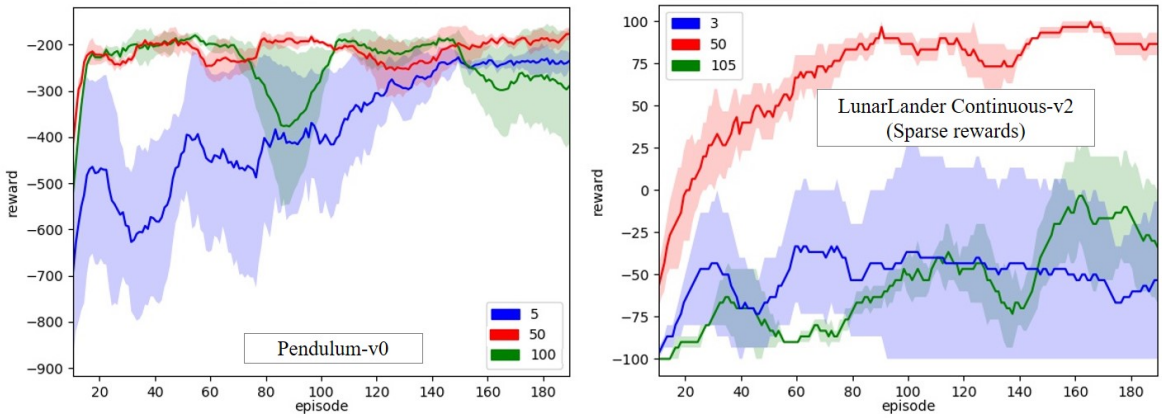


Abbildung 6.4: Wirkung der Anzahl von Demonstrationen auf das Training.
Hyperparameter für „Pendulum-v0: A.4.1,
Hyperparameter für „LunarLander Continuous-v2“: A.4.3
Quelle: Eigene Darstellung

Wirkung der Konstante ϵ_d Die Konstante ϵ_d wird zur Priorität einer Transition, die vom Experten vorgezeigt wurde, addiert. Die folgende Abbildung stellt den Gesamtreward des Agenten im Training bei unterschiedlichen Werten von ϵ_d: 0.0001, 0.01 und 1 dar. Wie hier deutlich wird, beginnt der Agent in beiden Umgebungen das Training mit einer höheren Leistung bei größeren Werten von ϵ_d. In „Pendulum-v0“ fällt der Unterschied zwischen den Lernkurven besonders auf. Der Agent erlernt bereits nach 20 Episoden eine gute Strategie mit $\epsilon_d = 1$ in allen drei Random-Seeds, jedoch steigt die Varianz des Gesamtrewards nach der Episode 80 an. Im Vergleich dazu braucht der Agent mit $\epsilon_d = 0.01$ länger, bis er anfängt, einen hohen Reward zu erzielen, die Leistung bleibt aber in den weiteren Episoden stabil.

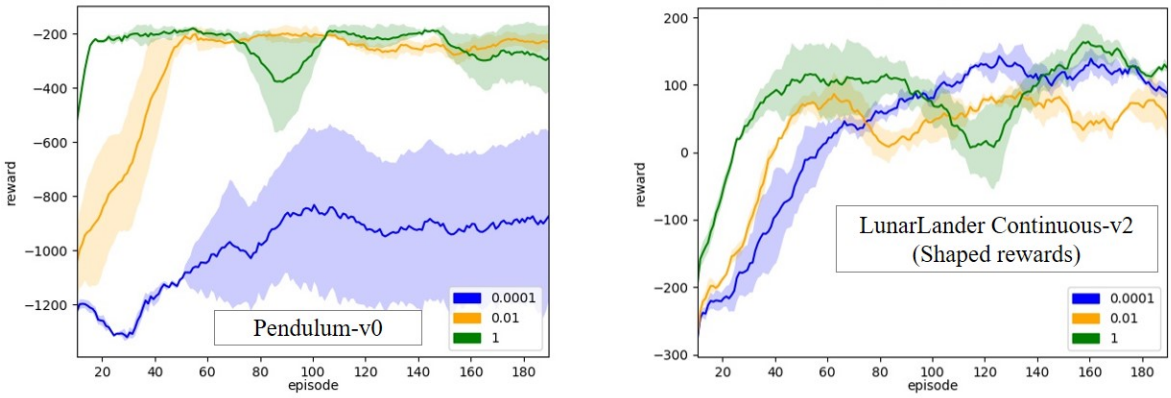


Abbildung 6.5: Wirkung der Konstante ϵ_d auf das Training.
Hyperparameter für die beiden Umgebungen: A.4.1
Quelle: Eigene Darstellung

N-step Das Liniendiagramm auf der Abbildung 6.6 sowie der Boxplot 6.7 zeigen den Gesamtreward des Agenten bei $n = \{5, 10, 20, 30\}$. Für die Erstellung des Boxplots wurden die Daten für jede Konfiguration von n als Mittelwerte des Gesamtrewards für jede einzelne Episode über drei Random-Seeds repräsentiert.
Aus dem Liniendiagram ist erkennbar, dass bei $n = 5$ eine große Varianz über die drei Random-Seeds sowie ein unstabiles Lernen in den beiden Umgebungen besteht. Der Boxplot verdeutlicht, dass die Erhöhung von n bis zum Wert $n = 20$ die Leistung des Agenten verbessert (vgl. Median auf dem Boxplot). Allerdings wenn $n = 30$ gewählt ist, wächst die Varianz des Gesamtrewards und die Leistung verschlechtert sich leicht.

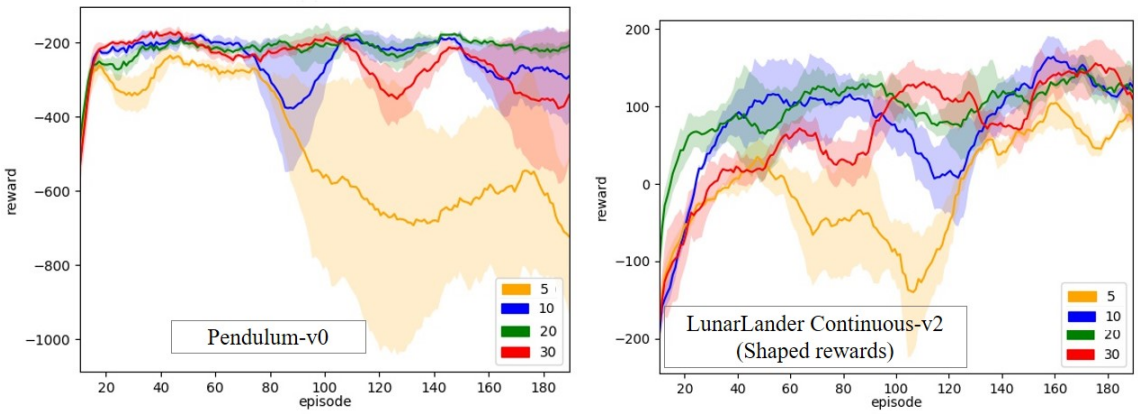


Abbildung 6.6: Wirkung von n-step auf das Training (Liniendiagramm).
Hyperparameter für die beiden Umgebungen: A.4.1
Quelle: Eigene Darstellung

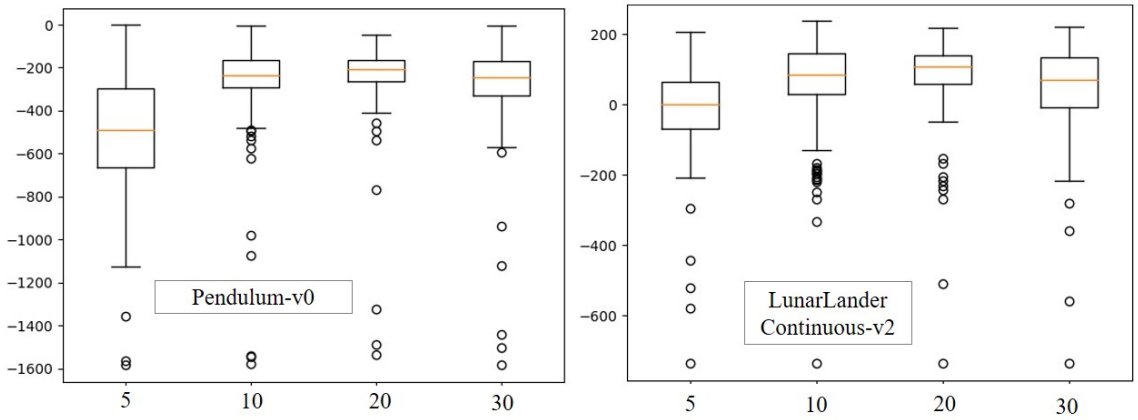


Abbildung 6.7: Wirkung von n-step auf das Training (Boxplot).
Hyperparameter für die beiden Umgebungen: A.4.1
Quelle: Eigene Darstellung

Die Abbildung 6.8 präsentiert die Wirkung von n-step auf das Training in der „LunarLander Continuous-v2“-Umgebung mit einer Sparse-Rewardfunktion. Im Gegensatz zu den Einstellungen mit der Shaped-Rewardfunktion, verschlechtert sich die Leistung des

Agenten mit der Erhöhung von n nicht. Diese Ergebnisse stimmen mit den in [Hernandez-Garcia & Sutton, 2019] berichteten Beobachtungen überein.
Bezüglich der Policy des Agenten ist anzumerken, dass bereits mit 8 Demonstrationen der Agent nach 100 Trainingsepisoden eine optimale Policy erlernt hat (Random-Seed 9001), mit der er in der Lage war, eine Erfolgsrate von 100 % bei 100 Testepisoden zu erreichen.

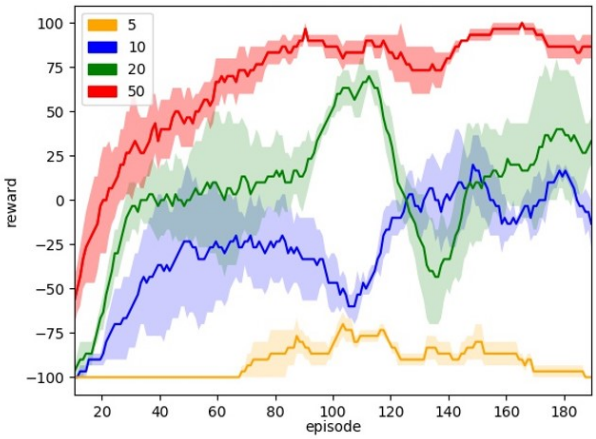


Abbildung 6.8: Wirkung von n-step auf das Training in der „LunarLander Continuous-v2“-Umgebung mit einer Sparse-Rewardfunktion.
Hyperparameter: A.4.3
Quelle: Eigene Darstellung

Netzgrößen In [Islam et al., 2017] und [Henderson et al., 2017] wird empirisch gezeigt, dass die Netzarchitektur eine signifikante Auswirkung auf die Leistung des DDPG-Algorithmus hat. Die Abbildung stellt die Lernkurven für drei verschiedenen Netzgrößen dar: (64, 64), (128, 128) und (256, 256) in der „LunarLanderContinuous-v2“-Umgebung sowie (32, 32), (64, 64) und (128, 128) in der „Pendulum-v0“-Umgebung. Es werden in jedem Versuch die gleichen Netzgrößen für die Actor- und die Critic-Komponente verwendet. Die Grafik für die „Pendulum-v0“-Umgebung zeigt, dass je mehr Neuronen das Netz in den versteckten Schichten enthält, desto größer ist die Anfangsleistung des Agenten. Bei der Netzgröße (32, 32) verläuft das Lernen verglichen mit den beiden anderen Netzgrößen am stabilsten.
Auch in der „LunarLanderContinuous-v2“-Umgebung weist das kleinere Netz (64, 64) einen stabileren Lernprozess als die Netze der Größe (256, 256) und (128, 128) auf. Die Schwankungen in den Kurven bei der Verwendung von größeren Netzen könnten dadurch erklärt werden, dass große Netze ohne starke Regularisierungsmaßnahmen die Trainingsdaten auswendig lernen, aber mit den neuen Daten nicht gut umgehen können.

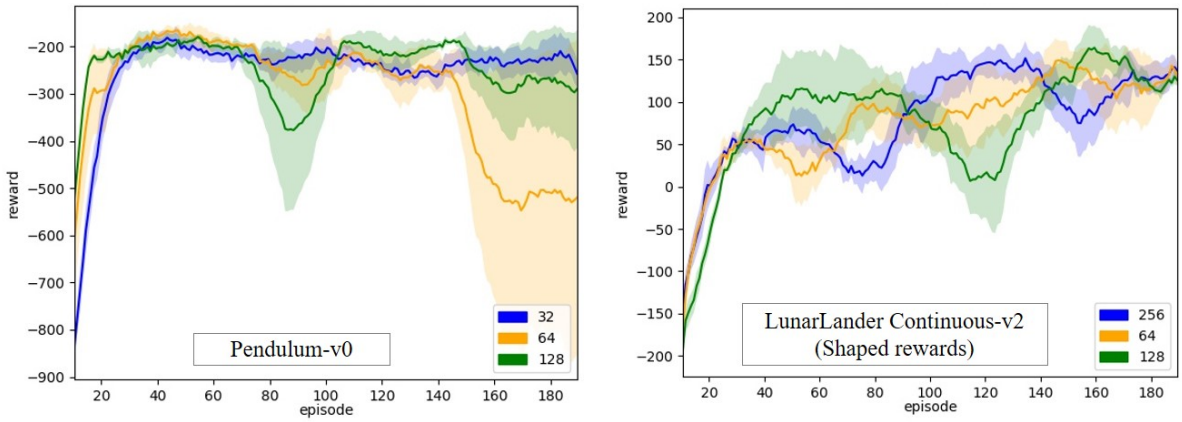


Abbildung 6.9: Wirkung der Netzgröße auf das Training,
Hyperparameter für die beiden Umgebungen: A.4.1
Quelle: Eigene Darstellung

Trainingsdauer vor der Interaktion mit der Umgebung Die Abbildung 6.10 zeigt die Kurve des Gesamtrewards abhängig von der Anzahl der Lernschritte *pretrain_steps*, die ausgeführt werden, bevor der Agent beginnt, seine eigene Erfahrung zu sammeln.
Es ist erkennbar, dass wenn die Anzahl der Lernschritte groß gewählt ist, z. B. *pretrain_steps* = 8000 Lernschritte mit 100 Demonstrationen, erzielt der Agent eine gute Leistung am Anfang des Lernens, seine Strategie verschlechtert sich aber nach ca. 90 Episoden deutlich. Wenn keine Lernschritte vor der Interaktion mit der Umgebung durchgeführt werden, beginnt der Agent mit einem kleineren Gesamtreward, zeigt aber bereits nach den ersten 20-40 Episoden eine hohe Leistung, die auf einigen Abschnitten des Trainings sogar größer ist, als bei *pretrain_steps* = 2000 oder *pretrain_steps* = 4000.

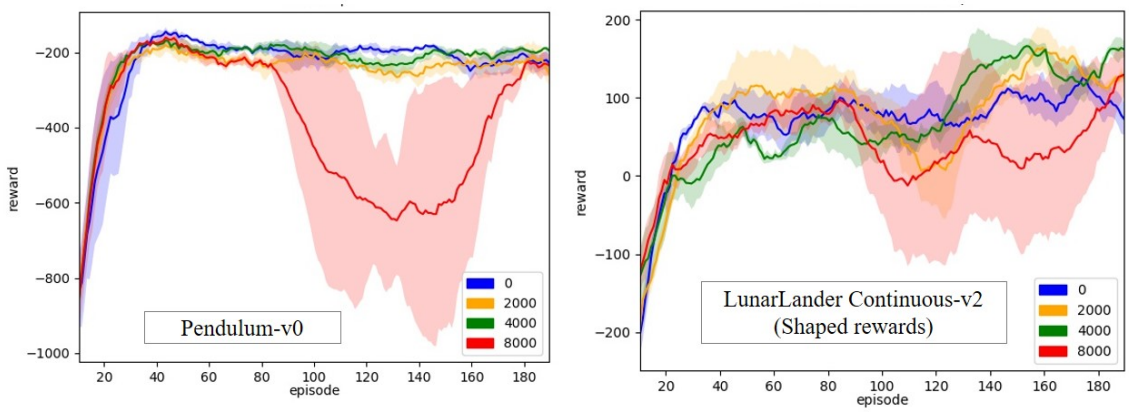


Abbildung 6.10: Wirkung der Trainingsdauer vor der Interaktion mit der Umgebung.
Hyperparameter für „Pendulum-v0“: A.4.2,
Hyperparameter für „LunarLander Continuous-v2“: A.4.1
Quelle: Eigene Darstellung

Die Abbildung 6.11 zeigt die Anzahl der Transitionen, die in jedem Lernschritt für die

Aktualisierung der neuronalen Netze verwendet werden. Die Grafik präsentiert die Kurven für den Trainingsprozess in der „Pendulum-v0“-Umgebung mit $pretrain\_step = 0$ (blau) und $pretrain\_steps = 8000$ (rot). Mit der gestrichelten Linie ist der Zeitpunkt markiert, in dem der Agent 100 Episoden ausgeführt hat. Wie erwartet, werden im Laufe der Zeit immer weniger Demonstrationen für die Aktualisierung der Netze verwendet.
Bei $pretrain\_step = 0$ ist eine leichte Steigung über die Trainingsschritte 250-500 (ca. 25. bis 50. Episode) zu beobachten. Im gleichen Abschnitt des Trainings steigt die Leistung des Agenten (vgl. Abb. 6.10).
Bei $pretrain\_steps = 8000$ verschlechtert sich die Leistung des Agenten nach ca. 90 Episoden erheblich. Dies bewirkt, dass nach ca. 110 Episode weniger Demonstrationen verglichen mit dem gleichen Abschnitt des Lernens bei $pretrain\_step = 0$ aus den Replay-Puffern entnommen werden.

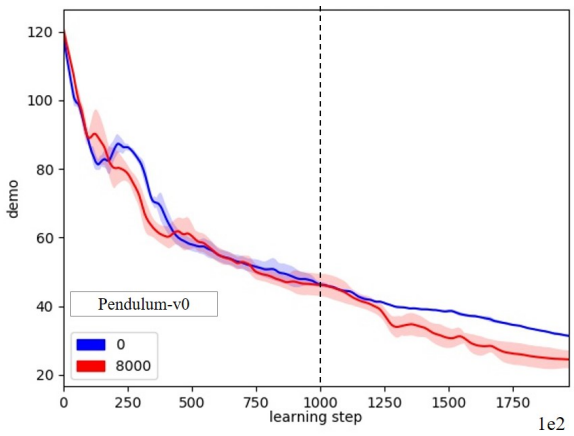


Abbildung 6.11: Anzahl der Demonstrationen in dem Mini-Batch über 200 Episoden
Quelle: Eigene Darstellung

6.1.2 Spiel Ball-in-a-Cup mit dem NAO-Roboter

Durchführung des Trainings Zu Beginn der Episode nimmt der Lerner die vordefinierte Position ein (s. Abb. 6.12). Der Beobachter sucht den roten Ball. Wenn der Ball erkannt ist und sich bewegt, darf der Lerner die Episode anfangen.

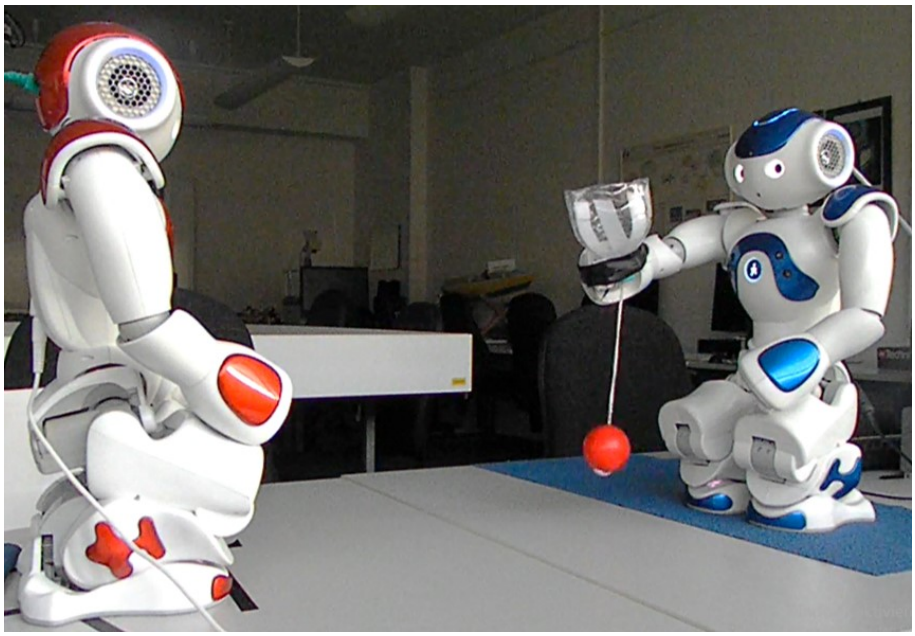


Abbildung 6.12: Die Pose des Roboters zu Beginn einer Episode
Quelle: Eigene Darstellung

Bei der Durchführung des Trainings sollten folgende Anpassungen an den Versuchsaufbau vorgenommen werden:

1. Es musste eine bestimmte Beleuchtung eingestellt werden, die die Ermittlung der Ballposition verbessert.
2. Die Position der Füße des Lerners sollte fixiert werden. Wenn der Roboter eine weite und schnelle Bewegungen mit der Hand ausführt, ändert sich die Position und die Orientierung seiner Füße. In diesem Fall stimmt die Orientierung des Koordinatensystems des Lerners nicht mehr mit der Orientierung des Koordinatensystems des Beobachters überein.

Des Weiteren ist zu beachten, dass wenn der Roboter sich für eine falsche Bewegung entscheidet, der Ball zwischen seiner Hand und dem Becher stecken bleibt (s. Abb. 6.13). Falls der Roboter nicht mit einer anderen Bewegung bewirkt, dass der Ball rausspringt, bleibt der Ball bis zum Ende der Episode stecken. Eine solche Bewegung wurde im Rahmen der Demonstrationen allerdings nicht vorgezeigt.
Wenn der Ball in der Position, die in der Abbildung 6.13a dargestellt ist, gelangt, bekommt der Roboter den Reward „0“. Wenn der Ball sich in einer ähnlichen Position wie in der Abbildung 6.13c befindet, erhält der Roboter gelegentlich den Reward „+5“, da der Ball sich nah an der Zielposition befindet. Die Ballposition in der Abbildung 6.13b kann als Gewinn erkannt werden und das würde dazu führen, dass der Roboter ein falsches Verhalten als Zielverhalten erlernt. Aus diesem Grund musste im Laufe des Trainings die Vergabe des positiven Rewards „+50“ durch den Entwickler bestätigt werden.

a. b. c.

Abbildung 6.13: Schwierigkeiten beim Spiel: Der Ball bleibt zwischen dem Becher und der Hand des Roboters stecken
Quelle: Eigene Darstellung

Ergebnisse Im finalen Trainingslauf wurden 200 Episoden ausgeführt. Das entsprach 89 Minuten der Interaktion mit der Umgebung. Die Durchführung des Versuchs dauerte jedoch länger, weil zum einen nach jeder zehnten Episode drei Testepisoden ausgeführt wurden und zum anderen das Gelenk `RShoulderPitch` durchschnittlich zwei Minuten zum Abkühlen zwischen den Episoden benötigte. Die durchschnittliche Episodendauer betrug 24.09 Sekunden, wobei die kürzeste Episode 5.5 Sekunden und die längste Episode 31.7 Sekunden dauerte. Der Roboter verfügte über 100 Demonstrationen.
Die Abbildung 6.14 stellt den Verlauf des erlangtem Gesamtrewards während des Lernens dar.

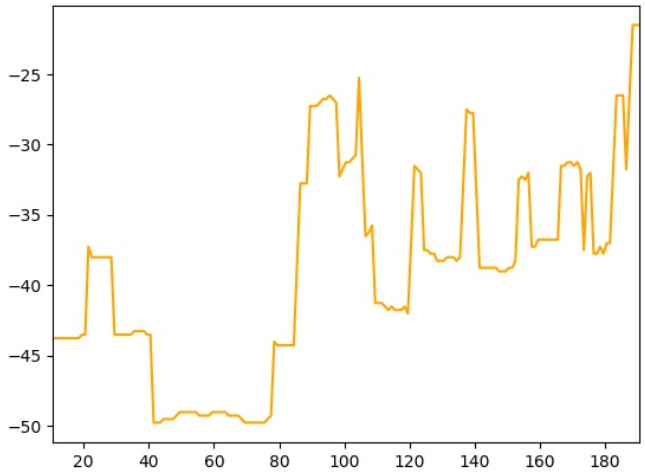


Abbildung 6.14: Gesamtreward im Spiel Ball-in-a-Cup,
Hyperparameter: A.4.5
Quelle: Eigene Darstellung

Den ersten Gewinn konnte der Roboter bereits nach 19 Episoden erzielen. Die Erfolgsrate der erlernten Policy beträgt 23% (getestet über 13 Episoden mit einer unverrauschten Policy). Die Policy des Roboters sieht der Policy des Demonstrators ähnlich, mit dem Unterschied, dass der Roboter eine weite Bewegung in dem `RShoulderPitch`-Gelenk ausübt.

6.2 Zusammenfassung

In diesem Kapitel wurde eine Reihe von Experimenten in den virtuellen Umgebungen und der finale Testlauf mit dem NAO-Roboter beschrieben. Es wurde gezeigt, dass der Algorithmus je nach der Problemstellung, der Anzahl von Demonstrationen, der Konfiguration der Hyperparameter und der Art der Rewardfunktion unterschiedliche Leistung erzielen kann.

7 Fazit

Im Rahmen dieser Arbeit wurde eine Softwareanwendung entwickelt, die es dem NAO-Roboter ermöglicht, die Bewegung seiner rechten Hand bei dem Spiel Ball-in-a-Cup selbstständig zu steuern.
Das Lernproblem wurde als ein MDP mit einem kontinuierlichen Zustandsraum und einem kontinuierlichen Aktionsraum formalisiert. Der umgesetzte Lernalgorithmus verwendet Demonstrationen eines Experten zur Beschleunigung des Lernprozesses und baut sich auf den Algorithmen Deep Deterministic Policy Gradient from Demonstration (DDPGfD) und Twin Delayed Policy Gradient (TD3) auf.
Die Implementierung des Lernalgorithmus erfolgte in der Programmiersprache Python unter Verwendung des NAOqi-Framework zur Steuerung des NAO-Roboters und des PyTorch-Framework zur Erstellung und Aktualisierung der neuronalen Netze. Die komplexen Berechnungen fanden auf der Grafikkarte NVIDIA Titan RTX statt.
Der umgesetzte Lösungsansatz wurde in dem Spiel Ball-in-a-Cup und in den virtuellen Umgebungen „Pendulum-v0“ und „LunarLanderContinuous-v2“ evaluiert. Bei der Evaluation standen die Performanz des Agenten in den ersten 200 Episoden und die Reproduzierbarkeit der Ergebnisse im Vordergrund.
Im Spiel Ball-in-a-Cup zeigte der NAO-Roboter im finalen Testlauf die Erfolgsrate von 23 % nach 200 Episoden, die 80 Min. Trainingszeit entsprechen.

7.1 Diskussion

Im Folgenden werden die Ergebnisse der Arbeit in Bezug auf die im Abschnitt 1.2 formulierte Forschungsfrage und die in den Abschnitten 1.4 und 4.1 definierten Anforderungen diskutiert.

Der umgesetzte Ansatz kann als eine Kombination von RL und IL betrachtet werden, jedoch wird aus IL nur die Idee der Demonstrationen übernommen. Der Roboter lernt mithilfe von RL anhand der Daten, die sowohl die Erfahrung des Experten (Demonstrationen) als auch seine eigene Erfahrung beinhalten. Obwohl in durchgeführten Experimenten die erlernte Policy des Agenten ähnlich der Policy des Experten aussieht, verfolgt der Agent beim Lernen nicht die Absicht, den Experten zu imitieren. Einerseits bietet dieser Ansatz dem Roboter die Möglichkeit, eine eigene Strategie zu entwickeln, die von der Strategie

des Experten unabhängig ist. Andererseits muss ein solcher Ansatz in den Anwendungsszenarien, in denen der Roboter sich selbst oder der Umgebung Schaden zufügen kann, mit Bedacht eingesetzt werden.
Im Lernprozess wird eine priorisierte Entnahme der Transitionen aus dem Replay-Puffer verwendet, wobei die Transitionen des Experten einen Bonus (die Konstante ϵ_d) zu ihrer Priorität enthalten. Des Weiteren werden die Demonstrationen im Laufe der Zeit nicht überschrieben. Da der Agent beim Lernen nicht das Ziel verfolgt, ähnliche Aktionen wie der Experte zu wählen, ergibt sich die folgende Frage: Welchen Nutzen bringen die Demonstrationen für den Lernprozess, wenn der Agent bereits eine Policy erlernt hat, mit der er einen größeren Gesamtreward erzielen kann als der Experte? Es ist ein Ansatz denkbar, in dem die Demonstrationen mit dem eigenen Wissen des Agenten überschrieben werden, falls er eine bessere Leistung als der Demonstrator zeigt. Im Spiel Ball-in-a-Cup könnte die Überschreibung der Demonstrationen mit der positiven Erfahrung des Agenten einen zusätzlichen Vorteil bringen. Da die Demonstrationen durch das kinästhetische Führen erfolgen und bzgl. der Dynamik inakkurat sein können, hat eine erfolgreiche Episode, die der Agent selbst erlebt hat, einen größeren Wert für den Lernprozess als die Demonstrationen.
Die Ergebnisse zeigen, dass der Lernprozess bei Deep RL mithilfe von Demonstrationen beschleunigt werden kann. Den ersten Gewinn erlangte der Roboter bereits nach der 19. Episode.
Die Leistung der nach 200 Episoden erlernten Policy ist als akzeptabel zu bewerten, wenn berücksichtigt wird, dass für den NAO-Roboter keine Testversuche in der Simulation erfolgten und dass die Sensordaten des Roboters ungenau sind. Die Einstellung der Hyperparameter sowie die Implementierung des Lernalgorithmus ohne Simulation hat sich jedoch als zeitaufwendig erwiesen und kann rückblickend angesichts des aktuellen Standes der Forschung als ein unpraktisches Szenario beurteilt werden.
Die Formalisierung des Spiels Ball-in-a-Cup kann überdacht werden. Bei der Zustandsbeschreibung sind die Winkelgeschwindigkeiten der Robotergelenke und die Geschwindigkeit des Balls, das Feedback der Gelenkmomente und eine genauere Lokalisierung des Bechers wünschenswert. Als Aktionen sollten die Winkelgeschwindigkeiten verwendet werden. Aufgrund eines eingeschränkten Zugriffs auf die Hardware des Roboters sind die erwünschten roboterbezogenen Informationen nicht direkt über das NAOqi-Framework verfügbar. In der Literatur lassen sich jedoch einige Ansätze zur Ermittlung dieser Angaben finden [Kim et al., 2016], [Hawley et al., 2018].
Weiterhin ist die Robustheit des Lernprozesses gegenüber der Sensorfehler zu diskutieren. Die neuronalen Netze können bei einer sorgfältigen Auswahl der Hyperparameter und Regularisierungsmaßnahmen eine hohe Generalisierungsfähigkeit aufweisen. Die Unterschiede und Zusammenhänge zwischen den Zuständen, Aktionen und Rewards sollen also trotz der Sensorfehler erkannt werden. Wenn die Netze sich jedoch zu sehr an die Daten anpassen, wird das Rauschen der Sensoren mitgelernt. Um das zu vermeiden, sollte zusätzlich zu der umgesetzten L2-Regularisierung die Technik Early-Stopping implementiert werden sowie andere Regularisierungsmaßnahmen evaluiert werden (z. B. Dropout statt L^2). Alternativ

könnte die Formalisierung des Spiels als POMPD die Robustheit des Lernprozesses gegenüber der Sensorfehler gewährleisten. Das würde jedoch zu einer höheren Komplexität des Lernalgorithmus führen.
Für die Analyse der Auswirkung der einzelnen Hyperparameter auf den Lernprozess wurden die virtuellen Umgebungen in einem deterministischen Modus gestartet. Für die Lernversuche wurden drei festgelegte Random-Seeds verwendet, sodass die Ergebnisse reproduziert werden können.
Der Random-Seed bei dem finalen Lernversuch mit dem NAO-Roboter wurde ebenfalls dokumentiert. Da die Umgebung stochastisch ist, kann allerdings hier die genaue Wiederholung der Ergebnisse nicht garantiert werden.
Bezüglich der Auswertung der Ergebnisse ist außerdem anzumerken, dass die Analyse des Lernprozesses auf einen weiteren Horizont als die in den Experimenten aufgeführten 200 Episoden notwendig ist.

7.2 Ausblick

Die im Rahmen dieser Arbeit implementierte Softwareanwendung sowie der zugrunde liegende Lernalgorithmus können verbessert und weiterentwickelt werden. Die wichtigste Voraussetzung dafür ist **die Erstellung einer Simulation** für das Spiel Ball-in-a-Cup mit dem NAO-Roboter. Die Simulation kann mithilfe des Webots-Framework [Webots, 2019] realisiert werden. Zur Ermittlung der Grenzen des Lernalgorithmus können mehrere Schwierigkeitsgrade der Aufgabe spezifiziert werden. Sie können sich je nach der Beweglichkeit und der Länge der Schnur, an dem der Ball befestigt ist, oder nach der Größe des Bechers und des Balls unterscheiden.
Die **Formalisierung des Spiels** soll auf die im Abschnitt 7.1 vorgestellten Vorschläge angepasst werden. Auch der **Versuchsaufbau** soll überarbeitet werden, sodass das Training ohne die Anwesenheit des Entwicklers möglich ist.
Die in der Arbeit verwendete Technik zur **Erfassung von Demonstrationen** kann verbessert werden. Um den Zustand und die Aktion des Agenten genauer zu ermitteln, kann die Policy des Experten ähnlich wie in [Vecerík et al., 2017] vorgezeigt werden: Der Demonstrator bewegt den Arm des anderen Roboter während der Lerner die Bewegung unmittelbar wiederholt. Eine weitere Möglichkeit wäre die Steuerung des Roboterarms mithilfe eines Controllers.
Die Benutzung der implementierten Softwareanwendung erfordert Programmierkenntnisse in der Sprache Python. Die **Implementierung einer grafischen Oberfläche** würde die Verwendung des Programms erleichtern.
Die Architektur der Anwendung soll modifiziert werden, sodass das Training und die Verwaltung der Replay-Puffer in zwei **unabhängigen Threads oder Prozessen** ausgeführt werden. Diese Modifikation würde ermöglichen, den Zustand der Replay-Puffer (insbesondere die Priorisierung der Transitionen) auch beim Abbruch des Trainings beizubehalten.

Ein Fehler in der Implementierung des vorgestellten Lernalgorithmus kann sich erst nach mehreren Episoden und Versuchen bemerkbar machen. Um effizient mit der Software weiter arbeiten zu können, müssen **die Maßnahmen zum automatisierten Testen** des Lernalgorithmus konzipiert und umgesetzt werden. Zur Information und Inspiration kann die Publikation von [Zhang et al., 2019] dienen, die die Differenzen im Testen einer klassischen Software und einer ML-Anwendung darstellt sowie einen umfangreichen Überblick über die aktuelle Forschung im Bereich ML-Testing gibt.
Eine weitere Richtung, in welche die Arbeit erweitert werden kann, ist die **Erklärung der gelernten Policy**. Die Erklärbarkeit einer ML-Anwendung ist für das Debugging des Codes, die Akzeptanz und die Zertifizierung wichtig (vgl. [Iyer et al., 2018], S. 1).

A Anhang

A.1 Algorithmen

A.1.1 Adam

Algorithm 4 Adam Algorithmus nach [Goodfellow et al., 2016]

Parameter: Learning rate α (Suggested default: 0.001)
Parameter: Exponential decay rates for moment estimates, p_1 and p_2 in [0;1) (Suggested defaults: 0.9 and 0.999 respectively)
Parameter: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Parameter: initial weights W
Initialize 1st and 2nd moment variables $s = 0$, $r = 0$
Initialize time step $t = 0$
repeat
  Sample a mini-batch of n examples $\{x^{(1)}, ..., x^{(n)}\}$ from the training set with corresponding targets $y^{(i)}$
  Compute gradient: $g \leftarrow \frac{1}{n} \nabla_W \sum_{i=1}^{n} L(f(x^i; W), y^i)$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $s \leftarrow p_1 s + (1 - p_1) g$
  Update biased second moment estimate: $r \leftarrow p_2 r + (1 - p_2) g^{(2)}$
  Correct bias in first moment: $\bar{s} = \frac{s}{1 - {p_1}^t}$
  Correct bias in second moment: $\bar{r} = \frac{r}{1 - {p_2}^t}$
  Compute update: $\Delta W = -\alpha \frac{\bar{s}}{\sqrt{\bar{r}} + \delta}$
  Apply update: $W \leftarrow W + \Delta W$
until stopping criterion not met

A.1.2 DDPG

Algorithm 5 DDPG nach [Lillicrap et al., 2016]

Randomly initialize critic network $Q(s, a; W)$ and actor $\mu(s; \theta)$ with weights W and θ
Initialize target network $\bar{Q}(s, a; \bar{W})$ and $\bar{\mu}(s, a; \bar{\theta})$ with weights $\bar{W} \leftarrow W, \bar{\theta} \leftarrow \theta$
Initialize replay buffer $\mathcal{D}$
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t=1, T **do**
    Select action $a_t = \mu(s_t; \theta) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{D}$
    Set $y_i = r_i + \gamma \cdot \bar{Q}(s_{i+1}, \bar{\mu}(s_{i+1}; \bar{\theta}); \bar{W}_i$
    Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i \left(y_i - Q(s_i, a_i; W)\right)^2$

    Update the actor policy using the sampled policy gradient:
    $\nabla_\theta J(\theta) = \frac{1}{N} \sum_i \nabla_a Q(s_i, \mu(s_i; \theta); W) \nabla_\theta \mu(s_i; \theta)$

    Update the target networks:
    $\bar{W} \leftarrow \tau W - (1 - \tau)\bar{W}$,
    $\bar{\theta} \leftarrow \tau\theta - (1 - \tau)\bar{\theta}$
  end for
end for

A.1.3 TD3

Algorithm 6 TD3 nach [Fujimoto et al., 2018]

Initialize critic networks $Q(s,a;W^1), Q(s,a;W^2)$ and actor network $\mu(s;\theta)$ with random weights W^1, W^2, θ
Initialize target networks $\bar{W}^1 \leftarrow W^1, \bar{W}^2 \leftarrow W^2, \bar{\theta} \leftarrow \theta$
Initialize replay buffer $\mathcal{D}$
for t=1, T **do**
  Select action with exploration noise $a \sim \mu(s;\theta) + \varphi, \quad \varphi \sim \mathcal{N}(0,\sigma)$ and observe reward r and new state s'
  Store transition tuple (s,a,r,s') in $\mathcal{D}$

  Sample mini-batch of n transitions (s,a,r,s') from $\mathcal{D}$
  $\tilde{a} \leftarrow \bar{\mu}(s';\bar{\theta}) + \varphi, \quad \varphi \sim clip((\mathcal{N},0,\tilde{\sigma}),-c,c)$
  Set $y \leftarrow r + \gamma \cdot \min_{k=1,2} \bar{Q}^k(s',\tilde{a};\bar{W}^k)$
  Update critics $W^k \leftarrow argmin_{W^k} \frac{1}{n} \sum \left(y - Q(s,a;W^k)\right)^2$
  if t mod d **then**
    Update θ by the determitistic policy gradient:
    $\nabla_\theta J(\theta) = \frac{1}{n} \sum_i \nabla_a Q(s_i, \mu(s_i;\theta); W^1) \nabla_\theta \mu(s_i;\theta)$
    Update the target networks:
    $\bar{W}^k \leftarrow \tau W^k - (1-\tau)\bar{W}^k$,
    $\bar{\theta} \leftarrow \tau\theta - (1-\tau)\bar{\theta}$
  end if
end for

A.2 Policy-Gradient

Die Abbildung A.1 stellt die Aufteilung des Policy-Gradienten dar.

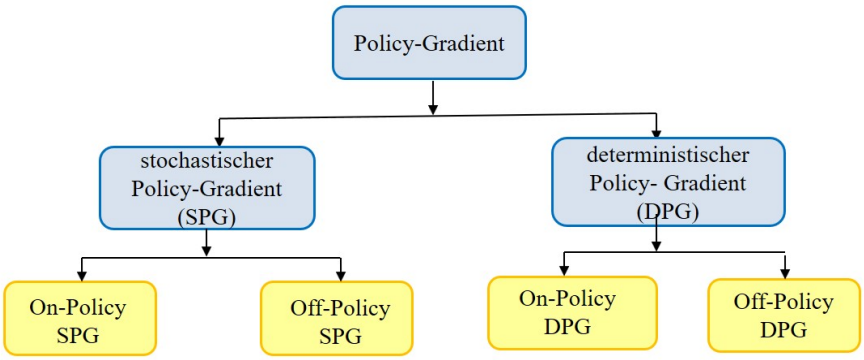


Abbildung A.1: Policy-Gradient
Quelle: Eigene Darstellung

A.2.1 Stochastischer Policy-Gradient (SPG)

On-Policy SPG Die Performanz einer stochastischen Policy $\pi_\theta(a|s)$ in kontinuierlichen $\mathcal{S}$, $\mathcal{A}$ kann wie folgt dargestellt werden:

$$\begin{aligned} J(\theta) = \mathbb{E}_{s_0 \sim \rho^\pi(s_0)}[v^{\pi_\theta}(s_0)] = \\ = \int_{\mathcal{S}} \rho^\pi(s) v^{\pi_\theta}(s) ds = \\ = \int_{\mathcal{S}} \rho^\pi(s) \mathbb{E}_{a \sim \pi_\theta}[q^{\pi_\theta}(s,a)] ds = \\ = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi_\theta(a|s) q^\pi(s,a) da ds \end{aligned}$$

Das *Policy-Gradient-Theorem* definiert den Gradienten einer stochastischen Policy $\pi_\theta(a|s)$ bzgl. der Parameter θ:

$$\begin{aligned} \nabla_\theta \ J(\theta) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) q^\pi(s,a) da ds = \\ = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta}[\nabla_\theta log \pi_\theta(a|s) q^\pi(s,a)] \end{aligned} \tag{A.1}$$

Der Beweis des Theorems ist in ([Sutton et al., 1999], S. 1062- 1063) zu finden.
Das Policy-Gradient-Theorem sagt aus, dass, obwohl die Zustandsverteilung ρ^π von θ abhängt, der Gradient $\nabla_\theta J(\theta)$ nicht vom Gradienten der Zustandsverteilung $\nabla_\theta \rho^\pi$ abhängt. Aus diesem Grund ist es möglich, den Policy-Gradienten modellfrei zu ermitteln.
Bei der Optimierung von $J(\theta)$ wird die Wahrscheinlichkeit der Aktionen proportional zur Größe ihrer Q-Werte mittels der Score-Funktion $\nabla_\theta log \pi_\theta(a|s) = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}$ gesetzt: Die Wahrscheinlichkeit der Aktionen, die einen größeren Q-Wert ergeben, wird erhöht und die

Wahrscheinlichkeit der Aktionen, die zu einem geringen Q-Wert führen, wird verkleinert. Der Erwartungswert in (A.1) kann aus N Episoden nach Policy π_θ ermittelt werden:

$$\nabla_\theta \; J(\theta) \;\approx\; \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})q^\pi(s_{i,t},a_{i,t})$$

Off-Policy SPG In off-Policy-Algorithmen, in denen $\pi_\theta(a|s)$ anhand der Erfahrung aktualisiert wird, die nach $\beta(a|s) \neq \pi_\theta(a|s)$ gesammelt wurde, wird die Policy-Performanz wie folgt modifiziert:

$$J_\beta(\theta) = \int_{\mathcal{S}} \rho^\beta v^\pi(s)ds =$$
$$= \int_{\mathcal{S}} \rho^\beta \int_{\mathcal{A}} \pi_\theta(a|s)q^\pi(s,a)dads$$

Das *Off-Policy-Gradient-Theorem* definiert den off-Policy-Gradienten von $\pi_\theta(a|s)$:

$$\nabla_\theta J_\beta(\theta) = \int_{\mathcal{S}} \rho^\beta \int_{\mathcal{A}} \pi_\theta(a|s)\, q^\pi(s,a)\, dads =$$
$$= \mathbb{E}_{s\sim\rho^\beta, a\sim\beta}[\frac{\pi(a|s)}{\beta(a|s)}\nabla_\theta log\pi_\theta(a|s)q^\pi(s,a)] \tag{A.2}$$

Der Beweis des Theorems ist in [Degris et al., 2012] zu finden.
Das Off-Policy-Gradient-Theorem enthält zwei bedeutende Aspekte:

- Der off-Policy-Gradient hängt nicht von der Q-Funktion der Policy β ab, sondern von der Q-Funktion der Policy π, die selbst off-Policy geschätzt werden muss.
- Zur off-Policy-Korrektur wird die 1-step Importance Sampling Ratio verwendet, d. h. der Unterschied zwischen ρ^β und ρ^π, der nach dem aktuellen Zeitschritt t entsteht, wird ignoriert (vgl. [Liu et al., 2019], S. 2). Dieser Aspekt zielt darauf, die durch IS vergrößerte Varianz des Gradienten-Schätzers zu verkleinern.

A.2.2 Deterministischer Policy-Gradient (DPG)

Bei einer deterministischen Policy $\mu_\theta(s)$ nimmt die Policy-Performanz $J(\theta)$ die folgende Form an:

$$J(\theta) = \mathbb{E}_{s_0\sim\rho^{\mu_\theta}(s_0)}[v^{\mu_\theta}(s_0)] =$$
$$= \int_{\mathcal{S}} \rho^{\mu_\theta}(s_0)v^{\mu_\theta}(s_0)ds =$$
$$= \int_{\mathcal{S}} \rho^{\mu_\theta}(s)q^{\mu_\theta}(s,\mu_\theta(s))ds$$

On-Policy DPG Das *Deterministic-Policy-Gradient-Theorem* definiert den Gradienten einer deterministischen Policy $\mu_\theta(s)$ bzgl. der Parameter θ:

$$\nabla_\theta \ J(\mu_\theta) = \int_{\mathcal{S}} \rho^\mu(s) \ \nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s)) ds = \\ \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s))]$$

Der Beweis des Theorems ist in [Silver et al., 2014] zu finden. Dort wird gezeigt, dass DPG ein Grenzfall von SPG ist.

Off-Policy DPG Der off-Policy-Gradient einer deterministischen Policy ist

$$\nabla_\theta \ J(\mu_\theta) = \int_{\mathcal{S}} \rho^\beta(s) \ \nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s)) ds = \\ = \mathbb{E}_{s\sim\rho^\beta}[\nabla_\theta \mu_\theta(s) \nabla_a q^\mu(s, \mu_\theta(s)] \tag{A.3}$$

Da im Gradienten einer deterministischen Policy die Notwendigkeit, über die Aktionen zu integrieren, entfällt, wird bei der off-Policy-Schätzung des DPG keine Korrektur durch das IS vorgenommen.

A.3 Befüllung des 1-step und des n-step Replay-Puffers

Auf der Abbildung A.2 ist der Prozess der Befüllung des 1-step und des n-step Replay-Puffers für $n = 3, t = 3$ und $T > t$ illustriert.
In jedem Zeitschritt t wird in den Zwischenpuffer rb^{helper} eine neue Transition $(s_{t-1}, a_{t-1}, r_t, s_t, d_t)$ hinzugefügt (Markierung „1“ in der Abb. A.2). Das Ziel ist es, anhand des befüllten Zwischenpuffers rb^{helper} die 1-step Transition $(s_{t-n}, a_{t-n}, r_{t-n+1}, s_{t-n+1}, d_{t-n+1})$ und die n-step Transition $(s_{t-n}, a_{t-n}, \sum_{i=0}^{n-1} \gamma^i r_{t-n+1+i}, s_t, d_t)$ zu erstellen.

1-step Transition
Der erster Eintrag von rb^{helper} ist die gewünschte 1-step Transition. Die Transition wird in rb^{1-step} hinzugefügt (Markierung „3“ auf der Abb. A.2).

n-step Transition
Als s_{t-n} und a_{t-n} werden der Zustand und die Aktion des ersten Eintrages von rb^{helper} zwischengespeichert. Die Variablen $\widetilde{r}_t, \widetilde{s}_t$ und $\widetilde{d}_t$ sind Hilfsvariablen, die im ersten Schritt die Werte der entsprechenden Variablen des letzten Eintrages von rb^{helper} annehmen. Um $\sum_{i=0}^{n-1} \gamma^i r_{t-n+1+i}$ zu bestimmen und neue Werte den Variablen $\widetilde{s}_t, \widetilde{d}_t$ zuzuweisen, falls t der letzte Zeitschritt ist, wird über die Elemente des invertierten Zwischenpuffers (ohne den letzten Eintrag) iteriert. Dieser Prozess ist auf der Abbildung A.2 mit „2“ markiert. Nach der Iteration entsprechen die Werte von Hilfsvariablen $\widetilde{r}_t, \widetilde{s}_t$ und $\widetilde{d}_t$ den benötigten Größen $\sum_{i=0}^{n-1} \gamma^i r_{t-n+1+i}, s_t, d_t$, die n-step Transition kann also in den r^{n-step} hinzugefügt werden.

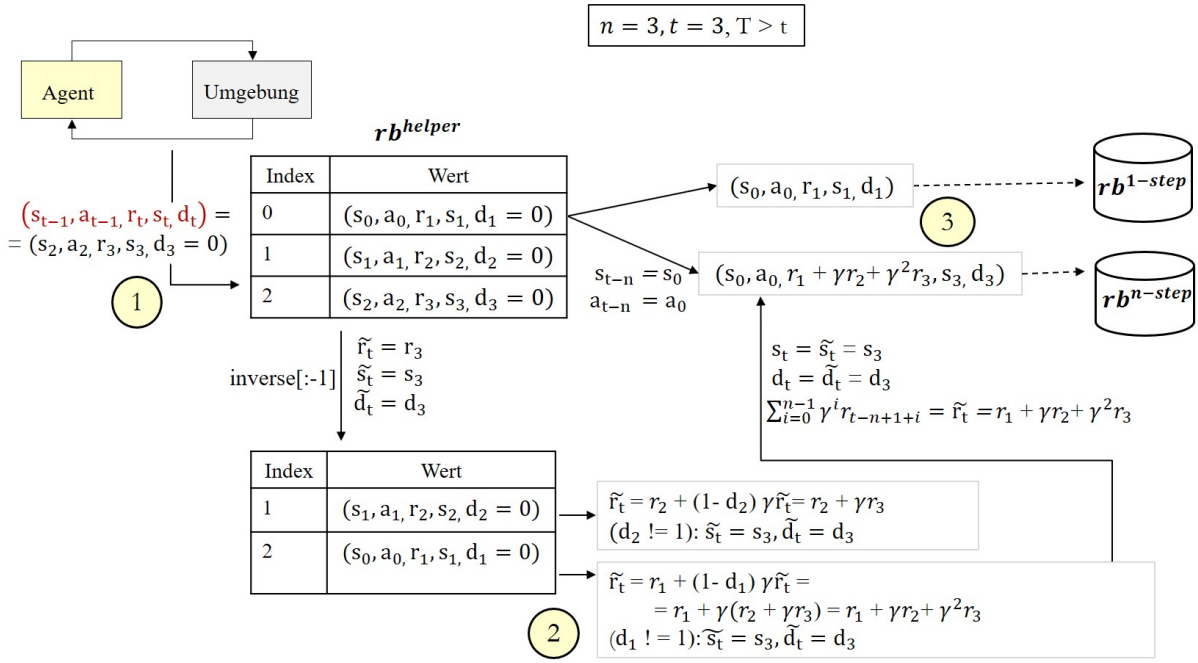


Abbildung A.2: Befüllung des 1-step und des n-step Replay-Puffers
Quelle: Eigene Darstellung

A.4 Die verwendeten Kombinationen der Hyperparameter

A.4.1 Hyperparameter-Kombination h1

```
hyper_params = {
    "N_STEP": 10,
    "PRETRAIN_STEPS": 2000,
    "NUM_EPISODES": 205,
    "GAMMA": 0.99,
    "TAU": 5e-3, # for the soft update of target networks weights
    "BUFFER_SIZE": 500000, # Max number of transitions to store in the buffer. When the buffer overflows the old own memories are dropped.
    "BATCH_SIZE": 128, # number of transitions to sample in each learning step
    "MULTIPLE_LEARN": 5,  # multiple learning updates
    "LAMBDA1": 1.0,  # n-step return weight
    "LAMBDA2": 1e-2,  # l2 regularization weight
    "LAMBDA3": 1.0,  # actor loss contribution of prior weight
    "ALPHA": 0.3, # how much prioritization is used (0 - no prioritization, 1 - full prioritization)
    "BETA": 1.0, # to what degree to use importance weights (0 - no corrections, 1 - full correction)
    "EPS": 1e-4, #  a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero
    "EPS_D": 1, # a small constant that is added to transition priorities from demonstrations
    "POLICY_NOISE": 0.2, # noise for target policy smoothing regularization
    "NOISE_CLIP": 0.1, # absolute value to clip noise
    "EXP_NOISE": 0.3, # noise added to the selected action in training
    "INITIAL_EPISODES": 0,  # number of initial episodes before learning
    "LR_ACTOR": 0.0003,
    "LR_CRITIC": 0.003,
    "HIDDEN_ACTOR": [128, 128],
    "HIDDEN_CRITIC": [128, 128],
    "DISTR_ACTOR": 3e-3, # uniform distribution for the initialisation of the last layer (actor)
    "DISTR_CRITIC": 3e-4, # uniform distribution for the initialisation of the last layer (critic)
    "SEED": [9001, 777, 523]  # random seeds
}
```

A.4.2 Hyperparameter-Kombination h2

Die folgenden Paramter weichen von der Kombination A.4.1 ab:

```
    "HIDDEN_ACTOR": [32, 32],
    "HIDDEN_CRITIC": [32, 32],
```

A.4.3 Hyperparameter-Kombination h3

```
hyper_params = {
    "N_STEP": 50,
    "PRETRAIN_STEPS": 7000,
    "NUM_EPISODES": 201,
    "NUM_FRAMES": 100000,
    "N_FOR_TARGET_UPDATE": 10000,
    "GAMMA": 0.99,
    "TAU": 5e-3, # for the soft update of target networks weights
    "BUFFER_SIZE": 500000, # Max number of transitions to store in the
        buffer. When the buffer overflows the old own
    # memories are dropped
    "BATCH_SIZE": 128, # number of transitions to sample in each learning
        step
    "MULTIPLE_LEARN": 7, # multiple learning updates
    "LAMBDA1": 1.0, # n-step return weight
    "LAMBDA2": 1e-2, # l2 regularization weight
    "LAMBDA3": 1.0, # actor loss contribution of prior weight
    "ALPHA": 0.3, # how much prioritization is used (0 - no prioritization
        , 1 - full prioritization)
    "BETA": 1.0, # to what degree to use importance weights (0 - no
        corrections, 1 - full correction)
    "EPS": 1e-4,
    # a small positive constant that prevents the edge-case of transitions
        not being revisited once their error is zero
    "EPS_D": 1, # a small constant that is added to transition priorities
        from demonstrations
    "POLICY_NOISE": 0.2, # noise for target policy smoothing
        regularization
    "NOISE_CLIP": 0.1, # absolute value to clip noise
    "EXP_NOISE": 0.1, # noise added to the selected action in training
    "INITIAL_EPISODES": 0, # number of initial episodes before learning
    "SOLVED": 200,
    "LR_ACTOR": 0.0003,
    "LR_CRITIC": 0.003,
    "HIDDEN_ACTOR": [256, 256],
    "HIDDEN_CRITIC": [256, 256],
    "SPARSE": True,
    "DISTR_ACTOR": 3e-3, # uniform distribution for the initialisation of
        the last layer (actor)
```

```
    "DISTR_CRITIC": 3e-4,  # uniform distribution for the initialisation of the last layer (critic)
    "SEED": [9001, 777, 523]  # random seeds
}
```

A.4.4 Hyperparameter-Kombination h4

```
hyper_params = {
    "N_STEP": 20,
    "PRETRAIN_STEPS": 4000,
    "NUM_EPISODES": 205,
    "NUM_FRAMES": 100000,
    "N_FOR_TARGET_UPDATE": 10000,
    "GAMMA": 0.99,
    "TAU": 5e-3,  # for the soft update of target networks weights
    "BUFFER_SIZE": 500000,  # max. number of transitions to store in the buffer. When the buffer overflows the old own
    # memories are dropped
    "BATCH_SIZE": 128,  # number of transitions to sample in each learning step
    "MULTIPLE_LEARN": 3,  # multiple learning updates
    "LAMBDA1": 1.0,  # n-step return weight
    "LAMBDA2": 1e-4,  # l2 regularization weight
    "LAMBDA3": 1.0,  # actor loss contribution of prior weight
    "ALPHA": 0.3,  # how much prioritization is used (0 - no prioritization, 1 - full prioritization)
    "BETA": 1.0,  # to what degree to use importance weights (0 - no corrections, 1 - full correction)
    "EPS": 1e-4, # a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero
    "EPS_D": 1,  # a small constant that is added to transition priorities from demonstrations
    "POLICY_NOISE": 0.2,  # noise for target policy smoothing regularization
    "NOISE_CLIP": 0.1,  # absolute value to clip noise
    "EXP_NOISE": 0.3,  # noise added to the selected action in training
    "INITIAL_EPISODES": 0,  # number of initial episodes before learning
    "LR_ACTOR": 0.0003,
    "LR_CRITIC": 0.003,
    "HIDDEN_ACTOR": [256, 256],
    "HIDDEN_CRITIC": [256, 256],
    "SPARSE": True,
    "DISTR_ACTOR": 3e-3,  # uniform distribution for the initialisation of the last layer (actor)
    "DISTR_CRITIC": 3e-4,  # uniform distribution for the initialisation of the last layer (critic)
    "SEED": [9001, 777, 523]  # random seeds
}
```

A.4.5 Hyperparameter-Kombination h5

```
hyper_params = {
    "N_STEP": 20,
    "PRETRAIN_STEPS": 3000,
    "TRAIN_EPISODES": 500,
    "N_FOR_TARGET_UPDATE": 10000,
    "GAMMA": 0.99,
    "TAU": 0.005, # for the soft update of target networks weights
    "BUFFER_SIZE": 500000, # max. number of transitions to store in the
        buffer. When the buffer overflows the old own
    "BATCH_SIZE": 128, # number of transitions to sample in each learning
        step memories are dropped
    "MULTIPLE_LEARN": 10, # multiple learning updates
    "LAMBDA1": 1, # n-step return weight
    "LAMBDA2": 1e-2, # l2 regularization weight
    "LAMBDA3": 1.0, # actor loss contribution of prior weight
    "ALPHA": 0.3, # how much prioritization is used (0 - no prioritization
        , 1 - full prioritization)
    "BETA": 1.0, # to what degree to use importance weights (0 - no
        corrections, 1 - full correction)
    "EPS": 1e-4,
    # a small positive constant that prevents the edge-case of transitions
        not being revisited once their error is zero
    "EPS_D": 1.0, # a small constant that is added to transition
        priorities from demonstrations
    "POLICY_NOISE": 0.2, # noise for target policy smoothing
        regularization
    "NOISE_CLIP": 0.1, # absolute value to clip noise
    "EXP_NOISE": 0.3, # noise added to the selected action in training
    "INITIAL_EPISODES": 0, # number of initial episodes before learning
    "LR_ACTOR": 0.0003,
    "LR_CRITIC": 0.003,
    "HIDDEN_ACTOR": [128, 128],
    "HIDDEN_CRITIC": [128, 128],
    "DISTR_ACTOR": 3e-3,
    "DISTR_CRITIC": 3e-4,
    "SEED": 777
}
```

Literaturverzeichnis

[Abbeel & Ng, 2004] Abbeel, P. & Ng, A. Y. (2004). Apprenticeship Learning via Inverse Reinforcement Learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04 New York, NY, USA: ACM. *https://ai.stanford.edu/~ang/papers/icml04-apprentice.pdf*. Zugriff am 15.11.2019.

[Achiam, 2019] Achiam, J. (2019). *Spinning Up Documentation.* OpenAI. *https://readthedocs.com/projects/openai-education-spinningup/downloads/pdf/latest/*. Zugriff am 30.10.2019.

[Andrychowicz et al., 2017] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., & Zaremba, W. (2017). Hindsight Experience Replay. In *NIPS*. *https://arxiv.org/abs/1707.01495*. Zugriff am 20.10.2019.

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A Brief Survey of Deep Reinforcement Learning. *ArXiv*, abs/1708.05866. *https://arxiv.org/abs/1708.05866*. Zugriff am 28.09.2019.

[Barron, 1993] Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3), 930–945. *https://ieeexplore.ieee.org/document/256500*. Zugriff am 10.11.2019.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. In *IJCAI. https://arxiv.org/pdf/1207.4708.pdf*. Zugriff am 26.11.2019.

[Billard et al., 2008] Billard, A., Calinon, S., Dillmann, R., & Schaal, S. (2008). *Robot Programming by Demonstration*, (pp. 1371–1394). Springer Berlin Heidelberg: Berlin, Heidelberg. *https://www.researchgate.net/publication/227012507_Robot_Programming_by_Demonstration*. Zugriff am 17.11.2019.

[Billard & Grollman, 2013] Billard, A. & Grollman, D. (2013). Robot learning by demonstration. *Scholarpedia*, 8(12), 3824. *http://www.scholarpedia.org/w/index.php?title=Robot_learning_by_demonstration&action=cite&rev=138061*. Zugriff am 17.11.2019.

[Boersch et al., 2007] Boersch, I., Heinsohn, J., & Socher, R. (2007). *Wissensverarbeitung: Eine Einführung in die Künstliche Intelligenz für Informatiker und Ingenieure.* Heidelberg: Spektrum, 2 edition.

[Bojarski et al., 2016] Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). End to End Learning for Self-Driving Cars. *ArXiv*, abs/1604.07316. *https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf*. Zugriff am 17.11.2019.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *ArXiv*, abs/1606.01540. *https://arxiv.org/abs/1606.01540*. Zugriff am 26.11.2019.

[Choi et al., 2019] Choi, S., Le, T. P., Nguyen, Q. D., Layek, M. A., Lee, S., & Chung, T. (2019). Toward Self-Driving Bicycles Using State-of-the-Art Deep Reinforcement Learning Algorithms. *Symmetry*, 11(2), 290. *https://www.mdpi.com/2073-8994/11/2/290/htm*. Zugriff am 30.10.2019.

[Costa et al., 2015] Costa, B., Caarls, W., & Menasché, D. S. (2015). Dyna-MLAC: Trading Computational and Sample Complexities in Actor-Critic Reinforcement Learning. *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, (pp. 37–42). *https://ieeexplore.ieee.org/document/7423912*. Zugriff am 28.09.2019.

[Craig, 2005] Craig, J. J. (2005). *Introduction to Robotics: Mechanics and Control*. Pearson Education International, third edition.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314. *https://link.springer.com/article/10.1007/BF02551274*. Zugriff am 10.11.2019.

[de Bruin et al., 2018] de Bruin, T., Kober, J., Tuyls, K., & Babuška, R. (2018). Experience Selection in Deep Reinforcement Learning for Control. *Journal of Machine Learning Research*, 19(9), 1–56. *http://www.jmlr.org/papers/volume19/17-131/17-131.pdf*. Zugriff am 14.12.2019.

[Degris et al., 2012] Degris, T., White, M., & Sutton, R. S. (2012). Off-Policy Actor-Critic. *CoRR*, abs/1205.4839. *https://arxiv.org/abs/1205.4839*. Zugriff am 20.10.2019.

[Deisenroth et al., 2013] Deisenroth, M. P., Neumann, G., & Peters, J. (2013). A Survey on Policy Search for Robotics. *Found. Trends Robot*, 2(1–2), 1–142. *https://core.ac.uk/download/pdf/84341151.pdf*. Zugriff am 28.09.2019.

[Deisenroth & Rasmussen, 2011] Deisenroth, M. P. & Rasmussen, C. E. (2011). PILCO: A Model-Based and Data-Efficient Approach to Policy Search. In *ICML*. *http://mlg.eng.cam.ac.uk/pub/pdf/DeiRas11.pdf*. Zugriff am 28.09.2019.

[Dhariwal et al., 2017] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., & Zhokhov, P. (2017). OpenAI Baselines. *https://github.com/openai/baselines*. Zugriff am 27.11.2019.

[Dulac-Arnold et al., 2015] Dulac-Arnold, G., Evans, R., Sunehag, P., & Coppin,

B. (2015). Reinforcement Learning in Large Discrete Action Spaces. *ArXiv*, abs/1512.07679. *https://arxiv.org/pdf/1512.07679.pdf*. Zugriff am 27.10.2019.

[Fang et al., 2019] Fang, M., Zhou, C., Shi, B., Gong, B., Xu, J., & Zhang, T. (2019). DHER: Hindsight Experience Replay for Dynamic Goals. In *ICLR*. *https://openreview.net/pdf?id=Byf5-30qFX*. Zugriff am 26.11.2019.

[Fujimoto et al., 2019] Fujimoto, S., Hoof, H., & Meger, D. (2018-2019). Git Repository: TD3. *https://github.com/sfujim/TD3*. Zugriff am 27.11.2019.

[Fujimoto et al., 2018] Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *CoRR*, abs/1802.09477. *https://arxiv.org/pdf/1802.09477.pdf*. Zugriff am 13.10.2019.

[García & Fernández, 2019] García, J. & Fernández, F. (2019). Probabilistic Policy Reuse for Safe Reinforcement Learning. *ACM Trans. Auton. Adapt. Syst.*, 13(3), 14:1 – 14:24. *https://dl.acm.org/citation.cfm?id=3310090*. Zugriff am 05.05.2019.

[Giusti et al., 2016] Giusti, A., Guzzi, J., Cireşan, D. C., He, F.-L., Rodríguez, J. P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Caro, G. D., Scaramuzza, D., & Gambardella, L. M. (2016). A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. *IEEE Robotics and Automation Letters*, 1, 661–667. *https://ieeexplore.ieee.org/document/7358076*. Zugriff am 17.11.2019.

[Goecks et al., 2019] Goecks, V. G., Gremillion, G. M., Lawhern, V. J., Valasek, J., & Waytowich, N. R. (2019). Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Sparse Reward Environments. *ArXiv*, abs/1910.04281. *https://arxiv.org/abs/1910.04281*. Zugriff am 17.11.2019.

[Goo & Niekum, 2018] Goo, W. & Niekum, S. (2018). One-Shot Learning of Multi-Step Tasks from Observation via Activity Localization in Auxiliary Video. *2019 International Conference on Robotics and Automation (ICRA)*, (pp. 7755–7761). *https://arxiv.org/pdf/1806.11244.pdf*. Zugriff am 17.11.2019.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. The MIT Press.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2018). *Soft Actor Critic - Deep Reinforcement Learning with Real-World Robots*. Technical report, Berkeley Artifical Intelligence Research (BAIR). *https://bair.berkeley.edu/blog/2018/12/14/sac/*. Zugriff am 28.10.2019.

[Hasselt, 2010] Hasselt, H. V. (2010). Double Q-learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10 (pp. 2613–2621). USA: Curran Associates Inc. *https://papers.nips.cc/paper/3964-double-q-learning.pdf*. Zugriff am 13.10.2019.

[Hawley et al., 2018] Hawley, L., Rahem, R., & Suleiman, W. (2018). Kalman Filter Based Observer for an External Force Applied to Medium-sized Humanoid Robots. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (pp. 1204–1211). *https://hal.archives-ouvertes.fr/hal-01862107/document*. Zugriff am 08.12.2019.

[Henderson et al., 2017] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2017). Deep Reinforcement Learning that Matters. *ArXiv*, abs/1709.06560. *https://arxiv.org/abs/1709.06560*. Zugriff am 27.11.2019.

[Hernandez-Garcia & Sutton, 2019] Hernandez-Garcia, J. F. & Sutton, R. S. (2019). Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target. *CoRR*, abs/1901.07510.

[Hester et al., 2017] Hester, T., Vecerík, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J., Leibo, J. Z., & Gruslys, A. (2017). Deep Q-learning from Demonstrations. *ArXiv*, abs/1704.03732. *https://arxiv.org/pdf/1704.03732.pdf*. Zugriff am 26.11.2019.

[Horgan et al., 2018] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., & Silver, D. (2018). Distributed Prioritized Experience Replay. *ArXiv*, abs/1803.00933. *https://arxiv.org/pdf/1803.00933.pdf*. Zugriff am 26.11.2019.

[Hussein et al., 2017] Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation Learning: A Survey of Learning Methods. *ACM Comput. Surv.*, 50(2), 21:1–21:35. *http://www.open-access.bcu.ac.uk/5045/1/Imitation%20Learning%20A%20Survey%20of%20Learning%20Methods.pdf*. Zugriff am 17.11.2019.

[IFR, 2018] IFR (2018). *Executive Summary World Robotics 2018 Service Robots*. Technical report, International Federation of Robotic. *https://ifr.org/downloads/press2018/Executive_Summary_WR_Service_Robots_2018.pdf*. Zugriff am 11.08.2019.

[Islam et al., 2017] Islam, R., Henderson, P., Gomrokchi, M., & Precup, D. (2017). Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *ArXiv*, abs/1708.04133. *https://arxiv.org/abs/1708.04133*. Zugriff am 04.12.2019.

[Iyer et al., 2018] Iyer, R., Li, Y., Li, H., Lewis, M., Sundar, R., & Sycara, K. P. (2018). Transparency and Explanation in Deep Reinforcement Learning Neural Networks. In *AIES '18. https://arxiv.org/pdf/1809.06061.pdf*. Zugriff am 16.12.2019.

[Kim et al., 2016] Kim, D., Jorgensen, S. J., Stone, P., & Sentis, L. (2016). Dynamic behaviors on the NAO robot with closed-loop whole body operational space control. *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, (pp. 1121–1128). *https://ieeexplore.ieee.org/abstract/document/7803411*. Zugriff am 08.12.2019.

[Kingma & Ba, 2014] Kingma, D. P. & Ba, J. (2014). Adam: A Method for Stochastic

Optimization. *CoRR*, abs/1412.6980. *https://arxiv.org/abs/1412.6980*. Zugriff am 10.11.2019.

[Kriesel, 2007] Kriesel, D. (2007). *Ein kleiner Überblick über Neuronale Netze. http://www.dkriesel.com/_media/science/neuronalenetze-de-zeta2-2col-dkrieselcom.pdf*. Zugriff am 29.08.2019.

[Kumar et al., 2018] Kumar, A., Paul, N., & Omkar, S. N. (2018). Bipedal Walking Robot using Deep Deterministic Policy Gradient. *CoRR*, abs/1807.05924. *https://arxiv.org/abs/1807.05924*. Zugriff am 30.10.2019.

[KurtHornik et al., 1989] KurtHornik, Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 359 – 366. *https://www.sciencedirect.com/science/article/pii/0893608089900208*. Zugriff am 10.11.2019.

[Levine, 2019a] Levine, S. (2019a). Deep RL with Q-Functions. CS 285 at UC Berkeley, *http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-8.pdf*. Zugriff am 13.10.2019.

[Levine, 2019b] Levine, S. (2019b). Introduction to Reinforcement Learning. CS 285 at UC Berkeley, *http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-4.pdf*. Zugriff am 06.11.2019.

[Levine, 2019c] Levine, S. (2019c). Inverse Reinforcement Learning. CS 285 at UC Berkeley, *http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_12_irl.pdf*. Zugriff am 17.11.2019.

[Lillicrap et al., 2016] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Y. Bengio & Y. LeCun (Eds.), *ICLR*. *https://arxiv.org/abs/1509.02971*. Zugriff am 21.06.2019.

[Lin, 1992] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3), 293–321. *https://link.springer.com/article/10.1007/BF00992699*. Zugriff am 12.10.2019.

[Liu et al., 2019] Liu, Y., Swaminathan, A., Agarwal, A., & Brunskill, E. (2019). Off-Policy Policy Gradient with State Distribution Correction. *CoRR*, abs/1904.08473. *https://arxiv.org/pdf/1904.08473.pdf*. Zugriff am 26.10.2019.

[Lynnerup et al., 2019] Lynnerup, N. A., Nolling, L., Hasle, R., & Hallam, J. (2019). A Survey on Reproducibility by Evaluating Deep Reinforcement Learning Algorithms on Real-World Robots. *ArXiv*, abs/1909.03772. *arxiv.org/pdf/1909.03772.pdf*. Zugriff am 13.12.2019.

[Medipixel Inc., 2019] Medipixel Inc. (2018-2019). Git Repository: RL Algorirhms. *https://github.com/medipixel/rl_algorithms*. Zugriff am 27.11.2019.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *CoRR*, abs/1602.01783. *https://arxiv.org/pdf/1602.01783.pdf*. Zugriff am 15.10.2019.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602. https://arxiv.org/abs/1312.5602*. Zugriff am 13.10.2019.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518, 529–533. *https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf*. Zugriff am 10.12.2019.

[Nagabandi et al., 2017] Nagabandi, A., Kahn, G., Fearing, R. S., & Levine, S. (2017). Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 7559–7566). *https://arxiv.org/abs/1708.02596*. Zugriff am 29.09.2019.

[Nair et al., 2017] Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2017). Overcoming Exploration in Reinforcement Learning with Demonstrations. *CoRR*, abs/1709.10089. *https://arxiv.org/pdf/1709.10089.pdf*. Zugriff am 11.08.2019.

[Ng, 2017] Ng, A. (2017). CS229 Lecture notes: Learning Theory. *http://cs229.stanford.edu/notes/cs229-notes4.pdf*. Zugriff am 09.12.2019.

[OpenAi, 2019] OpenAi (2019). How to Train Your OpenAI Five. *https://openai.com/blog/how-to-train-your-openai-five/*. Zugriff am 15.11.2019.

[OpenAI Spinning Up, 2018] OpenAI Spinning Up (2018). *Kinds of RL Algorithms. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below*. Zugriff am 30.10.2019.

[Osa et al., 2018] Osa, T., Pajarinen, J., Neumann, G., Bagnell, J. A., Abbeel, P., & Peters, J. (2018). An Algorithmic Perspective on Imitation Learning. *Foundations and Trends in Robotics*, 7, 1–179. *https://arxiv.org/ftp/arxiv/papers/1811/1811.06711.pdf*. Zugriff am 17.11.2019.

[Ott et al., 2013] Ott, C., Henze, B., & Lee, D. (2013). Kinesthetic teaching of humanoid motion based on whole-body compliance control with interaction-aware balancing. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 4615–4621). *https://ieeexplore.ieee.org/document/6697020*. Zugriff am 17.11.2019.

[Penedones et al., 2019] Penedones, H., Riquelme, C., Vincent, D., Maennel, H., Mann, T. A., Barreto, A., Gelly, S., & Neu, G. (2019). Adaptive Temporal-Difference

Learning for Policy Evaluation with Per-State Uncertainty Estimates. *CoRR*, abs/1906.07987. *https://arxiv.org/pdf/1906.07987.pdf*. Zugriff am 08.11.2019.

[Piot et al., 2014] Piot, B., Geist, M., & Pietquin, O. (2014). Boosted Bellman Residual Minimization Handling Expert Demonstrations. In *ECML/PKDD*. *http://chercheurs.lille.inria.fr/~bpiot/pdf/ECML2014_paper.pdf*. Zugriff am 26.11.2019.

[Plappert, 2016] Plappert, M. (2016). Keras-RL. *https://github.com/keras-rl/keras-rl*. Zugriff am 27.11.2019.

[Pohlen et al., 2018] Pohlen, T., Piot, B., Hester, T., Azar, M. G., Horgan, D., Budden, D., Barth-Maron, G., van Hasselt, H., Quan, J., Vecerík, M., Hessel, M., Munos, R., & Pietquin, O. (2018). Observe and Look Further: Achieving Consistent Performance on Atari. *ArXiv*, abs/1805.11593. *https://arxiv.org/pdf/1805.11593.pdf*. Zugriff am 26.11.2019.

[Poole & Mackworth, 2017] Poole, D. & Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge, UK: Cambridge University Press, 2. edition.

[Ramachandran & Amir, 2007] Ramachandran, D. & Amir, E. (2007). Bayesian Inverse Reinforcement Learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07 (pp. 2586–2591). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. *https://www.aaai.org/Papers/IJCAI/2007/IJCAI07-416.pdf*. Zugriff am 17.11.2019.

[Rashid, 2017] Rashid, T. (2017). *Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python*. Heidelberg: O'Reilly, 1. edition.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature*, 323(6088), 533–536. *https://www.nature.com/articles/323533a0*. Zugriff am 21.09.2019.

[Sampedro et al., 2019] Sampedro, C., Rodriguez-Ramos, A., Bavle, H., Carrio, A., de la Puente, P., & Campoy, P. (2019). A Fully-Autonomous Aerial Robot for Search and Rescue Applications in Indoor Environments using Learning-Based Techniques. *Journal of Intelligent & Robotic Systems*, 95(2), 601–627. *https://link.springer.com/article/10.1007/s10846-018-0898-1*. Zugriff am 29.10.2019.

[Schaul et al., 2015] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized Experience Replay. *CoRR*, abs/1511.05952. *https://arxiv.org/pdf/1511.05952.pdf*. Zugriff am 12.10.2019.

[Schwab et al., 2019] Schwab, D., Springenberg, J. T., Martins, M. F., Lampe, T., Neunert, M., Abdolmaleki, A., Hertweck, T., Hafner, R., Nori, F., & Riedmiller, M. A. (2019). Simultaneously Learning Vision and Feature-based Control Policies for Real-world Ball-in-a-Cup. *ArXiv*, abs/1902.04706. *https://arxiv.org/abs/1902.04706*. Zugriff am 24.09.2019.

[Silver, 2015] Silver, D. (2015). UCL Course on RL, Lecture 7: Policy Gradient. *http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf*. Zugriff am 24.10.2019.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815. *https://arxiv.org/abs/1712.01815*. Zugriff am 27.10.2019.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14 (pp. 387–395).: JMLR.org. *http://proceedings.mlr.press/v32/silver14.pdf*. Zugriff am 15.10.2019.

[SoftBank Robotics, 2013] SoftBank Robotics (2013). Architecture of the pref file Device.xml / of key/values in ALMemory. *http://doc.aldebaran.com/1-14/naoqi/sensors/dcm/pref_file_architecture.html?highlight=normal%20temperature*. Zugriff am 15.12.2019.

[SoftBank Robotics, 2017] SoftBank Robotics (2017). NAO Documentation. *http://doc.aldebaran.com/2-1/home_nao.html*. Zugriff am 28.07.2019.

[Sutton & Barto, 2018] Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

[Sutton et al., 1999] Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99 (pp. 1057–1063). Cambridge, MA, USA: MIT Press. *https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf*. Zugriff am 24.10.2019.

[Thrun & Schwartz, 1993] Thrun, S. & Schwartz, A. (1993). Issues in Using Function Approximation for Reinforcement Learning. In *Proceedings of the Fourth Connectionist Models Summer School*. *https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf*. Zugriff am 13.10.2019.

[Todorov et al., 2012] Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (pp. 5026–5033). *https://ieeexplore.ieee.org/document/6386109*. Zugriff am 10.12.2019.

[Tokic, 2013] Tokic, M. (2013). *Reinforcement Learning mit adaptiver Steuerung von Exploration und Exploitation*. PhD thesis, Universität Ulm, Institut für Neuroinformatik. *https://pdfs.semanticscholar.org/e318/60a420833f0f8c793e6fb3b747db9f36029e.pdf*. Zugriff am 08.11.2019.

[Uhlenbeck & Ornstein, 1930] Uhlenbeck, G. E. & Ornstein, L. S. (1930). On the Theory of the Brownian Motion. *Phys. Rev.*, 36, 823–841. *https://journals.aps.org/pr/abstract/10.1103/PhysRev.36.823*. Zugriff am 16.11.2019.

[Vecerík et al., 2017] Vecerík, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., & Riedmiller, M. A. (2017). Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards. *ArXiv*, abs/1707.08817. *https://arxiv.org/pdf/1707.08817.pdf*. Zugriff am 20.05.2019.

[Webots, 2019] Webots (2019). Webots: Commercial Mobile Robot Simulation Software. *http://www.cyberbotics.com*. Zugriff am 15.12.2019.

[White & White, 2019a] White, M. & White, A. (2019a). Bellman Equation Derivation. *https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/lecture/X5VDU/bellman-equation-derivation*. Zugriff am 07.11.2019.

[White & White, 2019b] White, M. & White, A. (2019b). Generalization and Discrimination. *https://www.coursera.org/learn/prediction-control-function-approximation/lecture/sJx1I/generalization-and-discrimination*. Zugriff am 07.11.2019.

[White & White, 2019c] White, M. & White, A. (2019c). Sample-based Learning Methods: How is Q-learning off-policy? *https://www.coursera.org/learn/sample-based-learning-methods/lecture/1OikH/how-is-q-learning-off-policy*. Zugriff am 06.10.2019.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), 229–256. *https://link.springer.com/article/10.1007/BF00992696*. Zugriff am 27.10.2019.

[Wulfmeier et al., 2015] Wulfmeier, M., Ondruska, P., & Posner, I. (2015). Maximum Entropy Deep Inverse Reinforcement Learning. *https://arxiv.org/pdf/1507.04888.pdf*. Zugriff am 17.11.2019.

[Zeng et al., 2019] Zeng, A., Song, S., Lee, J., Rodriguez, A., & Funkhouser, T. A. (2019). TossingBot: Learning to Throw Arbitrary Objects with Residual Physics. *ArXiv*, abs/1903.11239. *https://arxiv.org/pdf/1903.11239.pdf*. Zugriff am 29.10.2019.

[Zhang et al., 2019] Zhang, J. M., Harman, M., Ma, L., & Liu, Y. (2019). Machine Learning Testing: Survey, Landscapes and Horizons. *CoRR*, abs/1906.10742. *https://arxiv.org/pdf/1906.10742.pdf*. Zugriff am 16.12.2019.

[Zhu & Hu, 2018] Zhu, Z. & Hu, H. (2018). Robot Learning from Demonstration in Robotic Assembly: A Survey. *Robotics*, 7, 17. *https://res.mdpi.com/robotics/robotics-07-00017/article_deploy/robotics-07-00017.pdf?filename=&attachment=1*. Zugriff am 17.11.2019.

[Ziebart et al., 2008] Ziebart, B. D., Bagnell, J. A., & Dey, A. K. (2008). Maxi-

mum Entropy Inverse Reinforcement Learning. In *Proc. AAAI* (pp. 1433–1438). *https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf*. Zugriff am 17.11.2019.

[Ziebart et al., 2010] Ziebart, B. D., Bagnell, J. A., & Dey, A. K. (2010). Modeling Interaction via the Principle of Maximum Causal Entropy. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10 (pp. 1255–1262). USA: Omnipress. *http://www.cs.cmu.edu/~bziebart/publications/maximum-causal-entropy.pdf*. Zugriff am 17.11.2019.

Abbildungsverzeichnis

Tabellenverzeichnis