



MASTERSTUDIENGANG

EntwicklungsingenieurIn Maschinenbau

---

**Tetromino-basierte Prozessoptimierung von  
roboterunterstützten Chargierungs- und  
Parkettierungsaufgaben mittels neuronaler  
Netze**

Als MASTERARBEIT eingereicht

zur Erlangung des akademischen Grades

Diplom-Ingenieur für technisch-wissenschaftliche Berufe (Dipl.-Ing.)

von

**Christian Brandstätter, BSc.**

August 2021

---

Betreuung der Arbeit durch

FH-Prof. Dr. Roman Froschauer

# **Vorwort/Danksagung**

Ich bedanke mich bei meinem Betreuer Herrn FH-Prof. DI (FH) Dr. techn. Roman Franz Froschauer für die freundliche Unterstützung bei meiner Diplomarbeit. Des Weiteren bedanke ich mich bei den Mitarbeitern aus dem Unternehmen Rübig für die gute Zusammenarbeit.

Besonderer Dank gilt meiner Familie und meinen Freunden für den Beistand während meines Studiums. Danke.

## Eidesstattliche Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Die vorliegende Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

.....  
Christian Brandstätter, BSc.

Wels, August 2021

## **Kurzfassung**

In dieser Arbeit wird ein Automatisierungslösungsschema für den Chargierprozess von Härteanlagen entwickelt. Die Chargierung beinhaltet die Ablage von Bauteilen auf ein Chargiergestell und die Bestückung der Härteanlagen mit diesen Gestellen. Am Anfang der Arbeit steht eine Konkretisierung der Aufgabenstellung, dabei wird der abzubildende Prozess untersucht und in den Kontext zur Aufgabenbeschreibung gestellt.

Im Zuge der Bearbeitung wird der momentane Stand der Technik erarbeitet und mit Literatur gestützt. Im Fokus der Recherche stehen die Bereiche Automatisierungsstruktur, Kommunikation, Softwareentwicklung, Parkettierung und Künstliche Intelligenz. Auf Basis dieses Wissens werden Konzepte erstellt und deren Inhalte gegliedert. Anschließend werden die Inhalte der Konzepte für einen Prototypen implementiert.

Hierbei wird ein Konfigurationsprogramm erstellt, welches die teilnehmenden Anlagenkomponenten definiert und eine geometrische Anordnung dieser ermöglicht. Das generierte Layout wird anschließend in eine erstellte 2D-Simulationsumgebung geladen und anhand einer vorab festgelegten Prozesslogik animiert, welche die Parkettierung der Bauteile und den Ablauf des Gesamtprozesses beinhaltet. Des Weiteren wird dieses Programm herangezogen, um eine Verbindung zu einer Roboter-Trainingszelle herzustellen und somit die Prozesslogik auf eine Realanlage zu übertragen.

Für die Parkettierung komplexer Bauteile wird eine Approximation durch Tetrominos durchgeführt. Das so entstehende Parkettierungsproblem wird in einem ersten Ansatz mithilfe neuronaler Netzwerke gelöst.

Es folgen Lösungsansätze und Ideen für weitere Inhalte einer Umsetzung, wie Erweiterung der Funktionalität und Verbesserungsvorschläge der Implementierung. Dabei werden auch Möglichkeiten genannt, die aus Zeitgründen in die Arbeit nicht aufgenommen werden konnten.

# Abstract

In this work, an automation solution scheme for the charging process of hardening plants is developed. Charging includes the depositing of components on a charging rack and the loading of the hardening systems with these racks. At the beginning a concretization of the given task is taking place. Furthermore, the process to be mapped is examined under the context of the task description.

In the course of this, the current state of the art is elaborated and supported with proper literature. The research focuses on the areas of automation structure, communication, software development, tiling and artificial intelligence. Based on this knowledge, concepts are created and their contents structured. Subsequently, the contents of concepts are implemented for a prototype.

A configuration program is created that defines the participating system components and enables them to be arranged geometrically. Then, the created layout is loaded into a created 2D simulation environment and is animated using a predefined process logic. The process logic includes the tiling of the components and sequences of the overall process. This program is also used to establish a connection to a robot training cell and hence, transfers the process logic to a real system.

For packing of complex components, an approximation through Tetriminoes is carried out. The resulting packing problem is solved in a first approach with the help of neural networks.

This is followed by solutions, approaches and ideas for further contents of an implementation, such as the extension of functionality and suggestions for application improvements. Possibilities are also mentioned that could not be included in the work due to time constraints.

# Inhaltsverzeichnis

<b>Vorwort/Danksagung</b>	<b>II</b>
<b>Eidesstattliche Erklärung</b>	<b>III</b>
<b>Kurzfassung</b>	<b>IV</b>
<b>Abstract</b>	<b>V</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problemstellung . . . . .	1
1.3 Forschungsfrage . . . . .	1
1.4 Ziel und Inhalt dieser Arbeit . . . . .	2
1.5 Umsetzungsstrategie . . . . .	2
1.6 Voraussetzungen . . . . .	3
<b>2 Recherche</b>	<b>4</b>
2.1 Automatisierungspyramide . . . . .	4
2.2 IEC 61131-3 . . . . .	5
2.2.1 Projektstruktur . . . . .	5
2.2.2 Konfigurationselemente . . . . .	6
2.2.3 Programmorganisationseinheit . . . . .	7
2.2.4 Variablen und Datentypen . . . . .	7
2.2.5 Programmiersprachen . . . . .	8
2.3 OPC Unified-Architecture . . . . .	10
2.3.1 Adressraum . . . . .	11
2.3.2 Browsing . . . . .	11
2.3.3 Subscription . . . . .	12
2.3.4 Node-Modell . . . . .	13
2.3.5 Nodeart . . . . .	13
2.3.6 Kommunikationswege . . . . .	14
2.3.7 Funktions- und Programmaufruf . . . . .	15
2.4 Unified Markup Language . . . . .	15
2.4.1 Klasse . . . . .	16
2.4.2 Attribut . . . . .	16
2.4.3 Assoziation . . . . .	17

2.4.4	Abgeleitetes Attribut . . . . .	20
2.4.5	Vererbung . . . . .	21
2.4.6	Vorteile . . . . .	21
2.4.7	Aktivitätsdiagramm . . . . .	22
2.5	Parkettierung der Ebene . . . . .	23
2.5.1	Platonische Parkettierung . . . . .	23
2.5.2	Kepler'sche Vermutung . . . . .	24
2.6	Machine Learning . . . . .	25
2.6.1	Basics . . . . .	25
2.6.1.1	Aufbau eines Neurons . . . . .	25
2.6.1.2	Aktivierungsfunktionen eines Neurons . . . . .	27
2.6.1.3	Bestimmung der Gewichtungsfaktoren . . . . .	28
2.6.1.4	Initialisierung von Gewichtungsfaktoren . . . . .	31
2.6.1.5	Kostenfunktional . . . . .	32
2.6.1.6	Regularization und Dropout . . . . .	33
2.6.1.7	Notation für Features und Observations . . . . .	34
2.6.1.8	Hyper-Parameter-Tuning . . . . .	36
2.6.2	Supervised Learning . . . . .	37
2.6.2.1	Feedforward Neural Networks . . . . .	37
2.6.2.2	Convolutional Neural Networks . . . . .	40
2.6.2.3	Recurrent Neural Networks . . . . .	43
2.6.3	Unsupervised Learning . . . . .	45
2.6.4	Reinforcement Learning . . . . .	45
2.6.4.1	Policy . . . . .	46
2.6.4.2	Rewards und Exploration Versus Exploitation Dilemma . . . . .	47
2.6.4.3	Markov Process . . . . .	48
2.6.4.4	Bellman-Gleichung . . . . .	52
2.6.4.5	Lösen der Bellman-Gleichung . . . . .	54
2.6.4.6	Temporal Difference Learning . . . . .	56
2.6.4.7	Deep Q-Learning . . . . .	57
<b>3</b>	<b>Konzeptentwicklung</b>	<b>60</b>
3.1	Workflow . . . . .	60
3.2	Definition der Klassen . . . . .	62
3.3	Architekturentwurf . . . . .	63
3.3.1	Modul für die Anlagenkomponenten . . . . .	63
3.3.2	Modul für das User-Interface . . . . .	64
3.3.3	Modul für die Parkettierungsvorschrift . . . . .	65

3.3.4	Modul für die Kommunikationstreiber . . . . .	66
3.3.5	Aufbau der Gesamtapplikation . . . . .	66
3.3.6	Codegenerierung . . . . .	67
3.4	Parkettierungsstrategie . . . . .	68
3.4.1	Parkettierung von zylindrischen Bauteilen . . . . .	68
3.4.2	Parkettierung von Tetrominos . . . . .	74
3.4.2.1	Environment and Rewards . . . . .	74
3.4.2.2	Agent . . . . .	76
3.4.2.3	Observation und Features . . . . .	79
3.4.2.4	Neuronale Netzwerke . . . . .	81
3.4.2.5	Benchmark . . . . .	83
3.4.3	Bestimmung der Transitions-Wahrscheinlichkeiten . . . . .	83
3.4.4	Parkettierung durch Layoutvorgabe . . . . .	84
3.4.5	Parkettierungsfilter . . . . .	84
3.5	Steuerung der Trainingszelle . . . . .	85
3.5.1	Aufbau der Trainingszelle . . . . .	86
3.5.2	OPC UA Datenmodell . . . . .	86
3.5.3	Kommunikationsverlauf . . . . .	87
<b>4</b>	<b>Implementierung</b>	<b>89</b>
4.1	Simulationsprogramm . . . . .	89
4.1.1	Aufbau des Simulationsprogramms . . . . .	89
4.1.2	Layoutkonfiguration . . . . .	91
4.1.3	Lineare und rotatorische Bewegung . . . . .	92
4.1.4	Subscriptions . . . . .	94
4.1.5	Komplexe Rotation . . . . .	95
4.1.6	Zweidimensionale Wegoptimierung . . . . .	96
4.2	Anpassung an Trainingszelle . . . . .	97
4.2.1	Layout . . . . .	98
4.2.2	Simulation und Steuerung . . . . .	99
4.3	KI-Implementierung . . . . .	99
4.3.1	Lernroutine . . . . .	100
4.3.2	Neuronale Netzwerke . . . . .	104
4.4	KI-Training . . . . .	107
4.4.1	Placement Agent . . . . .	107
4.4.2	Controller Agent . . . . .	111
4.4.3	Conclusio . . . . .	114
4.5	Einbindung in das Simulationsprogramm . . . . .	115

<b>5 Zusammenfassung und Ausblick</b>	<b>117</b>
<b>6 Akronymverzeichnis</b>	<b>119</b>
<b>7 Abbildungsverzeichnis</b>	<b>120</b>
<b>8 Tabellenverzeichnis</b>	<b>123</b>
<b>9 Formelverzeichnis</b>	<b>124</b>
<b>10 Programmverzeichnis</b>	<b>126</b>
<b>11 Literaturverzeichnis</b>	<b>127</b>
<b>12 Anhang</b>	<b>131</b>
12.1 Grundlagen aus der Statistik . . . . .	131
12.2 Verwendete Hard- und Software . . . . .	131
12.3 Zustände der Simulation-Member-Klassen . . . . .	132
12.4 OPC UA Datenmodell . . . . .	135
12.5 Parameterlisten der Versuchsdurchführung . . . . .	136
12.6 Strategie für die manuelle Interaktion . . . . .	143

# 1 Einleitung

In diesem Kapitel wird die Problemstellung vorgestellt und geeignete Forschungsfragen daraus abgeleitet. Anschließend werden Ziele und Nicht-Ziele definiert und eine passende Richtlinie für die Konzeptentwicklung ausgewählt.

## 1.1 Motivation

In Kooperation mit dem Unternehmen RÜBIG wird eine geeignete Softwarearchitektur und Herangehensweise gesucht, um die Automatisierung deren Anlagen voranzutreiben. Der eigentliche Wunsch ist, die Bestückung von Plasmanitrieranlagen weitgehendst zu automatisieren und dies als Automatisierungslösung zu vermarkten. Bei den Anlagen handelt es sich um Prozessanlagen, denen Bauteile zugeführt werden, um deren Oberfläche aufzuhärten. Die Chargierung dieser Anlagen passiert aktuell meist von Hand. Dies soll in Zukunft von Robotern und Fließbändern durchgeführt werden.

## 1.2 Problemstellung

Die anfängliche Problemstellung besteht in der Findung von geeigneten Datenmodellen und Kommunikationsprotokollen, um die gewünschten Anlagenkomponenten (Roboter, Steuerung, usw.) zu vernetzen und deren Parameter bauteilabhängig möglichst automatisch auszuwählen. In weiterer Folge sollen darauf aufbauend, Automatisierungsabläufe für die Chargierung von Plasmanitrieranlagen abgeleitet werden. Im Fokus steht die Parkettierung der Anlage und die für die Einbindung nötige Softwarearchitektur.

## 1.3 Forschungsfrage

Aus der vorgestellten Problematik lassen sich folgende Fragestellungen ableiten:

- Wie könnte ein Interaktionsschema für die Chargierung von Plasmanitrieranlagen aussehen und wie kann ein solches System möglichst wiederverwendungs-freundlich gestaltet werden?

## 1 Einleitung

---

- Wie könnten Datenmodelle für die algorithmisch optimierte Chargierung der Anlage aussehen?

## 1.4 Ziel und Inhalt dieser Arbeit

Der Schwerpunkt dieser Arbeit liegt im Erstellen eines Systems zur voll-autonomen Parkettierung der Chargiereneben. Hierfür wird unter anderem die Technologie der künstlichen Intelligenz (KI) herangezogen und die Parkettierung als Optimierungsproblem behandelt. Des Weiteren wird eine Möglichkeit entwickelt, diese Technologie in einen Prozess einzubinden. Konkret können folgende Ziele und Nicht-Ziele definiert werden:

**Tabelle 1.1:** Ziele und Nicht-Ziele

Ziele	Nicht-Ziel
Prozesskonfiguration entwickeln	vollständige Konfigurierbarkeit
Chargieren von Bauteilklassen	Einteilung von Bauteilen in Klassen
Prozesssteuerungsschema entwickeln	Abbildung aller Prozessabläufe
Interaktionsschema für Benutzer entwickeln	Benutzeranalysen
Integration in eine mobile Trainingszelle	Integration in Versuchsanlagen

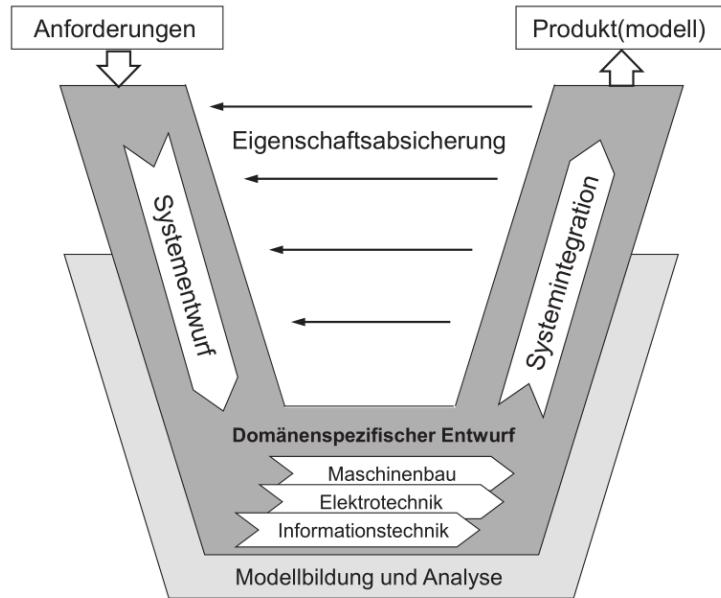
## 1.5 Umsetzungsstrategie

Um einen geeigneten Bereich für die Bearbeitung des Themeninhaltes zu finden, wird die Arbeit in vier Kategorien unterteilt. Zu Beginn wurden in diesem Kapitel die Anforderungen konkretisiert und Forschungsfragen sowie Ziele und Nicht-Ziele definiert. Darauffolgend wird im anschließenden Kapitel eine technische Recherche durchgeführt, um den Stand der Technik und Möglichkeiten offenzulegen. Anschließend werden die gewonnenen Informationen in einer Konzeptentwicklung zusammengefasst. Als Validierung wird zuletzt das vorgestellte Konzept als Prototyp implementiert, der die zuvor behandelten Konzeptmerkmale berücksichtigt. Die Struktur der Arbeit orientiert sich weitgehend am V-Modell (Abb. 1.1) [1], welches sich an der Verein Deutscher Ingenieure (VDI)-Richtlinie 2206 orientiert. Zu Beginn wurden, wie in der Abbildung gezeigt, die Anforderungen an das Projekt definiert. Anschließend widmet

## 1 Einleitung

---

man sich der Grundlagenermittlung, mithilfe dessen der Systementwurf aufgebaut und ein Prototyp erstellt wird.



**Abbildung 1.1:** V-Modell [1]

## 1.6 Voraussetzungen

Es wurde darauf geachtet, die Inhalte zwar kurz aber umfangreich zu behandeln, um dem Leser einen guten Überblick über die Sachverhalte zu vermitteln. Dennoch kann nicht alles in die Arbeit einfließen und setzt aus diesem Grund Basiswissen aus den folgenden Bereichen voraus:

- Programmieren: Syntax in PYTHON.
- Automatisierung: Aufgaben einer Steuerprogrammierbaren Steuerung (SPS).
- Mathematik: Matrizen- und Vektorrechnung, Statistik- und Wahrscheinlichkeitstheorie, Algebra, Differenzialgleichungen und Numerik.

## 2 Recherche

In diesem Kapitel wird eine Recherche zur bestehenden Problematik durchgeführt. Begonnen wird mit der klassischen Automatisierungspyramide, gefolgt von einem Standard für die Programmierung von Steuerungen: International Electrotechnical Commission (IEC) 61131-3. Anschließend, nachdem die Steuerungsseite betrachtet wurde, wird ein Protokoll für die Kommunikation mit diesen Steuerungen vorgestellt, nämlich Open-Platform-Communication Unified-Architecture (OPC UA). Danach wird eine Modellierungssprache für die Beschreibung von Softwarearchitekturen präsentiert, anhand derer die eigenen Programmroutine schematisiert werden. Darauf folgend wird die geometrische Parkettierung vorgestellt, aus welcher Konzepte für den Chargierungsprozess abgeleitet werden. Zu guter Letzt wird noch der Bereich Maschine Learning (ML) aufgearbeitet, wobei ein Großteil für die Realisierung der Eigenidee Verwendung findet.

### 2.1 Automatisierungspyramide

Für die Veranschaulichung industrieller Automatisierung wurde die Automatisierungspyramide [2] (Abb. 2.1) entwickelt. Sie zeigt die wesentlichen Ebenen in einem Unternehmen wieder, diese unterscheiden sich hinsichtlich Geschwindigkeit und der Menge von zu übertragender Daten. Man unterscheidet folgende Hierarchieebenen:

- **Unternehmensleitebene:** Diese Ebene dient der Führung des Unternehmens hinsichtlich Marketing, Personal-, Produkt- und Investitionsplanung.
- **Betriebsleitebene:** In dieser Ebene sind die Produktionsplanung, Terminüberwachung und Kostenanalyse beheimatet. Außerdem werden die Verwaltung und Bearbeitung von Lieferaufträgen hier durchgeführt.
- **Produktionsleitebene:** Sie beschäftigt sich mit der kurzfristigen Produktionsplanung wie beispielsweise der Einsatzplanung von Maschinen und Personal.
- **Prozessleitebene:** Auch Zellebene genannt, die Prozessleitebene dient zur Steuerung bzw. Regelung von Produktionsprozessen. Der Begriff Zellebene ergibt sich daher, dass die Prozesse oft in abgegrenzten Fertigungsbereichen laufen.
- **Feldebene:** Hier werden Daten, oft in Echtzeit, erfasst und entsprechend einer Anweisung reagiert. Des Weiteren werden in dieser Ebene die Daten aufbereitet und dem übergeordneten System zur Verfügung gestellt.

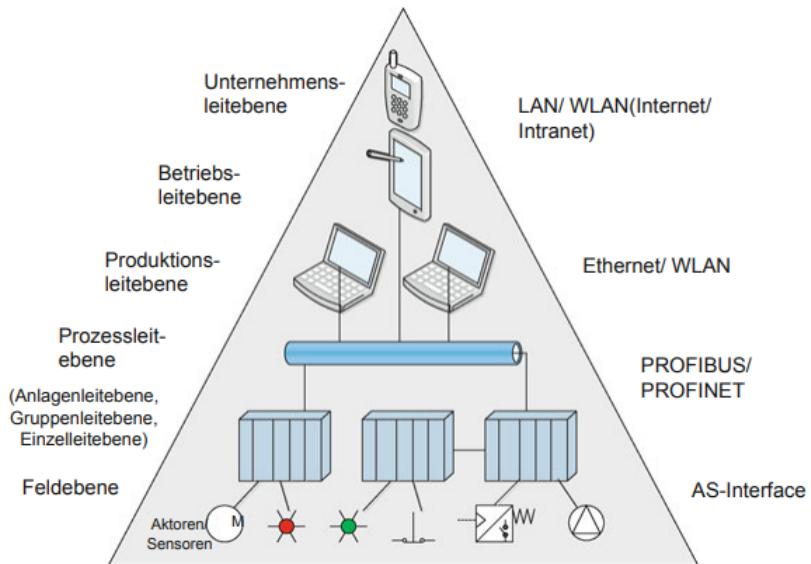


Abbildung 2.1: Automatisierungspyramide [2]

## 2.2 IEC 61131-3

Automatisierungslösungen beinhalten in der Industrie meist SPSen, die für die Abarbeitung von Steuerlogiken verantwortlich sind. Dazu gehört zudem nicht nur die Abarbeitung der Logik, sondern auch das Ansprechen von Relais und dem Übertragen von Daten. Da in den Anfängen die Steuerungshersteller unterschiedliche Programmiersprachen einführten, wurde der Schrei nach einem Standard größer, um die dadurch entstandenen Inkompatibilitäts-Probleme zu beheben. Dieser Standard konnte mit der Norm IEC 61131-3 [3] eingeführt werden, die mittlerweile sämtliche Steuerungshersteller in ihren Steuerungen meist vollständig implementiert haben.

### 2.2.1 Projektstruktur

Um die SPS-Projekte von der Struktur her zu standardisieren, führte die IEC unter der Norm-Nummer 61131-3 eine Hierarchie ein. Die Hierarchie baut sich über Konfigurationselemente und der Programm-Organisationseinheit (POE) auf. Alle in Abb. 2.2 dargestellten Elemente können mehrere Unterelemente enthalten.

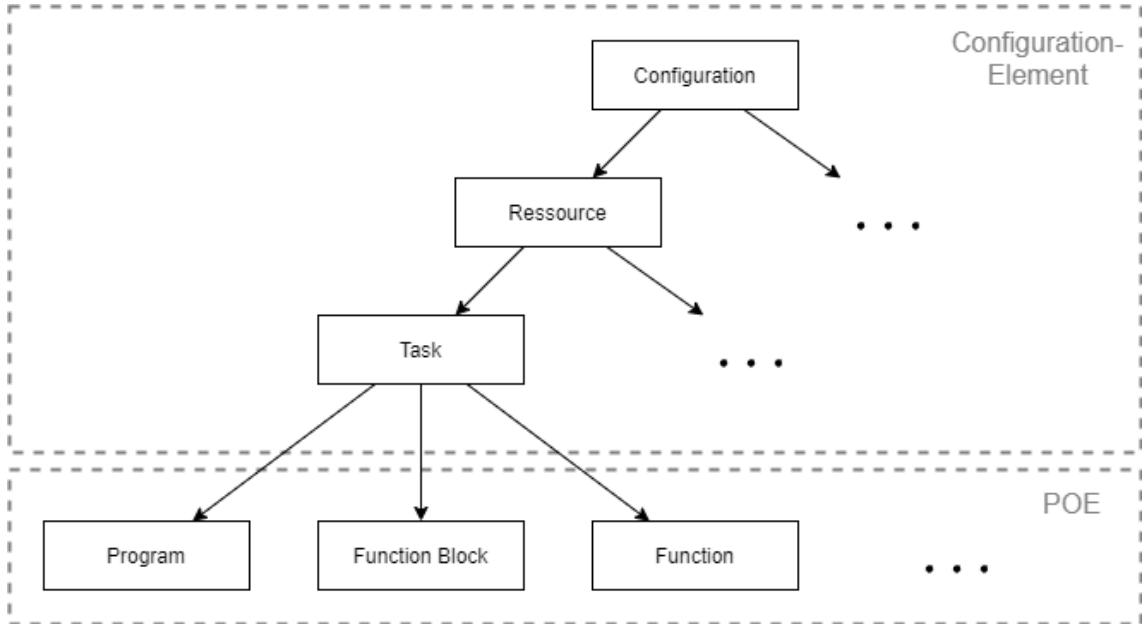


Abbildung 2.2: Projektstruktur nach IEC 61131-3

### 2.2.2 Konfigurationselemente

Die Konfigurationselemente sind wie in Abb. 2.2 dargestellt Konfiguration, Ressource und Task. Diese Elemente beinhalten selbst keinen Code, sondern dienen ausschließlich der Konfiguration der Steuerung. Meistens findet sich hierfür eine Einstellungsseite in der Entwicklungsumgebung der Hersteller.

- **Konfiguration:** Die Konfiguration definiert die zur Verfügung stehenden Ressourcen wie z. B. Busmodule und I/O-Karten, aber auch globale Variablen können hier angelegt werden.
- **Ressource:** Sie stehen für die Recheneinheiten der SPS der Central Processing Unit (CPU). Für die einzelnen Ressourcen können wieder Variablen deklariert werden, die in den darunterliegenden Komponenten zur Verfügung stehen.
- **Task:** Ein Task definiert die Laufzeiteigenschaft von Programmen. Ein Programm, geschrieben in den unterschiedlichen Programmiersprachen, kann zyklisch oder bei Erfüllung bestimmter Ereignisse abgerufen und durchlaufen werden. Dies und die Zuteilung einer Priorität für die Programmausführung können in einem Task deklariert werden.

### 2.2.3 Programmorganisationseinheit

Die POE bildet die eigentliche Logik des SPS-Programms ab. In einer POE können wieder Variablen angelegt werden, die nun jedoch nur innerhalb des Programms abgerufen werden können. Neben den Variablen stehen dem Benutzer auch Programme und Funktionsblöcke zum Aufruf bereit, falls diese in einer Bibliothek angelegt wurden. Hier gilt zu Beachten, dass Funktionsblöcke nicht von Funktionen aufgerufen werden können, sondern nur von anderen, vorgesetzten Funktionsblöcken. Selbst können sie jedoch Funktionen aufrufen. Im Unterschied zum Funktionsblock kann eine Funktion nur eine Variable zurückgeben und besitzt keinen eigenen Speicherbereich. Der Speicherbereich ermöglicht einem Funktionsblock das Speichern von Variablen, womit dem Funktionsblock die Fähigkeit verliehen wird, sich Zustände zu merken.

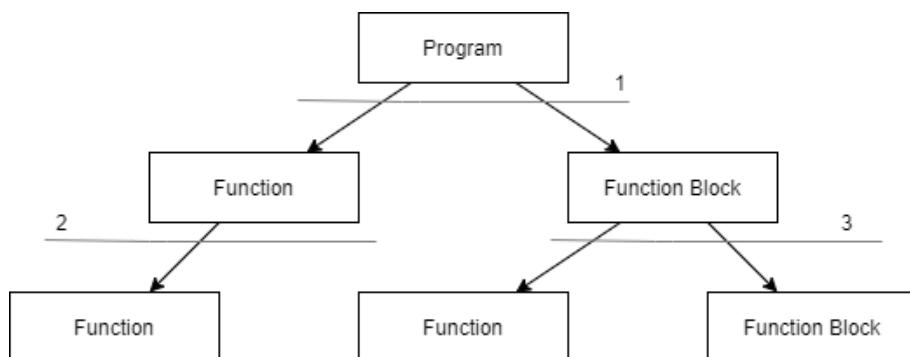


Abbildung 2.3: POE-Struktur

- 1: Das Programm ruft Funktion oder Funktionsblock auf.
- 2: Funktion ruft Funktion auf.
- 3: Funktionsblock ruft Funktion oder Funktionsblock auf.

### 2.2.4 Variablen und Datentypen

Die IEC 61131-3 definiert gängige Standard-Datentypen wie z.Bsp. Bool, Byte oder Integer. Des Weiteren können jedoch auch eigene Datentypen definiert und verwendet werden, wie:

- Strukturtypen
- Aufzählungstypen
- Aliastypen

Die Definition eines Datentyps kann innerhalb der Konfiguration oder innerhalb einer Bibliothek erfolgen. Abbildung 2.4 zeigt die Definition der Datentypen innerhalb der Konfiguration der Programmierumgebung CODESYS.

<pre> 1  TYPE TestStructure :      1  TYPE TestEnumeration :      1  TYPE TestAlias : INT; 2  STRUCT                      2  (                                2  END_TYPE 3    a:INT;                     3    member1 := 0,          3 4    b:BOOL;                    4    member2 := 5,          4 5    c:REAL;                    5    member3 := 10          5 6  END_STRUCT                  6  );                            6 7  END_TYPE                    7  END_TYPE                   7 8 </pre>	<pre> (a) Struct </pre>	<pre> (b) Enumeration </pre>	<pre> (c) Alias </pre>
---	-------------------------	------------------------------	------------------------

**Abbildung 2.4:** Definition der Datentypen

Variablen werden entweder in der Konfiguration, bei der Ressourcen-Definition oder innerhalb einer POE definiert. Bei der Definition selbst kann neben dem Variablen-Namen und des Variablen-Typs auch ein Initial-Wert festgelegt werden. Das Zugriffsverhalten einer Variable hängt dabei vom Ort ab, in welchem die Variable angelegt wurde.

## 2.2.5 Programmiersprachen

Die IEC 61131-3 definiert 5 Programmiersprachen. Diese können verwendet werden, um die benötigte Logik abzubilden. Manche davon sind abstrakter als andere, bzw. bieten sie die Möglichkeit, Programm Routinen verschiedener Komplexitätsstufen einfach oder weniger einfach abzubilden. Um einen kurzen Einblick zu gewähren, werden vier der Programmiersprachen anhand einer Motoransteuerung erläutert. Die Steuerung schaltet ein Motor-Relais, falls ein Knopf betätigt wurde und der Endschalter nicht erreicht ist. Außerdem wird mit einem Schalter geprüft, ob das Relais bereits geschaltet wurde (Abb. 2.5).

- **Kontaktplan:** Für das Lesen dieses Programmcodes werden keine Programmierkenntnisse benötigt, da sie die Logik rein grafisch darstellt. Der Aufbau erinnert an ein Flussdiagramm aus der Elektrotechnik. Die Programmierobjekte sind einfache, steuerungstechnische Komponenten wie Relais und Endschalter. Diese werden seriell oder parallel verbunden, wobei sie im Programmcode von links nach rechts abgearbeitet werden. Im Beispiel wird also das Motor-Relais geschaltet, falls der Knopf ‚Button‘ getätigigt wird oder der Schalter ‚%IX23.5‘ bereits aktiv ist. Außerdem muss der Schließer aktiv sein, also die Variable ‚EmStop‘, ein Null-Signal liefern.

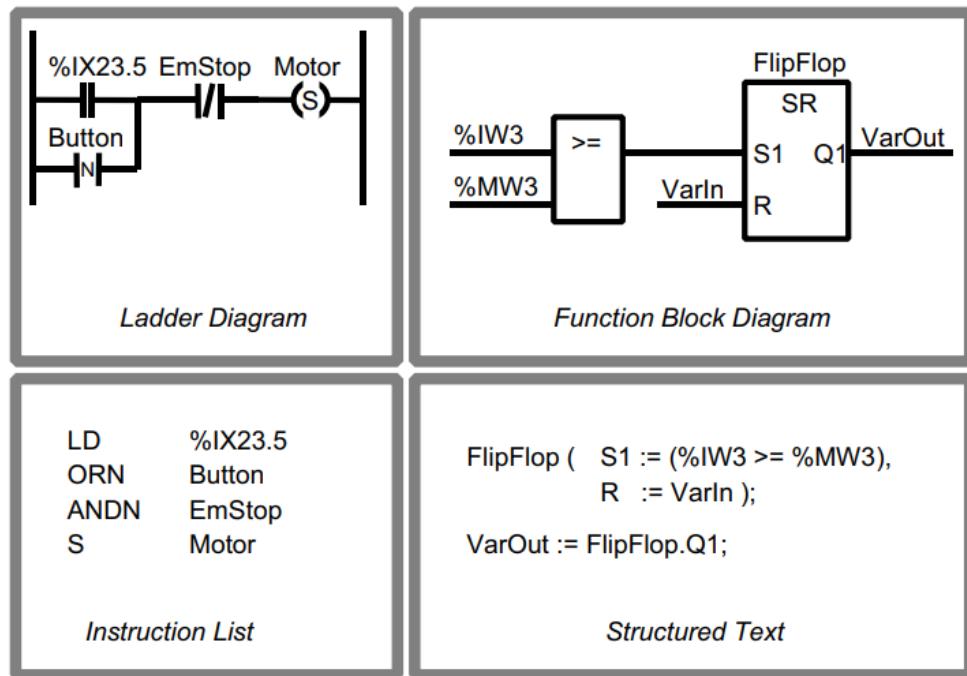


Abbildung 2.5: Programmiersprachen der IEC 61131-3 [3]

- **Funktionsblocksteinsprache:** Sie besteht wie der Name vermuten lässt, aus Funktionsblöcken. Ein Funktionsblock bildet meist reale Objekte ab. Ähnlich wie Klassen in der objektorientierten Programmierung, versucht der Funktionsblock die Eigenschaften von Objekten genau abzubilden. Funktionsblöcke besitzen einen eigenen Speicherbereich und können somit von vorhergehenden Ereignissen beeinflusst werden. Die Motorsteuerung wird mit zwei Blöcken realisiert, zunächst die Schwellwert-Prüfung mittels eines Größer-Gleich-Blocks und anschließend die Schaltung des Relais mittels eines Flip-Flops. Das Flip-Flop merkt sich, wenn am Eingang „S1“ ein Impuls ankommt und schaltet so lange durch, bis auch am Eingang „R“ ein Impuls wirkt. Als Anmerkung sei noch erwähnt, dass Funktionsblöcke selbst wieder mit unterschiedlichen Programmiersprachen aufgebaut werden können.
- **Anweisungsliste:** Ist eine sehr maschinennahe Programmiersprache und orientiert sich an ASSEMBLER. Es werden Variablen aus dem Arbeitsspeicher in den Cache-Speicher geladen und anschließend Operationen daran durchgeführt. Sind alle gewünschten Operationen durchgeführt, wird der aktuelle Wert im Cache wieder auf den Arbeitsspeicher übertragen oder gelöscht. Im Beispiel wird zunächst die Variable „%IX23.5“ in den Cache geladen. Anschließend wird eine „Oder“- und „Und“-Operation am Cache mit den Variablen „Button“ und

,EmStop‘ durchgeführt. Das Ergebnis dieser Operationen wird auf die Variable „Motor‘ übertragen.

- **Strukturierter Text:** Die wohl vielseitigste Programmiersprache der IEC-Norm. Sie lehnt sich stark an der Programmiersprache C bzw. PASCAL an und ermöglicht somit noch relativ übersichtlich komplexe Logiken darzustellen, zumindest für Anwender, die den Umgang mit Hochsprachen gewöhnt sind. In dieser Programmiersprache werden gerne Funktionsblöcke verwendet, um den Code lesbarer zu machen. Im Beispiel wird ein Objekt des Datentyps ,FlipFlop‘ angelegt und mit den Werten von ,%IW3‘, ,%MW3‘ und ,VarIn‘ initialisiert. Besonders kompakt kann die Bedingung für die Schwellwert-Überschreitung mit dem Größer-Gleich-Operator angeschrieben werden, die anschließend der internen Variablen ,S1‘ des Flip-Flops übergeben wird.
- **Ablausprache:** Diese Sprache besteht aus Schritte und Transitionen. Die Schritte beinhalten den Programmablauf, welcher durch andere Programmiersprachen aus-implementiert werden kann. Ist eine Transitionsbedingung erfüllt, wird in den jeweiligen Schritt gewechselt und der dahinterliegende Programmablauf durchgeführt.

## 2.3 OPC Unified-Architecture

OPC UA [4] [5] [6] ist eine Service orientierte Softwarearchitektur. Die standardisierte und offene Struktur ermöglicht einen vom SPS-Hersteller unabhängigen Austausch von Daten. Zumeist muss die Kompatibilität jedoch kostenpflichtig freigeschaltet werden. OPC UA beschäftigt sich überwiegend mit der Datenmodellierung und dem Daten-Transportmechanismus. Der Datenaustausch erfolgt über ein adaptiertes Standardprotokoll aus der Netzwerktechnik, dem Transmission Control Protocol (TCP), welches die Art und Weise des Datenaustausches festlegt. UA TCP ist für schnellen binären Datenaustausch geeignet, für verschlüsselten Datenaustausch verwendet OPC UA das XML-Format, welches über Hypertext Transfer Protocol Secure (HTTPS)-Protokolle übermittelt wird (Kap. 2.3.6). Um eine solche Datenverbindung aufzubauen, benötigt der Server eine im Netzwerk gültige Adresse und einen Port, aus denen sich ein Uniform Resource Locator (URL) zusammensetzt.

### 2.3.1 Adressraum

Der Datenaustausch erfolgt mit sogenannten *nodes*, welche als Datenmodelle zu verstehen sind. Des Weiteren können *nodes* auch als Adressblock im Speicher interpretiert werden, welche *node*-Attribute, Daten und Referenzen zu anderen *nodes* enthalten. Ein wichtiger Teil der *node*-Attribute stellt der *node*-Identifikator (ID) dar. Er besteht aus 3 Komponenten, die die Lokalisierung im Speicher erst ermöglichen. Die drei Komponenten sind: *namespace*-Uniform Resource Identifier (URI), Datentyp des *identifier* und der *identifier* selbst. Hierzu ein kurzes Bsp.:

**Programm 2.1:** OPC UA Node-ID

```
1 ns=http://example.org:80/some/directory;string=Temperature
```

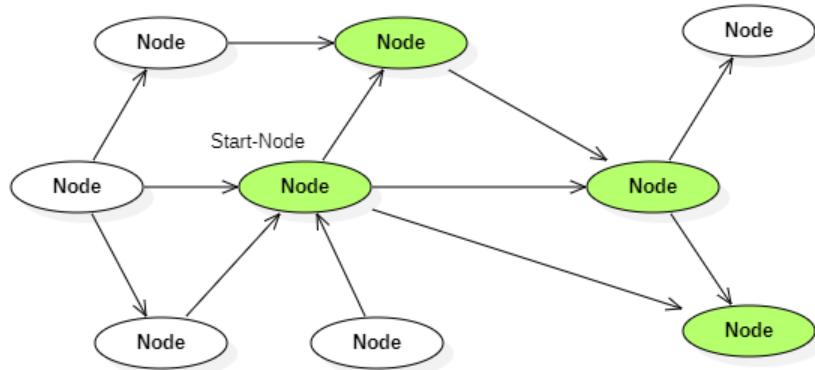
Die *node*-ID kann jedoch noch weiter verkürzt werden, nämlich indem man eine *lookup table* einführt (Tab. 2.1). Somit reduziert sich der *namespace* zu einem Integer-Wert.

**Tabelle 2.1:** Abkürzung der *namespace*-URI mittels *lookup table*

Index	Namespace-URI
0	http://opcfoundation.org:80/UA
1	http://example.org:80/some/directory
...	...

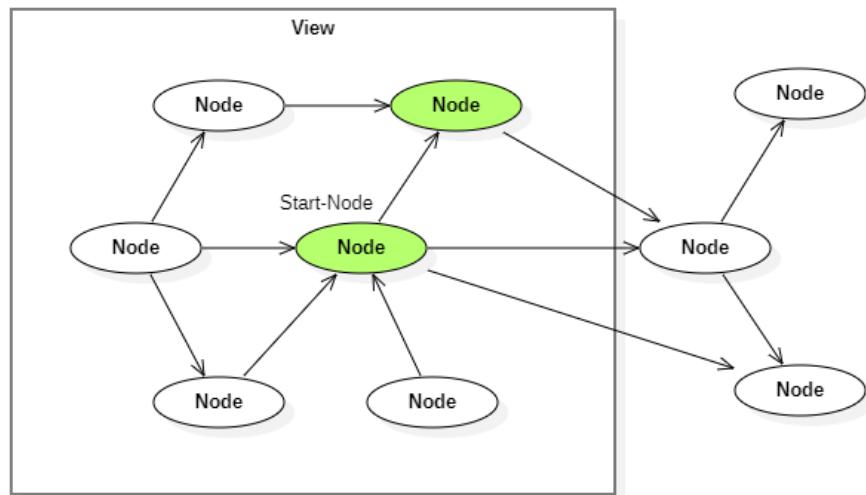
### 2.3.2 Browsing

Um den Adressraum zu durchsuchen, besteht für den Client die Möglichkeit eine Browsing-Anfrage an den OPC UA-Server zu stellen. Dabei wird dem Server ein Ausgangs-*node*/ Start-*node* übermittelt, der wiederum alle Referenzen dieses *nodes* zurückgibt. Der Prozess wird in Abb. 2.6 grafisch veranschaulicht, die grün markierten *nodes* werden vom Server zurückgegeben und die Pfeile repräsentieren die jeweiligen Referenzen.



**Abbildung 2.6:** Browsing im Adressraum

Zusätzlich besteht die Möglichkeit zur Verwendung von *views*, mithilfe derer der Adressraum in Unterräume unterteilt werden kann. Wird nun eine Suchanfrage an den Server gesendet, werden nur jene Referenzen des Ausgangs-*node* zurückgegeben, die sich in diesem Unterraum befinden (Abb. 2.7).



**Abbildung 2.7:** Browsing im Adressraum mit einer *view*

### 2.3.3 Subscription

Soll der Client Wertänderungen mitbekommen, ohne den Wert zyklisch abrufen zu müssen, so bietet OPC UA die Möglichkeit eine *subscription* anzulegen. Die *subscription* besteht aus ein oder mehreren *monitored items*, die eine Referenz zum überwachenden *node*-Attribut aufweisen. Zusätzlich kann eine Puffergröße und ein

## 2 Recherche

Abtastintervall festgelegt werden. Bei einer Wertänderung wird der Client vom Server benachrichtigt. Das Wahrnehmen einer Wertänderung erfolgt serverseitig mittels eines Filters, der festlegt, wann ein Wert als verändert zu betrachten ist. Um zu bestimmten Zeitpunkten eine Benachrichtigung vom Server zu erhalten, kann auch ein *publish*-Intervall festgelegt werden. Dieser beschreibt die minimale Zeitdifferenz, die zwischen zwei Nachrichten verstreichen darf. Dies kann dazu verwendet werden, um die Funktionsfähigkeit der *subscription* nachzuweisen.

### 2.3.4 Node-Modell

Um die Datenmodelle standardisiert aufzubauen, wurde von der OPC UA Foundation ein *node*-Modell entworfen (2.8). Dieses Modell besagt, dass alle *nodes* von einer *base-node-class* abgeleitet werden, die alle grundlegenden Attribute einer *node* besitzen. Diese grundlegenden Attribute sind: ‚NodeID‘, ‚NodeClass‘, ‚BrowsingName‘ und ‚DisplayName‘. Des Weiteren sieht das Modell auch die Verwendung von einigen *node*-Standardtypen vor, wie ‚Object‘, ‚Variable‘ und ‚Method‘.

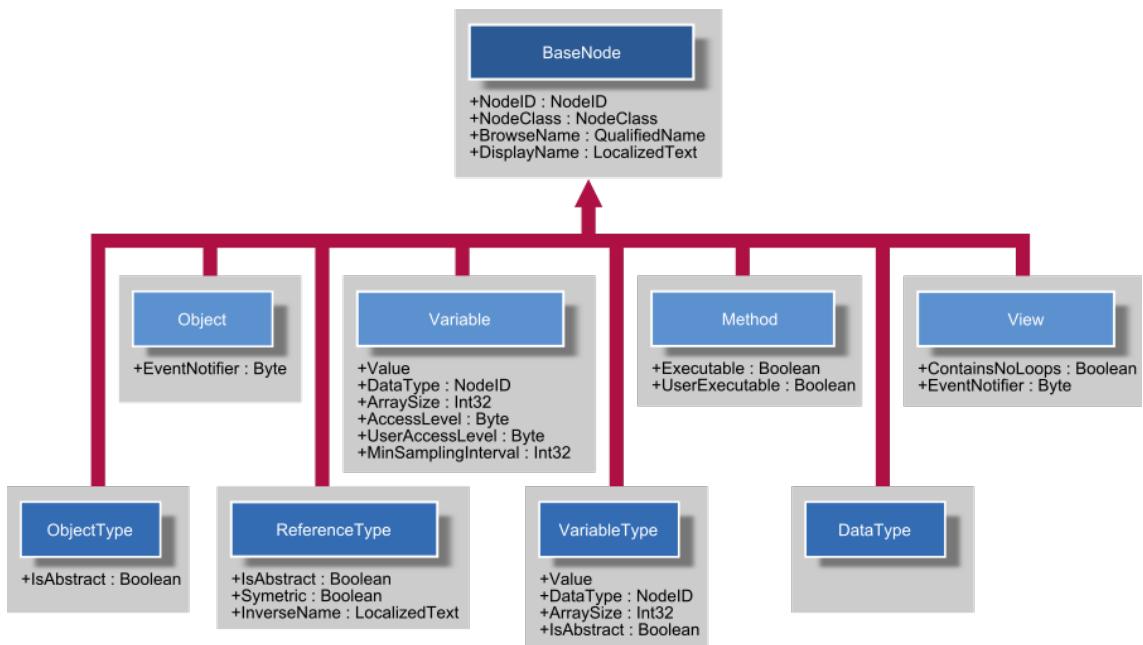


Abbildung 2.8: *Node*-Modell von OPC UA [5]

### 2.3.5 Nodeart

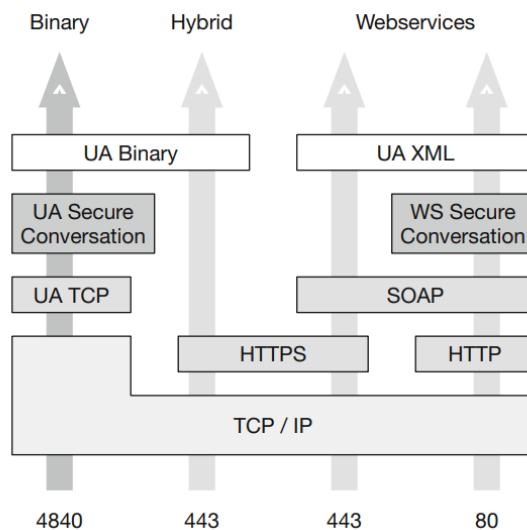
Sind die *nodes* definiert und implementiert, spricht man von einem OPC UA Basis-Informationsmodell. In dem Modell befinden sich alle benötigten Informationen, um den Prozess abzubilden. Um nun mit dem Modell zu interagieren, werden die vom

Server zur Verfügung gestellten *nodes* noch zu Gruppen zusammengefasst. Für diese Gruppierung werden von OPC UA folgende Deklarationen vorgesehen:

- OPC DA (Data-Access): Echtzeitlesen von Variablen, für Funktionen wie `read()`, `write()`, `browse()` oder `subscribe()`
- OPC HA (Historical-Access): Lesen von bereits gespeicherten Daten, für Funktionen wie `analyse()`, `collect()`
- OPC A&E (Alarm & Events): Benachrichtigungen bei Alarmen und Events, für Funktionen wie `receive()`, `filter()`
- OPC Prog (Programs): Ausführen eines gespeicherten Ablaufs

### 2.3.6 Kommunikationswege

OPC UA setzt, wie Eingangs erwähnt, bei der Kommunikation auf das TCP-Protokoll. Abbildung 2.9 zeigt die unterschiedlichen Kommunikationswege, die wiederum auf bereits etablierte Protokolle, wie Hypertext Transfer Protocol (HTTP) und HTTPS, aufsetzten. Der linke Zweig ‚UA Binary‘ wird als besonders sicher und effizient beworben [4].



**Abbildung 2.9:** OPC UA-Protokolle und mögliche Kommunikationswege [4]

Bei der richtigen Verwendung bietet OPC UA, verglichen mit anderen Industrieprotokollen, ein hohes Maß an Sicherheit [7]. Neben der klassischen Benutzername-Passwort-Kombination, werden auch weitere Authentifizierungsmethoden wie Web-Tokens und Zertifikate zur Verfügung gestellt. Die verwendeten Verschlüsselungsmethoden sind:

- Basic-128/RSA-15
- Basic-256
- Basic-256/SHA-256
- AES-128/SHA-256 (RSA-OAEP)
- AES-256/SHA-256 (RSA-PSS)

### 2.3.7 Funktions- und Programmaufruf

*Methods* werden dazu verwendet, um Funktionsaufrufe auf dem Server auszuführen. Der Client erhält dabei Informationen über die Übergabe- bzw. den Rückgabeparameter und deren Datentypen. Tätigt der Client eine Anfrage zur Ausführung einer *method* mit den benötigten Übergabeparameter, arbeitet der Server die entsprechenden Prozesse ab und übermittelt bei Beendigung des Prozesses die Rückgabeparameter an den Client. Aufgrund dieses Ablaufs eignen sich *methods* nur für kurzlebige Prozesse.

Im Unterschied zu den *methods* bilden OPC UA *programs* einen kompletten Zustandsautomaten ab, somit erhält der Client ein höheres Maß an Kontrolle über den Prozess. Dadurch erhält dieser die Möglichkeit, Prozesse zu starten, stoppen oder zu pausieren und wird des Weiteren vom Server über die Zustandsänderungen, sogenannte Transitionen, informiert. Der Client übergibt dabei der Start-*method* die benötigten Übergabeparameter und erhält anschließend vom Server Benachrichtigungen über jede stattgefundene Transition sowie etwaige Zwischenergebnisse. Im laufenden Prozess besteht für den Client die Möglichkeit, mittels anderer *methods* den Prozessablauf zu steuern. Ist der Prozess beendet, kann der Client die Rückgabeparameter abfragen. Im Vergleich zu den *methods*, eignen sich *programs* für langlebige Prozesse, wie sie durch Zustandsautomaten abgebildet werden.

## 2.4 Unified Markup Language

Unified Markup Language (UML) ist eine internationale standardisierte Modellierungssprache und wird von einer internationalen Organisation verwaltet, die Object Management Group (OMG). Mittels UML kann ein Softwareentwurf stattfinden, um ein möglichst klares Bild von dessen Funktionalität zu vermitteln. Sämtliche Erläuterungen dieses Kapitels wurden [8] entnommen und mithilfe eigener Beispiele ausgearbeitet.

### 2.4.1 Klasse

Eine Klasse dient der Beschreibung von Objekten, die bei der Funktion der Anwendung mitwirken. Die Darstellung einer Klasse erfolgt durch ein Rechteck, welches den Namen der Klasse enthält. Darunterliegend befindet sich ein weiteres, gleich-breites Rechteck für die Angabe von Attributen. Des Weiteren kann in der Darstellung ein weiteres Rechteck mit Angaben zu Operationen angrenzen. Abb. 2.10 zeigt diese Darstellung anhand einiger Beispiele, jedoch ohne Attribute oder Operationen.



Abbildung 2.10: Darstellung von Klassen in UML

### Objekt

Objekte leiten sich aus der Verwendung von Klassen ab. Zum Beispiel könnte ein Objekt mit dem Namen ‚Kugellager‘ durch die Klasse ‚Bauteil‘ instanziert werden. Es kann beliebig viele Instanzen einer Klasse geben.

### 2.4.2 Attribut

Attribute sind Merkmale einer Klasse, die aus den Objekten einer Klasse abgeleitet werden. Zum Beispiel hat das Objekt ‚Kugellager‘ eine Masse, diese wird als Attribut der Klasse angelegt und bei der Instanzierung mit einem Wert belegt. Attribute benötigen eine Bezeichnung und einen VariablenTyp. Falls nicht weiter angegeben, handelt es sich bei den Attributen immer um sogenannte Pflichtfelder. Ist dies nicht gewünscht, wird dies durch Setzen von [0..1] hinter den Attributtypen gekennzeichnet. Des Weiteren können noch Aufzählungen (*enumerations*) verwendet werden, dies sind Listen mit allen möglichen Werten, die das Attribut annehmen kann. Neben dem VariablenTyp wird noch eine Information für die Vererbung (*inheritance*) benötigt. Dies dient der Zugriffsregulierung auf Variablen, nachdem ein Objekt angelegt wurde. Diese Information lässt sich in UML durch folgende 3 Zeichen vor dem VariablenTyp darstellen:

# protected

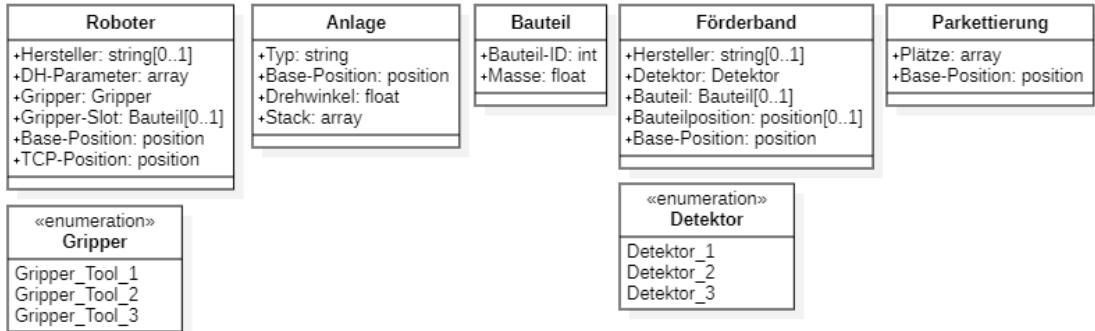
+ public

- private

## 2 Recherche

---

Abbildung 2.11 zeigt einige dieser Darstellungsweisen für Attribute. An dieser Stelle sei erwähnt, dass Python keine Methode zur Zugriffsregulierung bietet und daher alle Attribute als *public* gekennzeichnet sind.



**Abbildung 2.11:** Darstellung von Klassen-Attributen in UML

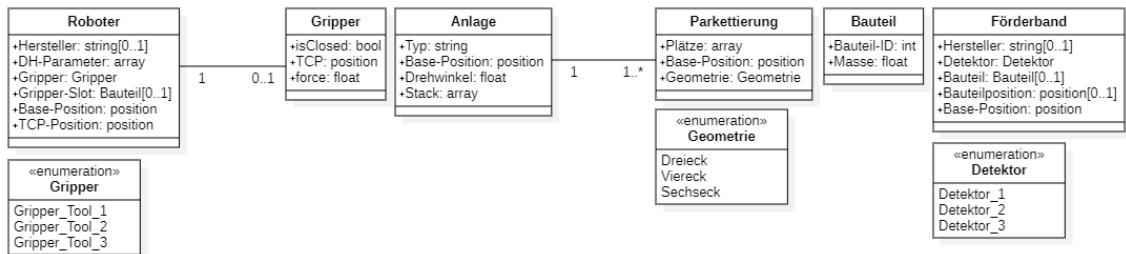
### 2.4.3 Assoziation

Die Assoziation zwischen Klassen wird mit einer verbindenden Linie gekennzeichnet. Zusammen mit den Attributen einer Klasse bildet die Assoziation die Eigenschaften von Objekten vollständig nach.

- **Multiplizität** ist eine Kennzeichnung der Assoziation. Sie gibt an, mit wie vielen Objekten einer Klasse eine andere verbunden werden darf. Die Angabe der Anzahl erfolgt am jeweiligen Ende der Linie zur Klasse, an der sie anliegt. Mögliche Multiplizitäten sind in der Tabelle 2.2 angegeben, jedoch können auch eigene Multiplizitäten definiert werden. Eine Möglichkeit wäre, dass ein Objekt der Klasse ‚Roboter‘ mit oder ohne der Klasse ‚Gripper‘ initialisiert wird, da während eines möglichen Greiferwechsels ein Ausfall der Anlage denkbar ist und beim Wiederanfahren der Anlage der Roboter nicht zwingend einen Greifer besitzt. Des Weiteren kann der Roboter im Normalfall, außer es handelt sich um mehrere Greifer auf einer Montageplatte, nur einen Greifer montiert haben. Im Vergleich dazu muss im Zuge dieses Projekts ein Objekt der Klasse ‚Anlage‘ auch mit einem oder mehreren Objekten der Klasse ‚Parkettierung‘ in Verbindung stehen, da man nicht von einer echten Anlage sprechen kann, wenn keine Parkettierung vorliegt (Abb. 2.12).

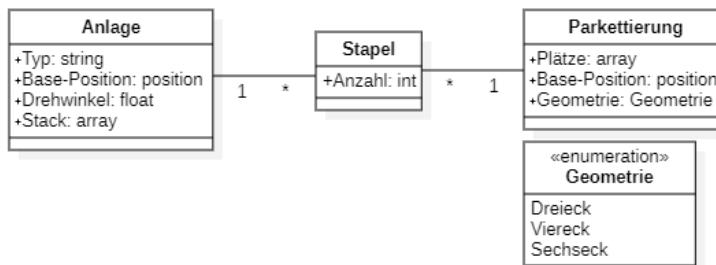
## 2 Recherche

---



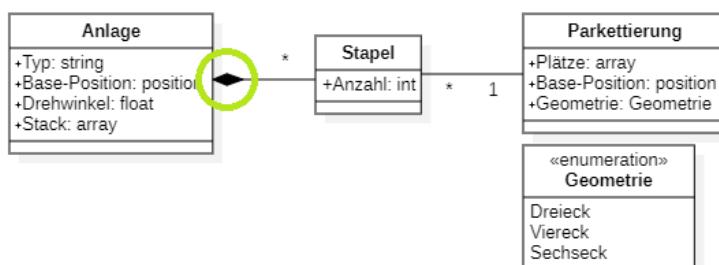
**Abbildung 2.12:** Darstellung von Assoziationen in UML

- Eine **Zwischenklasse** umfasst zusätzliche Information über die Verbindung zweier Klassen. Zum Beispiel könnte über eine Zwischenklasse ‚Stapel‘ die Information ausgedrückt werden, wie viele Parkettierungen gleicher Art eine Anlage aufnimmt (Abb. 2.13).



**Abbildung 2.13:** Darstellung einer Zwischenklasse in UML

- **Komposition** wird bei einer Teil-Ganzes-Beziehung (*part-whole association*) gebraucht. Sie legt zum einen fest, dass die Multiplizität auf der Seite des Ganzen Eins ist und zum anderen, dass bei der Löschung der ‚Ganz‘-Seite auch die durch die Komposition verbundene Seite gelöscht wird. Die Komposition kennzeichnet also eine Löschregel (Abb. 2.14).

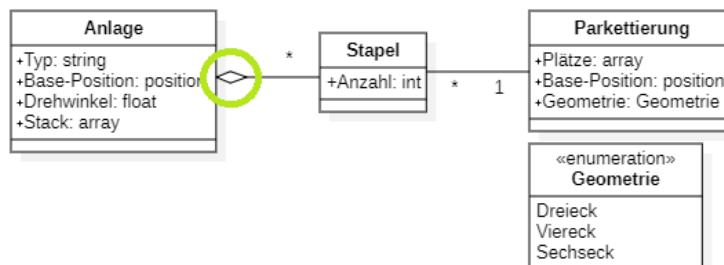


**Abbildung 2.14:** Darstellung einer Komposition in UML

## 2 Recherche

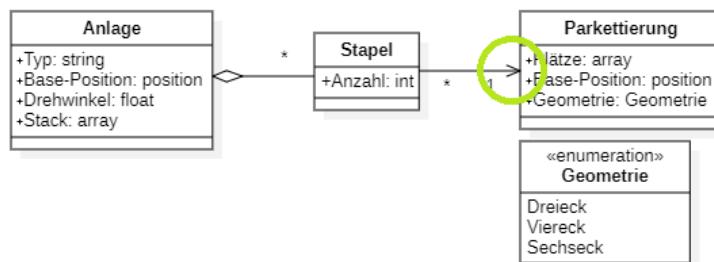
---

- **Aggregation** ist für Löschroutinen interessant und wird als nicht ausgefüllte Raute dargestellt (Abb. 2.15). Sie bietet die Möglichkeit bei einer Löschung von verbundenen Klassen, nochmals eine Interaktion mit dem Nutzer zu starten. So könnte dem Nutzer die Möglichkeit eröffnet werden, neben dem ‚Ganzen‘ auch den ‚Teil‘, oder nichts zu löschen. Man könnte also die Aggregation als Sonderfall der Komposition interpretieren.



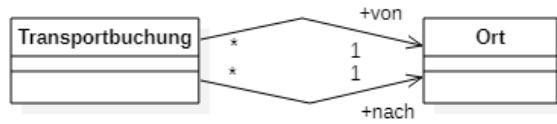
**Abbildung 2.15:** Darstellung einer Aggregation in UML

- **Navigation** dient zur Veranschaulichung der Navigierbarkeit zwischen Klassen. Als Beispiel könnte man die Navigierbarkeit von der zuvor verwendeten Zwischenklasse ‚Stapel‘ zu der Klasse ‚Parkettierung‘ wie in Abb. 2.16 dargestellt veranschaulichen. In diesem Beispiel ist die Beziehung allein vom Stapel zur Parkettierung navigierbar. Das heißt, wenn der Anwender einen Stapel betrachtet, ist die betroffene Parkettierung sichtbar. Umgekehrt ist dies jedoch nicht der Fall.



**Abbildung 2.16:** Darstellung einer Navigation in UML

- **Assoziationsname** wird benötigt, falls die Bedeutung der Assoziation nicht selbsterklärend ist. Zum Beispiel wäre eine Assoziation von der Klasse Transportbuchung zur Klasse ‚Ort‘ (Abb. 2.17) ohne Assoziationsname nicht eindeutig, da keine Klarheit besteht, ob es sich um einen Transport von Ort oder einen Transport nach Ort handelt.



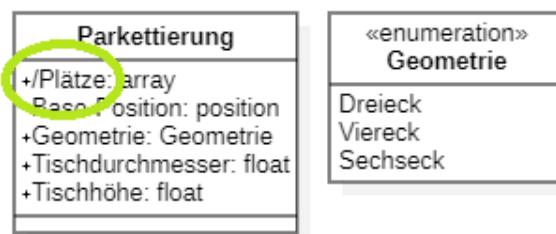
**Abbildung 2.17:** Darstellung von Assoziationsnamen in UML

**Tabelle 2.2:** Möglichkeiten an Multiplizitäten

Multiplizität	Anzahl der verbundenen Objekte
1	Genau 1
0..1	0 oder 1
*	0 oder mehr (Abkürzung für 0..*)
1..*	1 oder mehr

#### 2.4.4 Abgeleitetes Attribut

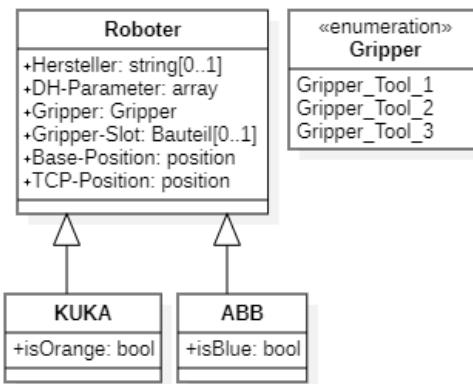
Abgeleitete Attribute werden durch einen ,/‘ vor dem Attributnamen gekennzeichnet. Sie symbolisieren, dass sich das Attribut aus anderen (abgeleiteten) Attributen berechnen lässt. Im Kontext zum Projekt könnte dies bedeuten, dass die Klasse „Parkettierung“ die verfügbaren Plätze aus der Geometrie-Information ableitet und selbstständig berechnet (Abb. 2.18).



**Abbildung 2.18:** Darstellung eines abgeleiteten Attributes in UML

### 2.4.5 Vererbung

Die Vererbung geschieht durch Verwendung einer *superclass*, welche auch als Spezialisierung bezeichnet wird. Sie wird verwendet, wenn diverse Klassen gemeinsame Eigenschaften aufweisen, die nicht jedes mal erneut implementiert werden sollen. Solche *subclasses* werden von der *superclass* abgeleitet und erben deren Attribute. Des Weiteren können anschließend noch weitere Attribute, die nur der abgeleiteten Klasse zugeordnet sind, definiert werden. Die Vererbung wird in UML durch einen dreieckigen Pfeil gekennzeichnet. Abbildung 2.19 zeigt die Ableitung der Klassen ‚KUKA‘ und ‚ABB‘ von der Klasse ‚Roboter‘. Die abgeleiteten Klassen besitzen alle Attribute der *superclass* und zusätzlich noch ein Attribut über deren Farbe.



**Abbildung 2.19:** Darstellung einer Vererbung in UML

### 2.4.6 Vorteile

Durch die Verwendung von UML-Diagrammen kann eine Anforderungsspezifikation sinngemäß wiedergegeben werden. Dadurch wird die Zusammenarbeit mit externen Kräften vereinfacht, sofern diese die UML-Sprache beherrschen. Des Weiteren können die erstellten Architekturen auch für ein Datenbanken-Layout oder ein *node*-Modell von OPC UA herangezogen werden.

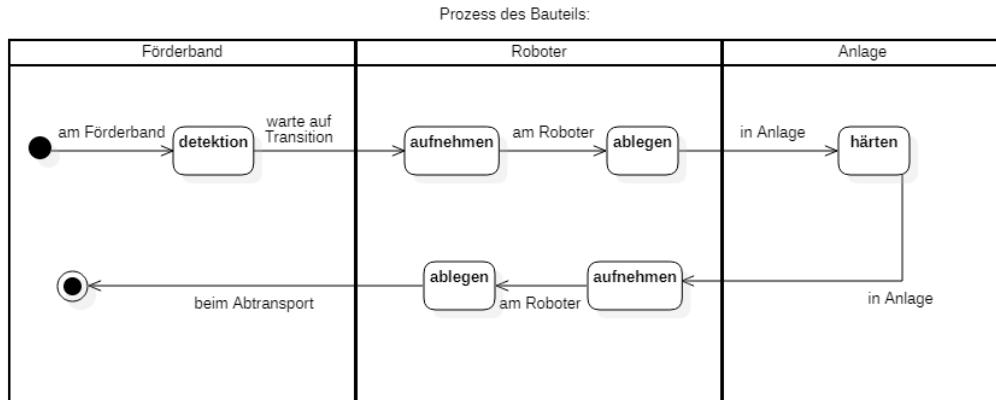
Als Anmerkung sei hier noch erwähnt, dass es als Modellierer abzuwiegeln gilt, ob man Einfachheit oder Flexibilität bieten möchte. Zum Beispiel kann man Einfachheitshalber die vorgestellte *enumeration* wählen, um die verschiedenen Geometrien der Parkettierung zu implementieren. Will man jedoch die Möglichkeit frei lassen, die Eigenschaften der Geometrien nachträglich zu ändern, müssten Referenzklassen verwendet werden.

### 2.4.7 Aktivitätsdiagramm

Das UML-Aktivitätsdiagramm beschreibt den Prozess, den ein Objekt einer Klasse durchläuft. Um den Prozess grafisch zu beschreiben, werden folgende Elemente im Aktivitätsdiagramm verwendet:

- **Aktion:** Gibt an, welche Schritte während des Prozesses ausgeführt werden.
- **Zustand:** Gibt den Status des Objekts an, welches den Prozess durchläuft, nachdem eine vorhergehende Aktion ausgeführt wurde und bevor die nächste Aktion ausgeführt wird.
- **Schwimmbahn:** Zeigt durch ihren Bereich an, welche Klasse die Aktionen ausführt.
- **Startknoten:** Aus dem Startknoten geht der Anfangszustand hervor, den das Objekt bei der Initialisierung annimmt.
- **Endknoten:** Gibt an, welchen Endzustand das Objekt aufweist, wenn der Prozess vollständig durchlaufen wurde und sich der Zustand somit nicht mehr verändern kann.
- **Entscheidungsknoten:** Entscheidet mittels Bedingungen, welchen Zustand das Objekt als nächstes annimmt.
- **Manuelle Wahl:** Ermöglicht den Anwender manuell zu wählen, welchen Zustand das Objekt annimmt.
- **Timer:** Drückt aus, wann ein Objekt einen gewissen Zustand annimmt.
- **Parallele Flüsse:** Ermöglicht dem Objekt gleichzeitig mehrere Zustände anzunehmen und zeitgleich verschiedene Aktionen auszuführen.
- **Unterprozess:** Ein Hauptprozess kann einen Unterprozess ausführen.
- **Signal aussenden:** Senden eines Signals.
- **Signal akzeptieren:** Kann ein ausgesendetes Signal akzeptieren.

Abbildung 2.20 zeigt hierzu noch ein simples Beispiel. Das Bauteil befindet sich nach der Instanziierung am Förderband, welches im Laufe des Prozesses von einem Objekt der Klasse ‚Förderband‘ detektiert wird. Diese Aktion befindet sich in der entsprechenden Schwimmbahn. Darauffolgend werden weitere Aktionen von den jeweiligen Instanzen durchgeführt, bis der Endknoten erreicht wird.



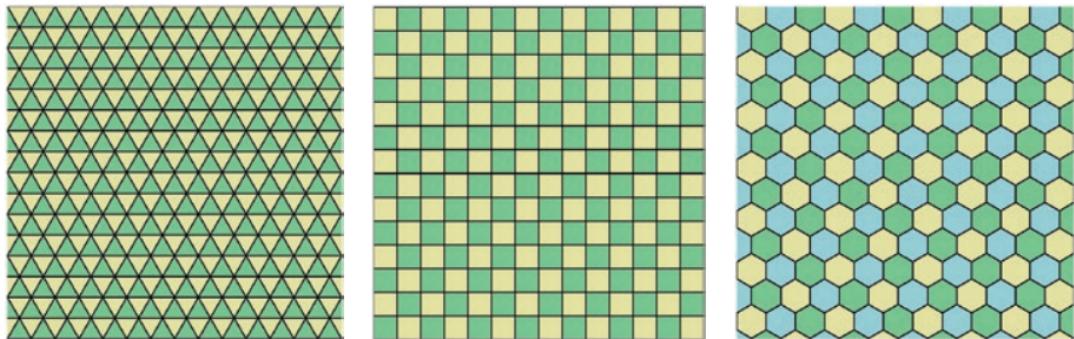
**Abbildung 2.20:** Beispiel eines Aktivitätsdiagramms in UML

## 2.5 Parkettierung der Ebene

Parkettierung [9] beschäftigt sich mit der Anordnung von geometrischen Elementen, um eine Ebene vollständig zu überdecken. Sie gehört zur Geometrie. Die Geometrie ist ein Teilbereich der Mathematik und beschäftigt sich mit der Untersuchung invarianter Größen. Geometrische Elemente sind Teil dieser Wissenschaft und werden in der euklidischen Geometrie beschrieben, benannt nach Euklid von Alexandria (3. Jahrhundert v. Chr.). In der Mathematik unterscheidet man zwischen vielen Arten der Parkettierung wie reguläre und halb-reguläre Parkettierung, periodische und aperiodische Parkettierung oder platonische Parkettierung. In dieser Arbeit liegt der Fokus auf der platonischen Parkettierung.

### 2.5.1 Platonische Parkettierung

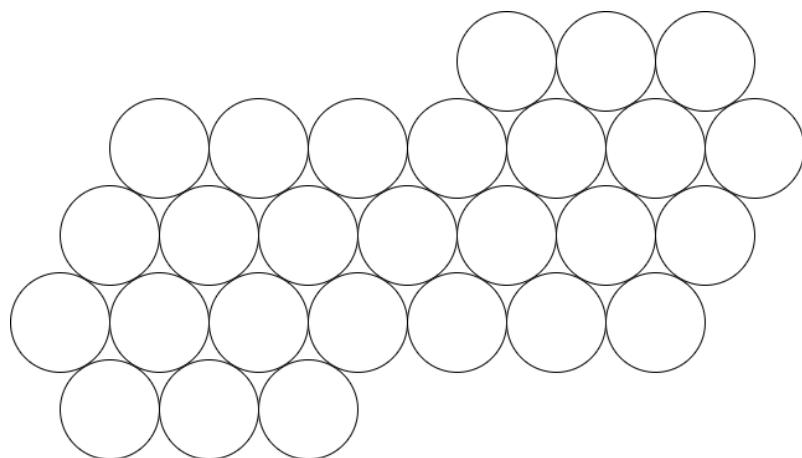
Für die Erstellung eines Layouts für die Chargierung ist die platonische Parkettierung [10] von wesentlicher Bedeutung. Sie beschreibt 3 Möglichkeiten (Abb. 2.21), eine Ebene mit nur einem Typ regelmäßiger Vielecke vollständig auszufüllen. Bei einer Diskretisierung der Chargierebenen könnten somit diese 3 Parkettierungsmuster herangezogen werden, um beliebige Größen und Formen der zu chargierenden Ebenen auszufüllen.



**Abbildung 2.21:** Platonische Parkettierung [10]

### 2.5.2 Kepler'sche Vermutung

Nach einer Vermutung von Johannes Kepler entspricht die dichteste Anordnung von Kugeln im euklidischen Raum, gemessen über die mittlere Dichte, die einer kubisch-flächenzentrierten oder hexagonalen Anordnung [11]. Beide Packungsarten besitzen nämlich die gleiche mittlere Dichte und können als gleichwertig betrachtet werden. Betrachtet man nun einen Schnitt durch eine Symmetrieebene der hexagonalen Packung, gelangt man zur Abb. 2.22 und man kann annehmen, dass es sich dabei um die dichteste Packung von Kreisen handelt. Nun könnte diese Anordnung für zylindrische Bauteile verwendet werden um die maximale Anzahl dieses Typs auf eine Chargiergestell abzulegen. Die Vermutung gilt durch [12] mittlerweile als bewiesen.



**Abbildung 2.22:** Schnitt durch eine hexagonale Kugelpackung

## 2.6 Machine Learning

ML wird seit einiger Zeit als vielversprechende Technologie angesehen. In den vergangenen Jahren wurden auch immer weitere Methoden entwickelt oder weiterentwickelt, um diese Technologie vielseitiger einsetzbar zu machen. In diesem Kapitel werden, nach einer Vorstellung der Grundlagen, einige dieser Methoden präsentiert. Grundlegend gliedert sich ML in 3 Teilbereiche: Supervised Learning (SL), Unsupervised Learning (UL) und Reinforcement Learning (RL) (Abb. 2.23). Die folgenden Erläuterungen wurden [13], [14] und [15] entnommen, Kapitelnummer und etwaige weitere Referenzen sind im Text angegeben. Für die geplante Implementierung werden außerdem die entsprechenden Klassen der Bibliothek PYTORCH [16] vorgestellt.

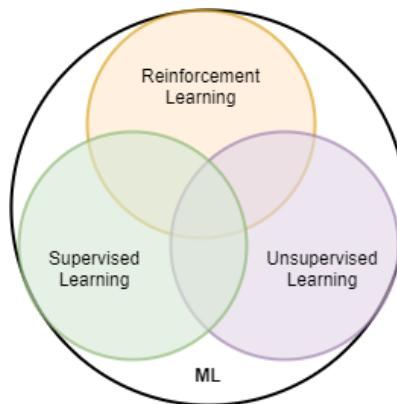


Abbildung 2.23: Teilbereiche des *machine learnings*

### 2.6.1 Basics

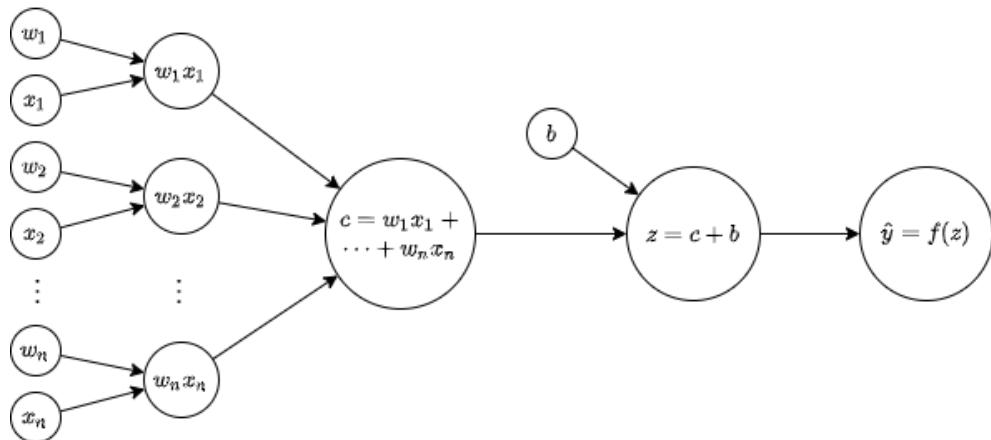
In diesem Abschnitt werden die Grundlagen von neuronalen Netzwerken erläutert, welche nötig sind, um die Methoden von ML anwenden zu können. Dies beinhaltete den Aufbau eines Neurons, Aktivierungsfunktionen, das Gradientenverfahren, Initialisierung von Gewichtungsfaktoren, Kostenfunktionale, *regularization*, den Aufbau einer Notation und das Parameter-Tuning.

#### 2.6.1.1 Aufbau eines Neurons

Das Neuron [13, Kap. 2][15, Kap. 10] bildet den Grundbaustein eines jeden neuronalen Netzwerkes und ist somit auch wesentlich für das Verständnis. Ein Neuron besitzt mehrere Eingänge, sogenannte *features*, die im allgemeinen Fall mit einem

Skalar gewichtet werden. Diese gewichteten Eingänge werden anschließend aufsummiert und man erhält eine gewichtete Summe. Des Weiteren wird zu dieser Summe noch ein konstanter, reeller Wert dazu addiert, der jedem Knoten individuell zugeordnet wird, die Rede ist vom sogenannten *bias*. Für die Ausgabe eines Wertes wird auf diese gewichtete Summe mit Gleichanteil noch eine *activation function*, manchmal *transfer function* genannt, angewendet. Der gesamte Sachverhalt ist in Abb. 2.24 grafisch veranschaulicht. Des Weiteren soll an dieser Stelle nicht unerwähnt bleiben, dass sich das gesamte Übertragungsverhalten, wie in Formel 2.1 dargestellt, durch Matrizenmultiplikation zusammenfassen lässt.

$$\hat{y} = f(\mathbf{w}^T \mathbf{x} + b) = f(z) \quad (2.1)$$



**Abbildung 2.24:** Schematischer Aufbau eines Neurons

Eine Änderung der *features*, also den Eingangsvariablen, wird im allgemeinen als *observation* bezeichnet. Für eine eindeutige Unterscheidung von *features* und *observations* wird nun folgende Konvention verwendet:

$$x = x_j^{(i)}$$

Hierbei steht der tiefgestellte Index  $j$  für das  $j$ te *feature* und der hochgestellte Index  $i$  für die  $i$ te *observation*. Somit schreibt sich die Übertragungsfunktion des einzelnen Neurons auch zu:

$$f(z_j^{(i)}) = \hat{y}_j^{(i)}$$

### 2.6.1.2 Aktivierungsfunktionen eines Neurons

Die Aktivierungsfunktion [13, Kap. 2] wird für die Wertausgabe eines Neurons verwendet. Im einfachsten Fall handelt es sich dabei um eine Einheitsmatrix (Glg. 2.2), mithilfe derer die gewichtete Summe mit Gleichanteil unverändert weitergegeben wird. Jedoch gibt es noch weitere Aktivierungsfunktionen, die mehr praktische Bedeutung besitzen. Im Folgenden seien kurz die 4 wesentlichsten angeführt: Sigmoide (Glg. 2.3), Tangens-Hyperbolicus (Glg. 2.4), Rectified-Linear-Unit (ReLU) (Glg. 2.5) sowie Leaky ReLU (Glg. 2.6) und Swish (Glg. 2.7). Einen Überblick über alle in Pytorch zur Verfügung stehende Aktivierungsfunktionen findet man unter [17].

$$f(z) = I(z) = z \quad (2.2) \quad f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)} \quad (2.3)$$

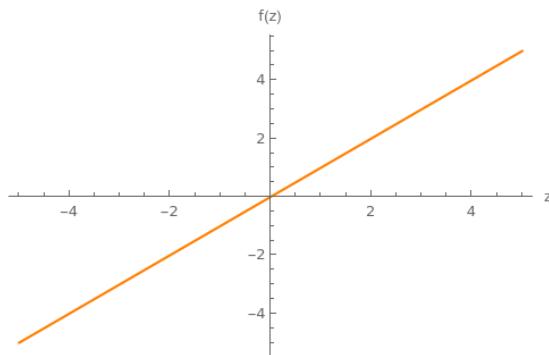


Abbildung 2.25: Identity-Activation

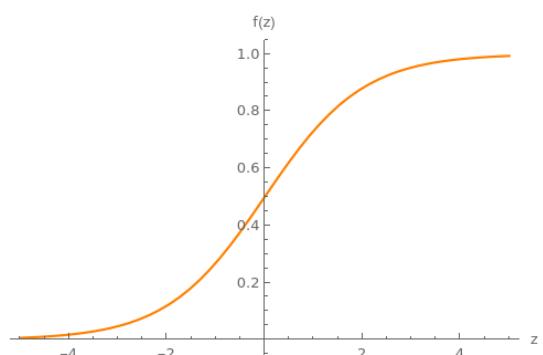


Abbildung 2.26: Sigmoid-Activation

$$f(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \quad (2.4)$$

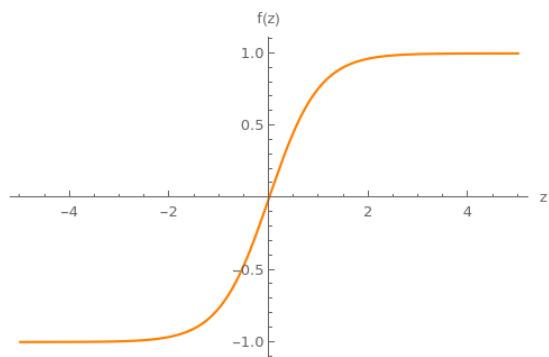


Abbildung 2.27: TanH-Activation

$$f(z) = \max(0, z) \quad (2.5)$$

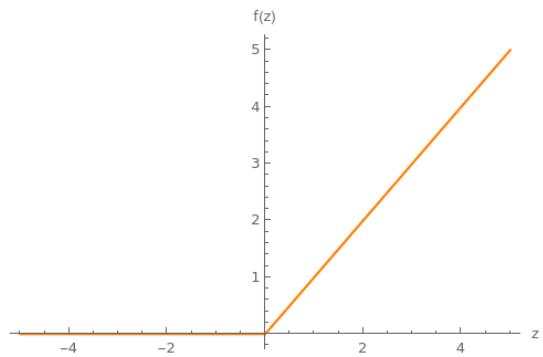


Abbildung 2.28: ReLU-Activation

$$f(z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \quad (2.6) \quad f(z) = z\sigma(\beta z) \quad (2.7)$$

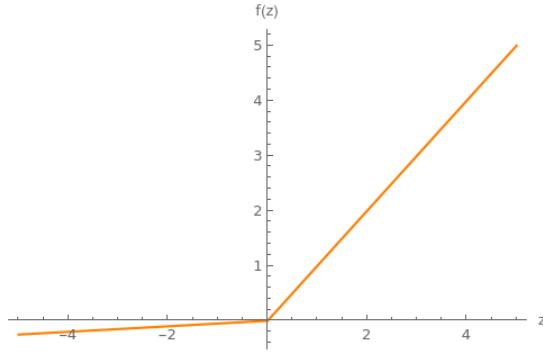


Abbildung 2.29: LeakyReLU-Activation

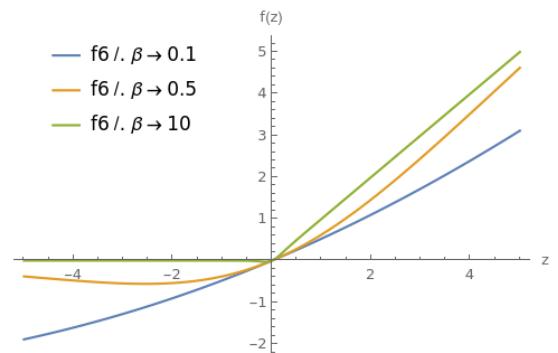


Abbildung 2.30: Swish-Activation

### 2.6.1.3 Bestimmung der Gewichtungsfaktoren

Um die Gewichtungsfaktoren für die Neuronen bestimmen zu können, muss zunächst ein Kostenfunktional der Form:  $J(\mathbf{w})$  definiert werden. Die Gewichtungsfaktoren resultieren aus einer Minimierung dieses Kostenfunktionalen. Man spricht auch von einer Minimierung im sogenannten *weight-space*. Hierfür kann unter anderem das Gradientenverfahren [13, Kap. 2] [15, Kap. 4] Anwendung finden (Glg. 2.8).

$$\begin{cases} \mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \nabla J(\mathbf{w}_n) \\ \mathbf{b}_{n+1} = \mathbf{b}_n - \gamma \nabla J(\mathbf{b}_n) \end{cases} \quad (2.8)$$

Namensgebend für dieses Verfahren ist der in der Formel enthaltene Gradient  $\nabla J(\mathbf{w})$ . Der Gradient, typischerweise als Nabla-Operator gekennzeichnet, gibt die Richtung des steilsten Anstieges einer multidimensionalen Funktion wieder, wobei die Gewichtungsfaktoren hier die Dimensionen repräsentieren. Der Gradient kann durch die partiellen Ableitungen nach den Dimensionsvariablen (Glg. 2.9) bestimmt werden.

$$\nabla J(\mathbf{w}) = \left[ \begin{array}{c} \frac{\partial J}{\partial w_1}(\mathbf{w}) \\ \frac{\partial J}{\partial w_2}(\mathbf{w}) \\ \vdots \\ \frac{\partial J}{\partial w_n}(\mathbf{w}) \end{array} \right] \quad (2.9)$$

Für das Gradienten-Verfahren werden zu Beginn die Gewichtungsfaktoren beliebig initialisiert und anschließend durch die Iteration laufend aktualisiert. Dies kann bis zum Erreichen einer beliebigen Genauigkeit durchgeführt werden. Um das Erreichen der gewünschten Genauigkeit nachzuweisen, wird oft bei einfacheren Aufgaben eine Abbruchbedingung der Form:  $|J(\mathbf{w}_{n+1}) - J(\mathbf{w}_n)| < \epsilon$  definiert. Da die Auswertung meist mit erheblichem Rechenaufwand verbunden ist, besonders im Gebiet der neuronalen Netzwerke, wird in der praktischen Anwendung oft darauf verzichtet und der Algorithmus einfach für eine bestimmte Anzahl an Iterationen ausgeführt. Das Konvergenzverhalten wird dabei oft grafisch, Kostenfunktional über die Iterationsanzahl, in einem dynamischen Plot dargestellt. Der Parameter  $\gamma$  ist in der Mathematik als ‚Relaxationsfaktor‘ bekannt, entspricht im Newton-Verfahren der ‚Schrittweite‘, wird jedoch im Gebiet der neuronalen Netzwerke immer als *learning rate* bezeichnet. Im Allgemeinen wird dieser Parameter zu den *hyperparameters* gezählt, eine Gruppe von Modell-Parameter, die für das Trainieren der Netzwerke von besonderer Bedeutung sind.

Besteht eine Iteration aus einem ganzen Durchgang durch das Datenset (kann aus vielen *observations* bestehen), spricht man auch vom Batch-Gradient Descent (BGD)-Verfahren. Zählt jede *observation* als Iteration, spricht man vom Stochastic-Gradient Descent (SGD)-Verfahren [13, Kap. 3]. Als Beispiel: Ein neuronales Netzwerk soll Bilder von Kühen und von Ziegen unterscheiden, hierfür hat man vielleicht ein Datenset von 40000 Bildern angefertigt. Schickt man nun alle 40000 Bilder durch das Netzwerk und updatet anschließend die Gewichtungsfaktoren mittels des Gradienten-Verfahrens, spricht man von einem BGD-Verfahren. Updatet man die Gewichtungsfaktoren hingegen nach jedem Durchlauf eines jeden einzelnen Bildes, spricht man vom SGD-Verfahren. Nun ergibt sich noch auf natürliche Weise die Möglichkeit, einen Zwischenschritt einzuführen, nämlich den gesamten *batch* in sogenannte *minibatches* aufzuteilen. Diese Möglichkeit findet in der Praxis häufig Anwendung und man nennt diese Vorgehensweise: Minibatch-Gradient Descent (MGD)-Verfahren. Dabei wird jedoch ein neuer *hyperparameter* eingeführt, die *batch size*. Im Allgemeinen hängt die Rechenzeit stark von den Aufrufen des Gradienten-Verfahrens ab, so mit ist das BGD-Verfahren am schnellsten und das SGD-Verfahren am langsamsten. Das MGD-Verfahren, abhängig von der *batch size*, bewegt sich irgendwo dazwischen. Es gibt jedoch noch eine Vielzahl von abgewandelten Formen der Gradienten-Methode [18] [13, Kap. 4] [15, Kap. 11]. Zum einen wäre da die Momentum-Methode (Glg. 2.10) (Glg. 2.11), welche die vergangenen Werte des Gradienten, multipliziert mit dem Parameter  $\beta$ , in die Updates miteinfließen lässt. Die Methode ist weniger sensitiv gegenüber Sattelpunkten.

$$\begin{cases} \nu_{w,n+1} = \beta\nu_{w,n} + (1 - \beta)\nabla J(\mathbf{w}_n) \\ \nu_{b,n+1} = \beta\nu_{b,n} + (1 - \beta)\nabla J(\mathbf{b}_n) \end{cases} \quad (2.10)$$

$$\begin{cases} \mathbf{w}_{n+1} = \mathbf{w}_n - \gamma\nu_{w,n} \\ \mathbf{b}_{n+1} = \mathbf{b}_n - \gamma\nu_{b,n} \end{cases} \quad (2.11)$$

Eine weitere Methode stellt die RMSProp-Methode dar (Glg. 2.12 und 2.13). Gegenüber der Momentum-Methode wurden zwei wesentliche Erweiterungen durchgeführt: Zum einen wurde in der Update-Gleichung für den methoden-spezifischen Parameter der Gradient nochmals über ein Hadamard-Produkt mit sich selbst multipliziert. Zum anderen wurde in der Update-Gleichung die Schrittrichtung durch den Gradienten dividiert durch einen Wurzausdruck ersetzt. Der Faktor  $\epsilon$  unter der Wurzel bleibt konstant und verhindert eine Division durch Null. Dieser Wert beträgt im Normalfall etwa 10e-8. Die Grundlegende Idee hinter dieser Methode ist die Folgende: Besteht der  $\nabla$  aus großen Werten, sind die Ableitungen groß, so ist auch der Parameter  $S$  groß und verlangsamt somit den Lernprozess, da wenn  $S \gg 1$  gilt, auch  $\frac{1}{S} \ll 1$  gilt.

$$\begin{cases} \mathbf{S}_{w,n+1} = \beta\mathbf{S}_{w,n} + (1 - \beta)\nabla J(\mathbf{w}_n) \circ \nabla J(\mathbf{w}_n) \\ \mathbf{S}_{b,n+1} = \beta\mathbf{S}_{b,n} + (1 - \beta)\nabla J(\mathbf{b}_n) \circ \nabla J(\mathbf{b}_n) \end{cases} \quad (2.12)$$

$$\begin{cases} \mathbf{w}_{n+1} = \mathbf{w}_n - \frac{\gamma}{\sqrt{\mathbf{S}_{w,n+1} + \epsilon}} \nabla J(\mathbf{w}_n) \\ \mathbf{b}_{n+1} = \mathbf{b}_n - \frac{\gamma}{\sqrt{\mathbf{S}_{b,n+1} + \epsilon}} \nabla J(\mathbf{b}_n) \end{cases} \quad (2.13)$$

Als Ergänzung zu den bisher betrachteten Optimierungsverfahren, wird nun noch eine weitere Methode vorgestellt: Adaptive Momentum Estimation (Adam). Diese Methode kombiniert Ideen der Optimierer Momentum und RMSProp. Zunächst müssen die methoden-spezifischen Parameter dieser Optimierer bestimmt werden (Glg. 2.10 und 2.12). Dabei gilt, dass die Parameter  $\beta$  in den Gleichungen nicht dieselben sind, daher werden diese nun mit  $\beta_1$  und  $\beta_2$  bezeichnet. Anschließend findet eine Korrektur, hier mit dem hochgestellten Index *mod* dargestellt, dieser Parameter statt (Glg. 2.14 und 2.15). Abschließend kann das Update für die Gewichtungsfaktoren berechnet werden (Glg. 2.16). Pytorch erlaubt eine einfache Benutzung dieser Optimierungs-Methode (Prog. 2.2).

$$\begin{cases} \nu_{w,n}^{mod} = \frac{\nu_{w,n}}{1-\beta_1^n} \\ \nu_{b,n}^{mod} = \frac{\nu_{b,n}}{1-\beta_1^n} \end{cases} \quad (2.14)$$

$$\begin{cases} \mathbf{S}_{w,n}^{mod} = \frac{\mathbf{S}_{w,n}}{1-\beta_2^n} \\ \mathbf{S}_{b,n}^{mod} = \frac{\mathbf{S}_{b,n}}{1-\beta_2^n} \end{cases} \quad (2.15)$$

$$\begin{cases} \mathbf{w}_{n+1} = \mathbf{w}_n - \frac{\gamma}{\sqrt{\mathbf{S}_{w,n+1}^{mod} + \epsilon}} \nu_{w,n}^{mod} \\ \mathbf{b}_{n+1} = \mathbf{b}_n - \frac{\gamma}{\sqrt{\mathbf{S}_{b,n}^{mod} + \epsilon}} \nu_{b,n}^{mod} \end{cases} \quad (2.16)$$

**Programm 2.2:** Adam Klasse Pytorch

```

1 import torch
2
3 torch.optim.Adam(net.params, lr=0.001, betas=(0.9, 0.999),
4                  eps=1e-08, weight_decay=0, amsgrad=False)

```

#### 2.6.1.4 Initialisierung von Gewichtungsfaktoren

Wie bereits in den vorhergehenden Unterkapitel erwähnt, können Gewichtungsfaktoren beliebig initialisiert werden. Die zufällige Initialisierung (Standardnormalverteilung  $\mathcal{N}(0, 1)$ ) kann jedoch unter gewissen Umständen zu numerischen Problemen führen [19]. Daher wurden geeignete Initialisierungsstrategien [13, Kap. 3] [15, Kap. 11] für die jeweiligen Aktivierungsfunktionen bzw. Netzwerkarchitekturen entwickelt. Tabelle 2.3 zeigt beispielsweise Standardabweichungen  $\sigma$  für die meist verwendeten Normal-Verteilte-Initialisierungen der Gewichtungsfaktoren, wobei der Erwartungswert  $\mu$  weiterhin auf Null gesetzt bleibt. Hierbei bezeichnet  $n_{in}$  die Anzahl an Eingangs-*features* der betreffenden Schicht und  $n_{out}$  die Anzahl an Ausgangs-*features*. Für eine Herleitung der Standardabweichung siehe [20], für eine kompakte Zusammenfassung anderer Initialisierungsmethoden siehe [21].

**Tabelle 2.3:** Initialisierungen für Aktivierungsfunktionen

Aktivierungsfunktion	Standardabweichung $\sigma$ für eine Layer	Bezeichnung
Sigmoide	$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$	Xavier-Initialisierung
ReLU	$\sigma = \sqrt{\frac{4}{n_{in} + n_{out}}}$	He-Initialisierung

Das Framework Pytorch bietet eine automatische Wahl der Initialisierungsmethode abhängig vom erstellten Netzwerk. Sind eigene Vorgaben gewünscht, können diese in einer eigenen Funktion in der zu erstellenden Netzwerkklasse vorgegeben werden (Prog. 2.3).

**Programm 2.3:** Gewichts-Initialisierung Pytorch

```

1 import torch.nn as nn
2
3 class Some_Network(nn.Module):
4     def __init__(self, input_dim, n_actions):
5         super(Some_Network, self).__init__()
6         <<insert some layers here>>
7         self._create_weights()
8
9     def _create_weights(self):
10        for m in self.modules():
11            if isinstance(m, nn.Linear):
12                nn.init.xavier_uniform_(m.weight)
13                nn.init.constant_(m.bias, 0)

```

### 2.6.1.5 Kostenfunktional

Als Kostenfunktional [13, Kap. 2] können jene Funktionale verwendet werden, die jene Eigenschaft aus Glg. 2.17 erfüllen, also eine Abbildung von einem beliebig-dimensionalen Raum (normalerweise  $\mathbb{R}^n$ ) auf einen Skalar realisieren.

$$J : \mathbb{R}^n \rightarrow \mathbb{R} \quad \text{mit} \quad \mathbf{w} \mapsto J(\mathbf{w}) \quad (2.17)$$

Das Kostenfunktional konvergiert im allgemeinen Fall nicht gegen Null, sondern nähert sich einem Minimum. In der Praxis oft verwendet wird der klassische, quadratische Mittelwert (Mean Square Error (MSE)). Angeschrieben für die  $i$ te von  $m$  observations ergibt sich der MSE wie in Glg. 2.18 dargestellt.

$$J(\mathbf{w}_j, b) = \frac{1}{m} \sum_{i=1}^m \left( y_j^{(i)} - f(\mathbf{w}_j, b, \mathbf{x}_j^{(i)}) \right)^2 \quad (2.18)$$

Der Term  $y_j^{(i)}$  entspricht dabei dem angestrebten Zielwert. Die *cross entropy* (Glg. 2.19) ist ein weiteres Kostenfunktional, welches beim Lernen von Wahrscheinlichkeitsaussagen Verwendung findet [22]. Als Anmerkung sei hier erwähnt, dass die

Verwendung dieser Funktion meist mit der Verwendung der Sigmoid-Aktivierungsfunktion einhergeht, da am Ausgang der Neuronen ein Wert aus dem Intervall  $[0, 1]$  erwartet wird.

$$J(\mathbf{w}_j, b) = \frac{1}{m} \sum_{i=1}^m L(y_j^{(i)}, f(\mathbf{w}_j, b, \mathbf{x}_j^{(i)})) \quad (2.19)$$

Der sogenannte *Huber loss* ist ein weiteres Kostenfunktional. Für eine übersichtlichere Notation definiert man nochmals das Residuum  $r$  mit  $r := y_j^{(i)} - f(\mathbf{w}_j, b, \mathbf{x}_j^{(i)})$ . Somit lässt sich das Kostenfunktional folgendermaßen anschreiben:

$$J(\mathbf{w}_j, b) = \frac{1}{m} \sum_{i=1}^m \begin{cases} \frac{1}{2}(r)^2 & \text{for } |r| \leq \delta \\ \delta(|r| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \quad (2.20)$$

Die Besonderheit hierbei ist die Quadratur für kleine Residuen und das lineare Verhalten bei Residuen jenseits des Schwellwerts  $\delta$ . Die Findung eines idealen Schwellwertes ist dabei jedoch nicht trivial und ist Gegenstand aktueller Forschung [23]. Pytorch bietet hier wieder eine Sammlung an vordefinierten Funktionen, welche im ‚`torch.nn`‘-Modul zur Verfügung stehen (Prog. 2.4).

**Programm 2.4:** Kostenfunktionale Pytorch

```

1 import torch.nn as nn
2
3 MSE = nn.MSELoss(size_average=None, reduce=None, reduction='mean')
4 CrossEntropy = nn.CrossEntropyLoss(weight=None, size_average=None,
5                                     ignore_index=-100, reduce=None,
6                                     reduction='mean')
7 HuberLoss = nn.HuberLoss(reduction='mean', delta=1.0)

```

### 2.6.1.6 Regularization und Dropout

Die *regularization* [13, Kap. 5][15, Kap. 11] ist ein Eingriff in das Update-Verhalten der Gewichtungsfaktoren. Ziel ist es, dem Problem des *overfittings* entgegenzuwirken. Durch Addition der Gewichtungsfaktoren zum Kostenfunktional, wird eine Reduzierung der Gewichtungsfaktoren auf Null, beim Einsatz der Optimierungs-Methoden, verhindert. Als Beispiel seien hier die  $l_1$ - und  $l_2$ -*regularization* angeführt (Glg. 2.21 und 2.22). Des Weiteren kommt dabei noch ein neuer Parameter  $\lambda$  zur Beschreibung unseres Modells hinzu. In Pytorch wird dieser Parameter als ‚`weight_decay`‘ bezeichnet und wird bei der Initialisierung des Optimierers mitübergeben (Prog. 2.2).

Eine weitere Methode um *overfitting* zu verhindern, ist das *dropout*-Verfahren. Hierfür wird eine *keep probability* (Behalte-Wahrscheinlichkeit) definiert, die angibt, mit welcher Wahrscheinlichkeit ein Neuron nach Abschluss eines Iteration-Schrittes weiter bestehen bleibt, also nicht gelöscht wird. Die Löschung findet dabei nur während des Trainings statt, für eine Anwendung des trainierten Netzwerkes auf neue Daten kommt wieder das vollständige Netzwerk zum Einsatz. Wieder bietet Pytorch eine dafür vorgesehene Klasse an (Prog. 2.5).

$$J(\tilde{\mathbf{w}}) = J(\mathbf{w}) + \frac{\lambda}{m} \|\mathbf{w}\|_1 \quad (2.21)$$

$$J(\tilde{\mathbf{w}}) = J(\mathbf{w}) + \frac{\lambda}{m} \|\mathbf{w}\|_2^2 \quad (2.22)$$

**Programm 2.5:** Dropout Pytorch

```

1 import torch.nn as nn
2
3 Dropout = Dropout(p=0.5, inplace=False)
4 test_set = torch.randn(20,20)
5 output = Dropout(test_set)

```

### 2.6.1.7 Notation für Features und Observations

Das gesamte Set an Input-Daten eines Neurons kann als Matrix dargestellt werden (Glg. 2.23) [13, Kap. 3], dabei beschreibt  $n$  die Anzahl an *features* und  $m$  die Anzahl an *observations*. Diese Notation ist dahergehend wichtig, da Frameworks wie TENSORFLOW oder Pytorch ihre Matrizen bei der Initialisierung anlegen müssen.

$$\mathbf{X} = \begin{pmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{pmatrix} \quad (2.23)$$

Durch diese Notation resultiert eine besonders angenehme Schreibweise für das Argument der Übertragungsfunktion einzelner Neuronen (Glg. 2.24).

$$\mathbf{z} = (z^{(1)} \ z^{(2)} \ \dots \ z^{(m)}) = \mathbf{w}^T \mathbf{X} + b \quad (2.24)$$

Hierdurch können die Übertragungsfunktionen der Neuronen für den gesamten Lernvorgang kompakt angeben können (Glg. 2.25), wobei mit  $f(\mathbf{z})$  die elementweise Anwendung auf  $\mathbf{z}$  zu verstehen ist.

$$\hat{\mathbf{y}} = (\hat{y}^{(1)} \quad \hat{y}^{(2)} \quad \dots \quad \hat{y}^{(m)}) = (f(z^{(1)}) \quad f(z^{(2)}) \quad \dots \quad f(z^{(m)})) = f(\mathbf{z}) \quad (2.25)$$

Bei Netzwerken mit mehreren Neuronen, was im Allgemeinen meistens der Fall ist, hat es sich bewährt, auch die Gewichtungsfaktoren als Matrizen, abhängig von der aktuellen Schicht  $l$ , anzuschreiben (Glg. 2.26). Die Dimension dieser Matrix ergibt sich aus  $n_l * n_x$ :  $n_x$  entspricht der Anzahl an Neuronen in der Ausgangsschicht und  $n_l$  der Anzahl an Neuronen in der Folgeschicht. Auch der *bias* aller Neuronen (der Folgeschicht) wird in einen Vektor bzw. Matrix zusammengefasst (Glg. 2.27).

$$\mathbf{W}^{[l]} = \begin{pmatrix} w_{1,l}^{(l)} & w_{1,2}^{(l)} & \dots & w_{1,n_x}^{(l)} \\ w_{2,l}^{(l)} & w_{2,2}^{(l)} & \dots & w_{2,n_x}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,l}^{(l)} & w_{n_l,2}^{(l)} & \dots & w_{n_l,n_x}^{(l)} \end{pmatrix} \quad (2.26)$$

$$\mathbf{B}^{[l]} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{pmatrix} \quad (2.27)$$

Mit dieser Notation erhält man somit auch eine weitere Matrix für die gewichtete Summe  $z$  aller Neuronen der Folgeschicht, ausgehend vom Input  $\mathbf{X}$  (Glg. 2.28). Nun kann dies auch allgemeiner für eine beliebige Schicht angeschrieben werden (Glg. 2.29), da der Input schließlich nur die vorhergehende Schicht der Folgeschicht darstellt.

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \mathbf{X} + \mathbf{B}^{[l]} \quad (2.28)$$

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \mathbf{Z}^{[l-1]} + \mathbf{B}^{[l]} \quad (2.29)$$

Schlussendlich können somit auch noch die Übertragungsfunktionen der Neuronen für alle Schichten neu angeschrieben werden (Glg. 2.30), wobei hier vorausgesetzt wird, dass alle Neuronen die selbe Aktivierungsfunktionen verwenden und der Vektor der Aktivierungsfunktionen  $\mathbf{f}$  nur gleiche Einträge beherbergen.

$$\mathbf{Y}^{[l]} = \mathbf{f}^{[l]}(\mathbf{Z}^{[l]}) \quad (2.30)$$

Da es nicht als selbstverständlich erscheint, sich alle Dimensionen dieser Matrizen für die Implementierung zu merken, werden in Tabelle 2.4 die Dimensionen der Matrizen für mehrschichtige Netzwerke nochmals zusammengefasst. Der Indizes  $l$  geht dabei von 1 bis  $L$ , der Anzahl an Schichten des neuronalen Netzwerkes.

**Tabelle 2.4:** Dimensionen der Matrizen aus diesem Kapitel

Matrix	Dimension
$\mathbf{W}^{[l]}$	$n_l \times n_{l-1}$
$\mathbf{B}^{[l]}$	$n_l \times 1$
$\mathbf{Z}^{[l-1]}$	$n_{l-1} \times 1$
$\mathbf{Z}^{[l]}$	$n_l \times 1$
$\mathbf{Y}^{[l]}$	$n_l \times 1$

### 2.6.1.8 Hyper-Parameter-Tuning

Für die Suche nach geeigneten *hyperparameters* [13, Kap. 7] [15, Kap. 2], allgemein das Lösen eines *black-box*-Problems, wird in der Praxis zwischen folgenden Vorgehen unterschieden: *grid search*, *random search*, *coarse-to-fine optimization* und *bayesian optimization*. Die Methoden *grid-search* und *random-search* sind hierbei die Einfachsten. Bei *grid search* wird das Intervall, aus dem ein Parameter entnommen wird, äquidistant diskretisiert und das System für alle diese Parameter ausgewertet. Der Extremwert wird anschließend gleich dem größten oder kleinsten dieser diskreten Ergebnisse angenommen. Für *random search* wird das gleiche Vorgehen gewählt, mit dem einzigen Unterschied, dass das Intervall für die Parameter nicht äquidistant aufgeteilt wird, sonder dem Intervall zufällig Werte entnommen werden. Interessant ist hier, dass diese Methode im Normalfall dem *grid search* überlegen ist. Eine Verbesserung von *random search* ist die *coarse-to-fine optimization*. Hierbei wird die Funktion, wie in unserem Fall das Kostenfunktional, zunächst für eine beliebige

Anzahl an Zufallswerten für die Parameter ausgewertet. Aus den Ergebnissen wird anschließend das Maximum bestimmt. Anschließend werden wieder Zufallszahlen, normal-verteilt mit Mittelwert gleich der Extremstelle, generiert, um die Funktion erneut auszuwerten. Dieser Vorgang wird wiederholt, bis die Stelle des Extremwertes keine wesentlichen Änderungen mehr erfährt. Die Streuung ist dabei dem eigenen Erfahrungsschatz zu entnehmen.

Auch hierfür wurden wieder Bibliotheken entwickelt, welche die Implementierung des Optimierungsverfahrens vereinfachen. Pytorch empfiehlt hierfür die Bibliothek RAYTUNE und erklärt dessen Anwendung mithilfe eines Beispiels [24]. Wesentliche Erweiterung des Programmcodes besteht im Anlegen eines *search space* wie in Prog. 2.6.

**Programm 2.6:** Search space Raytune

```

1 from ray import tune
2 import numpy.random.randint as rndI
3
4 search_space = {
5     "nodes_layer1": tune.sample_from(lambda _: 2**rndI(2, 9)),
6     "nodes_layer2": tune.sample_from(lambda _: 2**rndI(2, 9)),
7     "lr": tune.loguniform(1e-4, 1e-1),
8     "batch_size": tune.choice([2, 4, 8, 16])
9 }
```

## 2.6.2 Supervised Learning

In diesem Kapitel werden die 3 Netzwerke: Feedforward Neural Network (FNN), Convolutional Neural Network (CNN) und Recurrent Neural Network (RNN) vorgestellt, welche in das Gebiet SL fallen. Das FNN besteht primär aus Neuronen welche vernetzt werden, um Operationen an den Eingangsdaten durchzuführen. CNNs führt zusätzliche *layers* ein, welche eine *feature extraction* ermöglichen und somit auch auf *raw pixel* angewendet werden können. Beim RNN hängt die Ausgabe vom vorherigen Zustand der *features* am Eingang ab, was eine Berücksichtigung der Vergangenheit ermöglicht.

### 2.6.2.1 Feedforward Neural Networks

In diesem Abschnitt wird die Funktionsweise von FNNs [13, Kap. 3] erläutert. Unter *feedforward* versteht man den Vorgang, Input-Daten durch ein neuronales Netzwerk und dessen Schichten zu schicken, um Werte am Ausgang zu erhalten (Abb. 2.31). Die dafür benötigte Notation wurde im Kap. 2.6.1 vorgestellt. Implizit wurden somit

auch neue *hyperparameter* vorgestellt, nämlich  $L$  die Anzahl an Schichten,  $n_i$  die Anzahl an Neuronen pro Schicht und  $\mathbf{f}^{[l]}$  die Aktivierungsfunktionen der einzelnen Schichten.

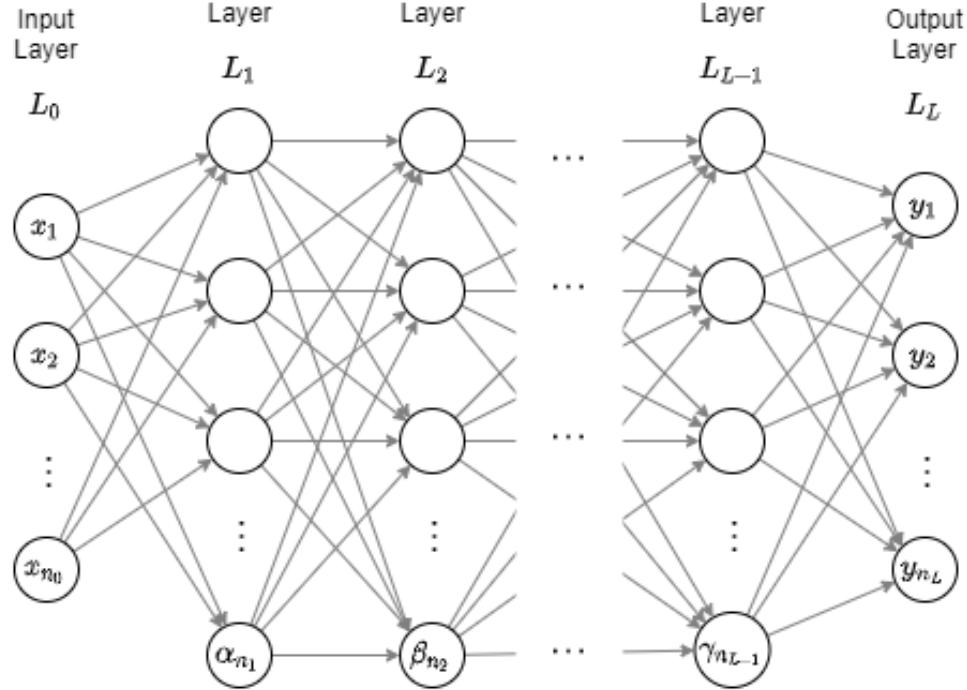


Abbildung 2.31: FNN

Bei der internen Weitergabe der gewichteten Summanden der letzten Schicht gilt es noch zu beachten, dass diese für Wahrscheinlichkeitsausgaben noch normiert, also auf das Intervall  $[0, 1]$  projiziert werden müssen. Außerdem muss die Summe aller Summanden den Wert 1 ergeben. Diese Bedingungen können mittels einer Softmax-Funktion erfüllt werden (Glg. 2.31 und 2.32).

$$\mathbf{S}(\mathbf{Z})_i = \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)} \quad (2.31)$$

$$\sum_{i=1}^k \mathbf{S}(\mathbf{Z})_i = \sum_{i=1}^k \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)} = 1 \quad (2.32)$$

Mit dem bis jetzt erarbeiteten Wissen könnte man bereits einen Klassifizierer mithilfe eines Frameworks implementieren. Zunächst sollte man sich jedoch noch

Gedanken zu der Anzahl an lernbaren *fit*-Parametern machen. Die Anzahl an zu findende Parameter ergibt sich für jede Schicht aus der Anzahl an Elementen in der Matrix  $\mathbf{W}^{[l]}$  und den Elementen in  $\mathbf{B}^{[l]}$ . Aus den in Tabelle 2.4 gegebenen Dimensionen lässt sich somit die Anzahl der Freiheitsgrade bestimmen (Glg. 2.33). Außerdem kann man somit die Anzahl der Freiheitsgrade der gesamten Netzarchitektur  $K$  berechnen (Glg. 2.34). Über diesen Kennwert lässt sich zwar keine konkrete Aussage über die Güte des Netzwerkes tätigen, sollte jedoch vom Anwender berücksichtigt werden. Beispielsweise könnte man die gewonnene Information wie folgt verwenden: Man trainiert Netzwerke mit gleicher Anzahl von Schichten, jedoch ungleicher Anzahl an Neuronen und bestimmt anschließend den Favoriten. Anschließend könnte man den Favoriten für konstante Werte von  $K$ , in Netzwerke mit gleicher Anzahl an Neuronen, jedoch unterschiedlicher Anzahl an Schichten, umwandeln und so weitere Optimierungsschritte durchführen.

Zu der benötigten Anzahl an Freiheitsgraden gibt es eigene Forschungsarbeiten im Bereich *effective degrees of freedom* [25].

$$K^{[l]} = n_l n_{l-1} + n_l = n_l(n_{l-1} + 1) \quad (2.33)$$

$$K = \sum_{l=1}^L n_l(n_{l-1} + 1) \quad (2.34)$$

Programm 2.7 zeigt nun ein solches Netzwerk angelegt in Pytorch. Um den Programmcode kompakt zu halten, wird eine Sequenz von linearen Schichten der dafür vorgesehenen Klasse übergeben.

**Programm 2.7:** FNN Pytorch

```

1 import torch.nn as nn
2
3 class FeedForwardNet(nn.Module):
4     def __init__(FeedForwardNet, input_dim, output_dim,
5                  nodes_layer1=120, nodes_layer2=84):
6         super(Net, self).__init__()
7         self.ff = nn.Sequential(
8             nn.Linear(input_dim, nodes_layer1), nn.ReLU(),
9             nn.Linear(nodes_layer1, nodes_layer2), nn.ReLU(),
10            nn.Linear(nodes_layer2, output_dim))
11
12     def forward(self, x):
13         return self.ff(x)

```

### 2.6.2.2 Convolutional Neural Networks

Eine Erweiterung des FNNs stellt das CNN [13, Kap. 8] [15, Kap. 14] dar. Diese Netzwerke ermöglichen es, einem neuronalen Netz effizient ganze *frames* zu übergeben, da es bei größeren Datenmengen eine *feature extraction* verwirklicht. Um auf den Unterschied der beiden Netzwerke eingehen zu können, wird nun kurz die Faltung (*convolution*) und das *pooling* vorgestellt.

#### Faltung

Bei Faltung ist die Filterung, auch bekannt aus der Bildverarbeitung, mittels Filtermatrizen (*kernels*) gemeint. Die Filtermatrizen sind im Normalfall quadratisch und in ihrer Dimension ungerade. Für die Faltung wird die zu filternde Matrix  $A$  vom Filterkern  $K$  mittels einer Schrittweite  $s$  durchwandert und eine neue, gefilterte Matrix erstellt (Glg. 2.35). Dabei besitzt der Filterkern die Dimension  $(2n_a + 1) \times (2n_b + 1)$ . Bei neuronalen Netzen, besonders bei Klassifizierungsaufgaben, wird diese Methode eingesetzt, um Merkmale aus einem gegebenen Datensatz bessere herauszuarbeiten. In der Bildverarbeitung häufig verwendete Filterkerne sind: Glättungs- bzw. Mittelwertfilter, Schärfungsfilter, Laplace-Filter usw. Hier kurz vorgestellt wird nur der Einsatz auf zweidimensionale Matrizen, jedoch kann die Methode auch für drei- oder höherdimensionale Matrizen verwendet werden, wobei der Filterkern in seiner Dimension einfach angepasst wird. Man beachte: durch die Einführung von solchen Rechenoperationen in die eigene Netzarchitektur wird die Anzahl an *hyperparameters* weiter erhöht.

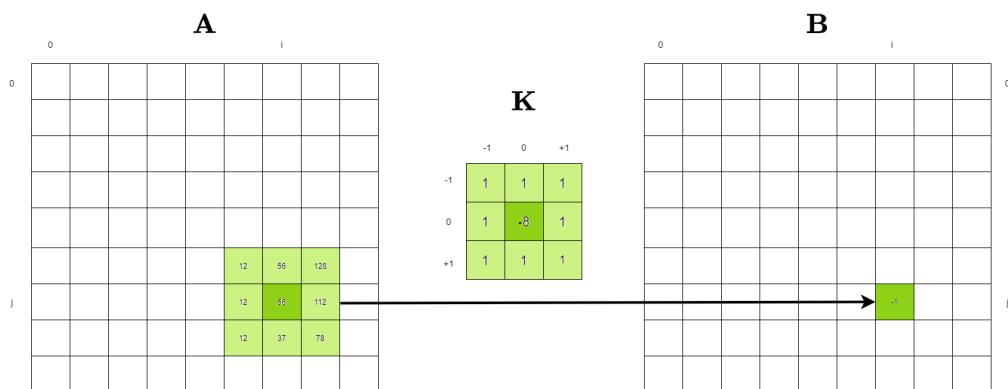


Abbildung 2.32: Einfacher Laplace-Filter-Kern

$$B_{i,j} = \sum_{k=-n_a}^{n_a} \sum_{l=-n_b}^{n_b} A_{i+k,j+l} K_{k,l} \quad (2.35)$$

Abbildung 2.32 zeigt eine Anwendung der Formel 2.35 mit  $n_a = n_b = 1$  und einer Schrittweite  $s = 1$ . Die Abbildung ist zudem ein Spezialfall der Anwendung ( $\text{Dim}(A) = \text{Dim}(B)$ ), da für  $i = j = 0$  die Rechenvorschrift nicht direkt verwendet werden kann. Hier wurde das sogenannten *padding* angewendet, um die Dimension der Eingangsmatrix zu erhalten. Beim *padding* erweitert man die Eingangsmatrix gedanklich mit den benötigten Zeilen und Spalten. Die Einträge der Spalten- und Zeilenvektoren sind zwar beliebig, werden zumeist jedoch mit Null gewählt.

Würde die Schrittweite nicht mit  $s = 1$  gewählt werden, würden die Dimensionen von  $A$  und  $B$  auseinanderdriften. In einem solchen Fall wäre das Inkrement von  $i$  und  $j$  verantwortlich für die Dimensionsreduktion und die Formel 2.35 müsste umgeschrieben werden. Für tiefergehende Details wird an dieser Stelle jedoch auf entsprechende Literatur verwiesen [26].

Um diese Filter nun einsetzen zu können, wird die im Programmcode 2.8 gezeigte Klasse verwendet. Bei den Einträgen des Filterkerns handelt es sich um gesuchte Gewichtungsfaktoren.

#### Programm 2.8: Convolution layer Pytorch

```

1 import torch.nn as nn
2
3 conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
4                  padding=0, dilation=1, groups=1, bias=True,
5                  padding_mode='zeros')
```

## Pooling

Beim *pooling* handelt es sich um eine ähnliche Operation wie beim Filtern mittels Faltung. Der Unterschied besteht darin, dass im Bereich des Filterkerns (*pool*) keine Rechenoperation im eigentlichen Sinn durchgeführt wird, sondern der Maximal- oder Mittelwert des *pools* in die Ausgangs-Matrix übertragen wird. Hierbei ist die resultierende Dimensionsreduktion gewünscht und wird durch größere Schrittweiten zumeist noch verstärkt. Fürs Verständnis ein kurzes Beispiel:  $A$  sei eine 4x4-Matrix, als *pool* wird eine 2x2-Matrix festgelegt und als Schrittweite wird  $s = 2$  gewählt. Nun lässt man den *pool* die Eingangsmatrix durchlaufen. In Formel 2.36 wurden die Position des *pools* durch fett markierte Zahlenwerte verdeutlicht. An jeder Position des *pools* wird das Maximum bestimmt und in  $B$  übertragen (Glg. 2.37). Die resultierende Dimension von  $B$  kann mittels Formel 2.38 vorab bestimmt werden.

$$\begin{aligned}
 \mathbf{A} &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \\
 &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \tag{2.36}
 \end{aligned}$$

$$\mathbf{A} \rightarrow \mathbf{B} = \begin{pmatrix} 6 & 8 \\ 14 & 16 \end{pmatrix} \tag{2.37}$$

$$n_B = \frac{n_A - n_K}{s} + 1 \tag{2.38}$$

Abschließend zum *pooling* sei noch die dafür vorgesehene Klasse von Pytorch angeführt (Prog. 2.9):

**Programm 2.9:** *Pooling layer* Pytorch

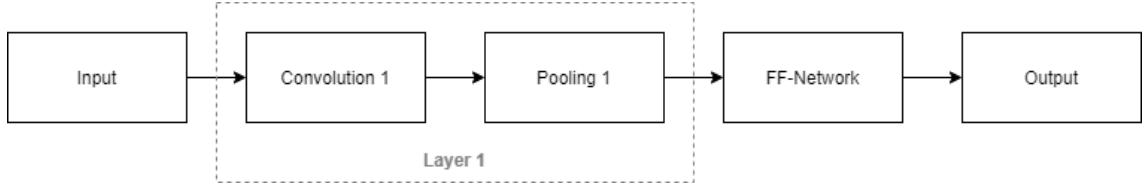
```

1 import torch.nn as nn
2
3 pool = nn.MaxPool2d(kernel_size, stride=None, padding=0,
4                      dilation=1, return_indices=False,
5                      ceil_mode=False)

```

Nun, nach dieser kurzen Einführung, kann der Aufbau eines CNNs vorgestellt werden. Diese Netzwerke bestehen nämlich aus Filterschichten und einem FNN. Eine Filterschicht besteht meist wiederum aus einer Faltung, gefolgt von einem *pooling*. Die Abbildung 2.33 zeigt eine solches Netzwerk. Mit den Bezeichnungen bereits angedeutet, können auch mehrere Schichten bestehend aus Filter und *pooling* vor das FNN geschaltet werden.

## 2 Recherche



**Abbildung 2.33:** Schichten eines CNNs

Programm 2.10 zeigt einen Aufbau eines solchen Netzwerks. Hierfür wurde das Programm 2.7 um die benötigten Schichten erweitert. Die in Abb. 2.33 gezeigte *pipeline* wird von der Funktion ‚forward(self, x)‘ abgebildet. Außerdem bietet Pytorch die Möglichkeit bereits etablierte Architekturen wie ALEXNET, INCEPTION und viele weitere zu verwenden [27].

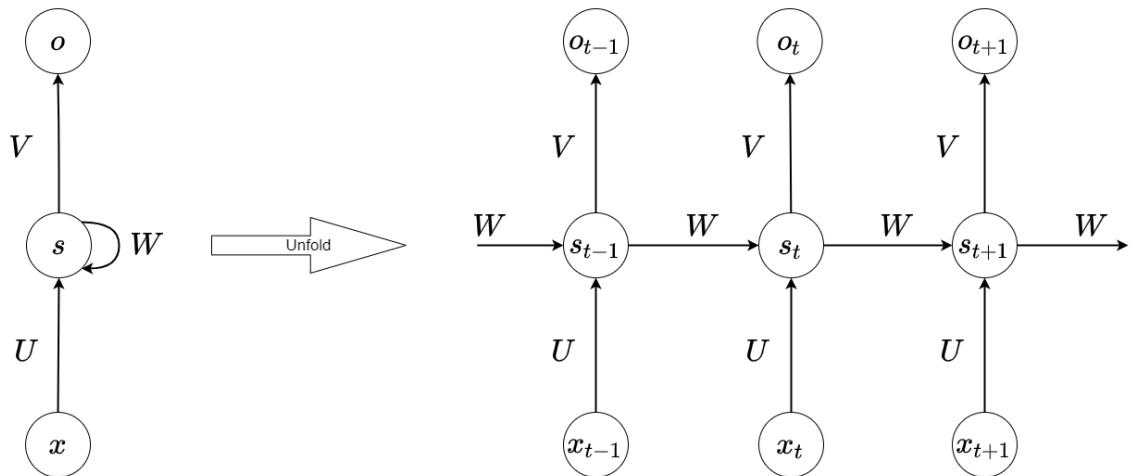
**Programm 2.10:** CNN Pytorch

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 class FeedForwardNet(nn.Module):
5     def __init__(FeedForwardNet, input_dim, output_dim,
6                  nodes_layer1=120, nodes_layer2=84):
7         super(Net, self).__init__()
8         self.conv1 = nn.Conv2d(3, 6, 5)
9         self.pool = nn.MaxPool2d(2, 2)
10        self.conv2 = nn.Conv2d(6, 16, 5)
11        self.ff = nn.Sequential(
12            nn.Linear(input_dim, nodes_layer1), nn.ReLU(),
13            nn.Linear(nodes_layer1, nodes_layer2), nn.ReLU(),
14            nn.Linear(nodes_layer2, output_dim))
15        self.input_dim = input_dim
16
17    def forward(self, x):
18        x = self.pool(F.relu(self.conv1(x))) #Layer 1
19        x = self.pool(F.relu(self.conv2(x))) #Layer 2
20        x = x.view(-1, self.input_dim) #reshape
21        return self.ff(x)
```

### 2.6.2.3 Recurrent Neural Networks

RNNs [13, Kap. 8] unterscheiden sich wesentlich zu den bisher behandelten Netzwerken und werden für Daten verwendet, bei denen die Reihenfolge entscheidend ist. Solche Daten könnten zum Beispiel Sätze sein, wie sie bei *chatbots* verarbeitet werden. Die Eingabedaten bestehen nun also als Sequenzen und das Netzwerk

wendet seine Operationen auf all diese Sequenzen an. Des Weiteren werden vorhergehende Sequenzen indirekt über einen internen Zustand gespeichert, um darauf eine Vorhersage für die nächste Sequenz zu bilden. Abbildung 2.34 zeigt die zwei Darstellungsweisen dieser Netzwerke (1 Neuron), links die zusammengeklappte (*folded*) und rechts die ausgebreitete (*unfolded*).  $x$  beschreibt die Eingangsdaten bzw. Sequenzen, zum Beispiel die Buchstaben eines Wortes. Jede Sequenz wird in seiner Reihenfolge dem Netzwerk übergeben, dabei ändert sich der Zustand der aktuellen Schicht, jedoch nicht deren Gewichtungsfaktoren. Wird nun eine weitere Sequenz dem Netzwerk übergeben, wird die Ausgabe erheblich vom aktuellen Zustand beeinflusst. Die Reihenfolge der Sequenzen wird durch den Index  $t$  gekennzeichnet. Wie in der Abbildung ersichtlich, spielen hier zwei Sätze von Gewichtungsfaktoren eine Rolle,  $\mathbf{W}$  und  $\mathbf{U}$ .



**Abbildung 2.34:** Recurrent network

Der interne Zustand berechnet sich mittels dieser beiden Gewichtungsmatrizen. Mathematisch könnte man den Zusammenhang wie in Formel 2.39 dargestellt, wiedergeben. Hier erkennt man auch den Grund der Bezeichnung *recurrent*, denn der Zustand ist rekursiv definiert, hängt also von allen vorangegangenen Werten ab.

$$s_t = f(\mathbf{U}x_t + \mathbf{W}s_{t-1}) \quad (2.39)$$

Anwendungsgebiete sind hierfür: Textgenerierung, Übersetzungen, Spracherkennung, *chatbots* und vieles mehr. Des Weiteren gibt es auch noch ausgefeilte Neuronen für diese Art von Netzwerk, wie zum Beispiel Long-Short-Term Memory (LSTM) und Gated Recurrent Unit (GRU), welche unter anderem auch in *source-codes* von RL-agents wiedergefunden werden können [28].

### 2.6.3 Unsupervised Learning

Der Bereich UL [15, Kap. 9] beschäftigt sich mit Komprimierungs- und Segmentierungsaufgaben (*clustering*). Als Eingang werden meist Bilder verwendet, wie es auch bei CNNs der Fall ist. Eine Möglichkeit der Implementierung der Segmentierungsaufgabe in Pytorch wird von [29] vorgestellt. Der Autor implementierte hierfür eine *encoder*- und *decoder*-Klasse, welche in einer *autoencoder*-Klasse zusammengefügt wurden. Diese Klasse wurde anschließend in einer Trainingsroutine verwendet, um die Parameter der *encoder*-Klasse anzupassen. Das *encoding* richtet sich dabei nach der VGG-16 Architektur (CNN). Das Clustern wurde im nächsten Schritt mithilfe der Local Aggregation (LA)-Methode verwirklicht, welche eine Funktion definiert, die angibt, wie gut eine Ansammlung von *extracted features* in eine Gruppe passt. Die Implementierung dieser Methode ist durchwegs anspruchsvoll, daher wird hier auf nähere Betrachtungen verzichtet, da der Bereich UL hier nur zur Vervollständigung angeführt wird.

### 2.6.4 Reinforcement Learning

In diesem Kapitel wird nun die Basis von RL [14, Kap. 1] erarbeitet, um in der Konzeptentwicklung Methoden aus diesem Bereich anwenden zu können. Als Einführung folgt eine Vorstellung der wesentlichsten Begriffe, darauffolgend werden diese Begriffe noch in den Kontext zueinander gesetzt. Anschließend werden die Details in den jeweiligen Unterkapiteln ausgearbeitet

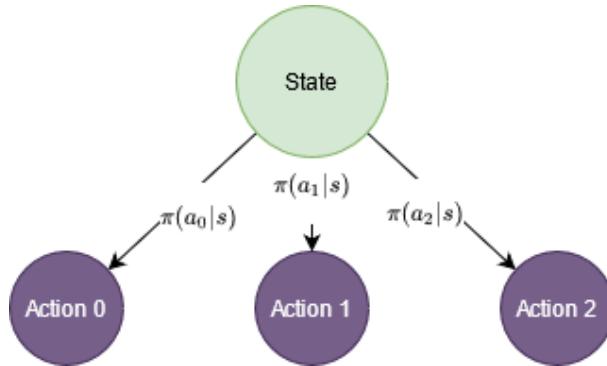
- *Agent*: Ein Konstrukt zur Interaktion mit der Umgebung.
- *Environment*: Die Umgebung, in der ein *agent* eine Aktion ausführen kann.
- *Action*: Eine mögliche Aktion, die der *agent* in der Umgebung ausführen kann.
- *Current state*: Informationen, die dem *agent* übermittelt werden.
- *Policy*: Die Vorschrift, die der *agent* lernt. Sie beschreibt, welche Aktion bei einem Zustand ausgeführt werden soll.
- *Reward*: Die Belohnung oder Bestrafung, die aus einer Aktion resultiert.
- *New state*: Der Zustand, den die Umgebung nach der getätigten Aktion einnimmt. Ein beobachtbarer Teil des Zustands entspricht einer *observation*.
- *Value*: Der Wert eines Zustandes. Er besteht aus der Summe aller folgenden Belohnungen und widerspiegelt die Wertigkeit des Zustandes, wenn die Interaktion nach gewählter Vorschrift abläuft.

Zusammenfassend: *Value* entsteht durch *rewards*, welche man durch *actions* erhält. Der *agent* sammelt diese *rewards*, währenddessen er einer gewissen *policy* folgt und summiert diese auf, um den *value* eines *current states* zu bestimmen. Nachdem der *current state value* bestimmt wurde, ändert der *agent* seine *policy* dahingehend ab, sodass er diesen *value* maximiert.

#### 2.6.4.1 Policy

Man unterscheidet zwischen *deterministic* (Glg. 2.42) und *stochastic policies* (Glg. 2.41) [14, Kap. 2]. Mit deterministisch ist gemeint, dass jedem Zustand eine optimale Aktion zugeteilt werden kann, die nur vom aktuellen Zustand abhängt. Der Begriff stochastisch wird verwendet um auszudrücken, dass Aktionen zunächst in einen Verlust resultieren können, jedoch auf lange Sicht den Gesamtgewinn erhöhen. Zu Gunsten welcher Aktion die Entscheidung fällt, wird durch die *policy* ausgedrückt (Glg. 2.40). Der Sachverhalt kann, wie in Abb. 2.35 dargestellt, grafisch veranschaulicht werden.

$$\pi = p(a|s) \quad (2.40)$$



**Abbildung 2.35:** Wahl einer Aktion mittels *policy*

$$\{(\pi(a_0|s), \pi(a_1|s), \dots, \pi(a_n|s)) : \pi(a_i|s) \in [0, 1[, \sum_{i=0}^n \pi(a_i|s) = 1\} \quad (2.41)$$

$$\{(\pi(a_0|s), \pi(a_1|s), \dots, \pi(a_n|s)) : \pi(a_i|s) \in \{0, 1\}, \sum_{i=0}^n \pi(a_i|s) = 1\} \quad (2.42)$$

Die Wahrscheinlichkeit, dass ein Zustand und eine Belohnung eintritt, wenn der aktuelle Zustand und die gewählte Aktion bekannt sind, wird durch die *transition dynamic* beschrieben (Glg. 2.43) [14, Kap. 2]. Dabei entspricht  $S_t$  und  $S_{t-1}$  der Menge aller möglichen Zustände,  $A_{t-1}$  entspricht der Menge aller möglichen Aktionen. Gesprochen bedeutet die Formel: Die Wahrscheinlichkeit, dass im nächsten Schritt die Umgebung den Zustand  $s'$  annimmt und daraus eine Belohnung  $r$  resultiert, wenn der letzte Zustand  $s$  war und die Aktion  $a$  ausgeführt wurde. Die Formel lässt sich auch wie in Abb. 2.36 gezeigt grafisch darstellen, man spricht hierbei von einem sogenannten *backup diagram*.

$$p(s', r) = \mathbb{P}(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a) \quad (2.43)$$

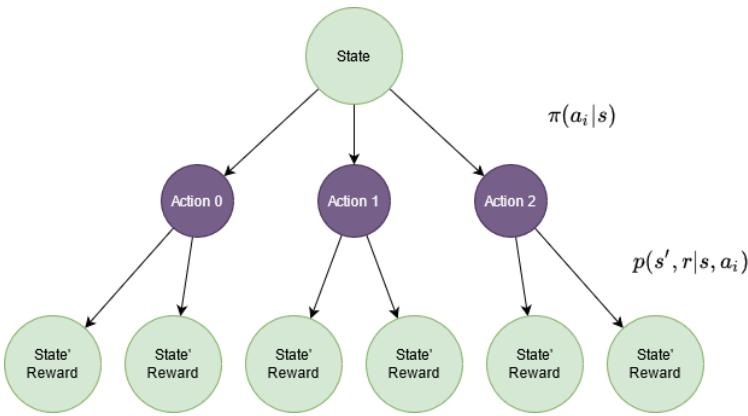


Abbildung 2.36: Backup diagram

#### 2.6.4.2 Rewards und Exploration Versus Exploitation Dilemma

Um den Effekt der Belohnung auf das Anlernen der KI zu steuern, wird ähnlich wie bei der Zinsrechnung ein *discount factor* eingeführt [14, Kap. 2]. Der *discount factor* gibt an, wie sich der Wert einer Belohnung mit fortgeschrittener Zeit verändert. So wie in der Finanzwelt ein Geldbetrag sofort mehr wert ist als der selbe Geldbetrag später, so ist auch beim Lernen eine schnelle Belohnung einer späteren oftmals vorzuziehen. Daher werden spätere Belohnungen anders gewichtet als die sofortigen. Bei kontinuierlichen Aufgaben könnte außerdem die Summe der Belohnungen ins Unendliche gehen, was zu numerischen Problemen führt. Wählt man nun den *discount factor*  $\gamma = 0$ , so wird die KI sehr kurzsichtig, was Belohnungen betrifft. Wählt man hingegen  $\gamma = 1$ , so wird sich beim Training eine gewisse Weitsicht zeigen.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T \quad (2.44)$$

Durch eine gegebene *policy* durchläuft der *agent* gewisse Zustände, deren Wertigkeiten er festzustellen versucht. Aufgrund der festgestellten Werte ändert er die *policy* zu einer besseren. Nun können jedoch Zustände, die der *agent* nie beobachten konnte, aber in der Umgebung existieren, nicht berücksichtigt werden. Hier spricht man vom sogenannten *exploration versus exploitation dilemma* [14, Kap. 2, 9]. Eine Möglichkeit, dem Effekt des eingeschränkten Sichtfeldes zu entgehen, ist die Ausführung einer zufälligen Aktion, die sich nicht nach der *policy* richtet. Wie soll der Entscheidungsprozess für diese Aktionen jedoch aussehen? Hierfür wurde unter anderem die *epsilon greedy policy* entwickelt (Glg. 2.45).

$$\pi(a|s) = \begin{cases} a = \underset{a}{\operatorname{argmax}} Q(s, a) & 1 - \epsilon \\ a = \underset{a}{\operatorname{rand}} Q(s, a) & \epsilon \end{cases} \quad (2.45)$$

Bei der *epsilon greedy policy* führt der *agent* die Aktion mit dem höchsten q-Wert mit der Wahrscheinlichkeit von  $1 - \epsilon$  aus. Eine zufällige Aktion wird mit der Wahrscheinlichkeit von  $\epsilon$  ausgeführt und ist auf alle Aktionen gleich aufgeteilt, um zu gewährleisten, dass der *agent* nicht-maximierende Aktionen erforscht. Dieser Sachverhalt ist im Allgemeinen unter dem Begriff *off-policy* bekannt. Der Parameter  $\epsilon$  ist ein weiterer *hyperparameter* und kann während des Trainings verändert werden. Für die *exploration* können auch andere *policies*, wie *upper confidence bound* oder *Thompson sampling*, verwendet werden.

#### 2.6.4.3 Markov Process

Der *Markov process* [14, Kap. 2] wird in 3 Bereiche unterteilt: *Markov chain*, *Markov reward process* und *Markov decision process*. All diese Prozesse beinhalten den Zufall als gemeinsame Konstante. Dieser Zufall, beschrieben durch Wahrscheinlichkeitswerte, bleibt dabei unverändert und die Prozesse sind somit unabhängig von der Vergangenheit. In folgenden Absätzen, werden die 3 Prozesse näher anhand eines kleinen Beispiels betrachtet.

#### Markov Chain

Die *Markov chain* beschreibt die Wahrscheinlichkeit für einen Übergang in einen anderen Zustand. Betrachtet wird hierfür ein kurzes Beispiel: Abb. 2.37 zeigt vier Zustände und die Transitionswahrscheinlichkeiten  $p$ , mit der man in die jeweiligen Zustände wechselt. Die vier Zustände sind:  $s_0$  entspricht dem Startzustand,  $s_1$  und  $s_2$  entsprechen vom Anfangszustand erreichbare weitere Zustände und  $s_3$  repräsentiert den Endzustand.

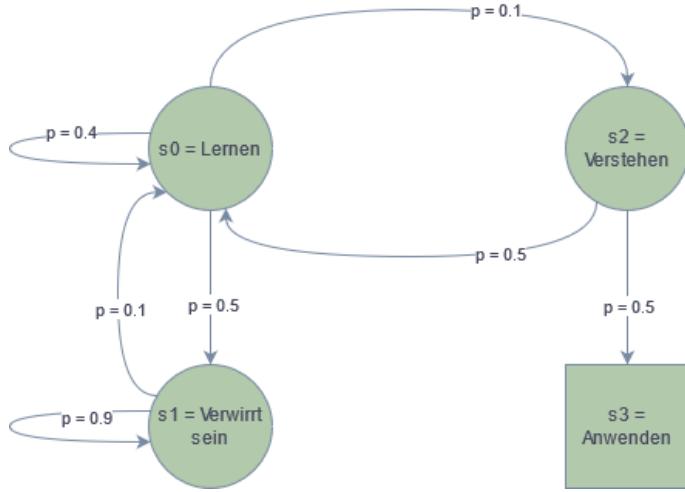


Abbildung 2.37: Beispiel einer *Markov chain*

Dieser Sachverhalt lässt sich auch mathematisch beschreiben (Glg. 2.46). Die Wahrscheinlichkeit für eine Transition hängt hierbei nur vom vorhergehenden Zustand ab. Somit kann die Wahrscheinlichkeit ausgedrückt werden als: Die Wahrscheinlichkeit das der Zustand  $s'$  eingenommen wird, wenn der momentane Zustand  $s$  entspricht. Diese Wahrscheinlichkeiten können in eine Transition-Wahrscheinlichkeit-Matrix zusammengefasst werden (Glg. 2.47).

$$p_i = \mathbb{P}(S_{t+1} = s' | S_t = s) \quad (2.46)$$

$$\mathbf{P} = \begin{pmatrix} 0.4 & 0.5 & 0.1 & 0 \\ 0.1 & 0.9 & 0 & 0 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.47)$$

In diesem Beispiel handelt es sich um eine *periodic Markov chain*, mit einem Endzustand. Da sich der Endzustand, bei Start im Anfangszustand, immer nach einer endlichen Reihe an Transitionen einstellen muss, wird hier im Allgemeinen von einer Episode (*episode*) gesprochen. Verschiedene Episoden könnten wie folgt aussehen:

*Episode 1 :*  $s_0 \rightarrow s_0 \rightarrow s_1 \rightarrow s_1 \rightarrow s_1 \rightarrow s_1 \rightarrow s_1 \rightarrow s_0 \rightarrow s_2 \rightarrow s_3$

*Episode 2 :*  $s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_2 \rightarrow s_0 \rightarrow s_1 \rightarrow s_0 \rightarrow s_2 \rightarrow s_3$

### Markov Reward Process

Um den *Markov reward process* zu veranschaulichen, wird auf das Beispiel von *Markov chain* zurückgegriffen. Nun werden in diesem Beispiel Belohnungen für eine Transition eingeführt (Abb. 2.38). Somit lässt sich ein Wert für die einzelnen Zustände bestimmen (Glg. 2.48).

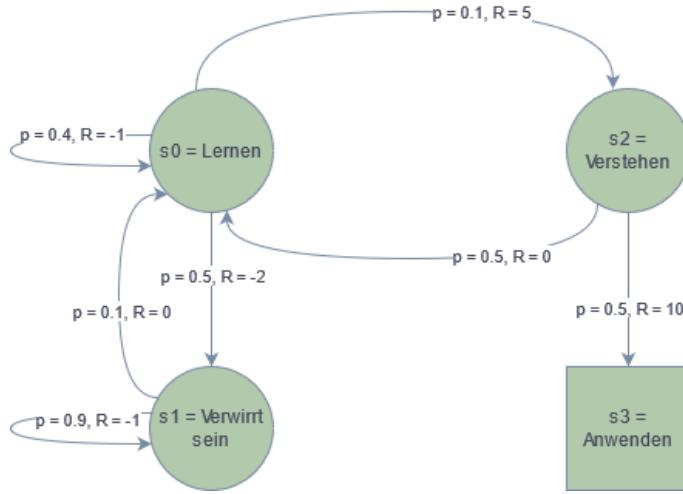


Abbildung 2.38: Beispiel *Markov reward*

$$v(s) = \mathbb{E}[G_t | S_t = s] \quad (2.48)$$

Der Wert eines Zustandes ergibt sich aus dem Erwartungswert  $\mathbb{E}$  von  $G_t$  (Glg. 2.44), wenn der Zustand  $s$  vorherrscht. Der Erwartungswert wird aber in der Praxis durch einen Mittelwert über viele Episoden ersetzt (*Monte Carlo simulation*). Die Transitions-Wahrscheinlichkeiten und die *rewards* des Beispiels in Glg. 2.48 eingesetzt ergibt:

$$v(s_0) = 0.1 * 5 + 0.5 * (-2) + 0.4 * (-1) = -0.9$$

$$v(s_1) = 0.9 * (-1) + 0.1 * 0 = -0.9$$

$$v(s_2) = 0.5 * 0 + 0.4 * 10 = 5$$

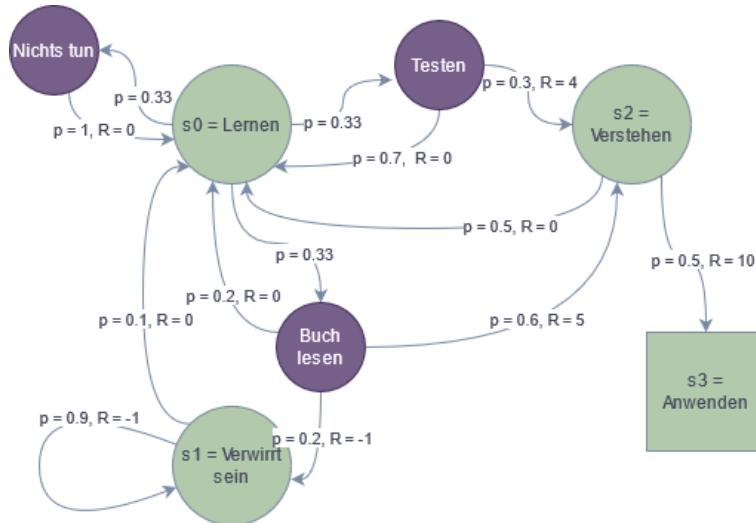
### Markov Decision Process

Betrachtet wird nun den *Markov decision process* anhand des Beispiels aus *Markov chain* und *Markov reward process*. Da Zustände nicht immer nach einer Laune der Natur wechseln, sondern normalerweise durch Aktionen beeinflusst werden, muss

## 2 Recherche

---

man jedem Zustand diese möglichen Aktionen hinzufügen. Zum Beispiel kann man folgende Aktionen für den Zustand  $s_0$  definieren:  $a_0$  Nichts tun,  $a_1$  Buch lesen und  $a_2$  Testen (Abb. 2.39). Wird für die Aktion  $a_0$  entschieden, dann resultiert dies zwar weder in einer Belohnung noch in einer Bestrafung, jedoch zieht man damit die Episode nur in die Länge. Dies hat negativen Einfluss auf  $G_t$ . Wird die Aktion  $a_2$  gewählt, beträgt die Chance auf Transition in den Zustand  $s_2$  30% und man wird außerdem belohnt. Auch könnte die Aktion  $a_1$  gewählt werden. Damit hätte man sogar eine Chance von 60% den Zustand  $s_2$  zu erreichen und dem Ziel näher zu rücken. Jedoch besteht auch eine 20% Wahrscheinlichkeit, dass man in den Zustand  $s_1$  wechselt und dort Minuspunkte sammelt. Welche Aktionen sollen nun also ausführlich werden? Somit landet man wieder bei der Beschreibung einer *policy*, wie sie in Formel 2.43 angegeben wurde. Daraus ergibt sich nun ein Wert für den jeweiligen Zustand, der abhängig von der verwendeten *policy* ist (Glg. 2.49).



**Abbildung 2.39:** Beispiel *Markov process*

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \quad (2.49)$$

Der Wert eines Zustandes ergibt sich nun aus dem Erwartungswert  $E_\pi$  von  $G_t$  (Glg. 2.44) und ist von der gewählten *policy* abhängig. Ein weiteres Konzept, neben der *value function*, ist die *action value function* (Glg. 2.50). Man stelle sich vor, dass es dem *agent* zum Zeitpunkt  $t$  freisteht, welche Aktion dieser ausführt und bedenkt, dass anschließend, für weitere Zeitpunkte, die *policy* eingehalten werden muss.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \quad (2.50)$$

#### 2.6.4.4 Bellman-Gleichung

In diesem Kapitel wird mithilfe der *value function* (Glg. 2.49) und der *discounting function* (Glg. 2.44) die Bellman-Gleichung hergeleitet [14, Kap. 2], welcher im RL-Bereich eine wesentliche Rolle zukommt. Zu Beginn werden die Gleichungen hier nochmals angeschrieben.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

Die *discounting function* kann umgeschrieben werden zu:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots + \gamma^{T-2} R_T)$$

Führt man noch  $t' = t + 1$  als neue Abkürzung ein, so erhält man für den Klammerausdruck:

$$G_t = R_{t'+1} + \gamma R_{t'+2} + \gamma^2 R_{t'+3} + \dots + \gamma^{T'-1} R_{T'}$$

Substituiert man diesen Sachverhalt in die ursprüngliche Gleichung erhält man weiter:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Dieser Sachverhalt kann wiederum in die *value function* eingesetzt werden:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s]$$

Der Erwartungswert  $\mathbb{E}_\pi$  bildet sich über die Summe aller möglichen Aktionen  $a$ , welche der *agent* in Zustand  $s$  tätigen kann und über alle weiteren Zustände, in die die Umgebung den *agent* überführt, welche definiert sind durch die *transition function*  $p(s', r | s, a)$ . Das Einsetzen dieser Definition für den Erwartungswert (Glg. 12.1) führt zur erweiterten Form von  $v_\pi(s)$ , der Bellman-Gleichung.

$$v_\pi(s) = \sum_a \left( \pi(a | s) \sum_{s', r} \left( p(s', r | s, a) (r + \gamma v_\pi(s')) \right) \right) \quad (2.51)$$

Die Gleichung kann nun folgendermaßen interpretiert werden: Der Wert eines Zustandes  $s$  ist der Mittelwert (eine gewichtete Summe) über alle *rewards* und *state values* der folgenden Zustände  $s'$ . Die Mittelwertbildung geschieht über die *policy* bei Ausführung einer Aktion  $a$  im Zustand  $s$ , gefolgt von der Transitionswahrscheinlichkeit  $p(s', r|s, a)$  basierend auf das *state action pair* ( $s, a$ ) (Abb. 2.40). Die Bellman-Gleichung zeigt die rekursive Natur bei der Verlinkung vom *state value* des aktuellen Zustandes mit dem *state value* des Folgezustandes (*bootstrapping*). Eine ähnliche Beziehung besteht für die *action value function* (Glg. 2.50). Mit:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_t + \gamma G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

erhält man:

$$q_\pi(s, a) = \sum_{s', r} \left( p(s', r|s, a)(r + \gamma v_\pi(s')) \right)$$

Es kann auch eine Beziehung zwischen  $v_\pi(s)$  und  $q_\pi(s, a)$  angegeben werden, da beide durch die *policy*  $\pi(a|s)$  verknüpft sind. Daraus ergibt sich für  $v_\pi(s)$ :

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \quad (2.52)$$

Dieser Sachverhalt lässt sich nun wieder in die Gleichung von  $q_\pi(s, a)$  einsetzen und wir gelangen zur Bellman-Gleichung für  $q_\pi(s, a)$ .

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a) \left( r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right) \quad (2.53)$$

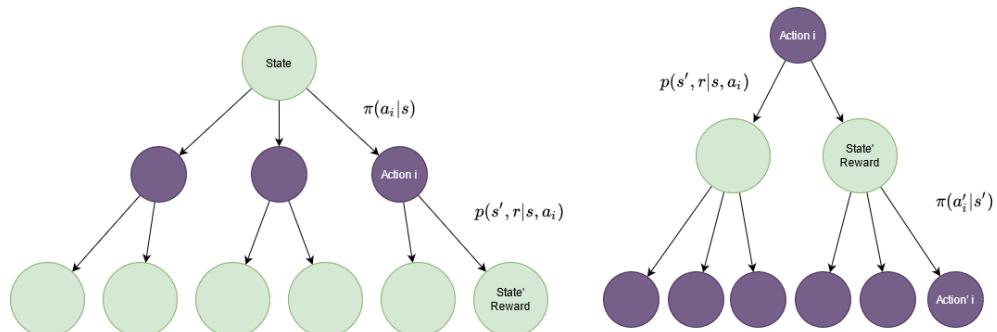


Abbildung 2.40: Backup diagram für *state* und *action values*

#### 2.6.4.5 Lösen der Bellman-Gleichung

Ein RL-Problem zu lösen heißt, eine *policy* zu finden, mit der die *state value function* maximiert wird. Wenn der *agent* die optimale *policy* befolgt, dann wird er im Zustand  $s$  jene Aktion wählen, welche auch die *action value function*  $q(s, a)$  maximiert. Im folgenden Beispiel (abgeleitet von [14, Kap. 2]) werden diese Zusammenhänge mathematisch beschrieben:

$$v(s)^* = \max_{\pi} v_{\pi}(s)$$

$$v(s)^* = \max_a q_{\pi}(s, a)$$

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

Nun wird an dieser Stelle das Beispiel aus *Markov process* fortgesetzt. Man kann alle Kombinationen aus: aktuellenm Zustand, Aktion, Belohnung und neuem Zustand wie in Tabelle 2.5 zusammenfassen.

**Tabelle 2.5:** Alle Möglichen (s,a,s',r)-Kombinationen aus Abb. 2.39

State (s)	Action (a)	New State (s')	Reward (r)	Transision $p(s', r   s, a)$
$s_0$ Lernen	$a_0$ Nichts tun	$s_0$ Lernen	0	1.0
$s_0$ Lernen	$a_1$ Testen	$s_0$ Lernen	0	0.7
$s_0$ Lernen	$a_1$ Testen	$s_2$ Verstehen	4	0.3
$s_0$ Lernen	$a_2$ Buch lesen	$s_0$ Lernen	0	0.2
$s_0$ Lernen	$a_2$ Buch lesen	$s_1$ Verwirrt sein	-1	0.2
$s_0$ Lernen	$a_2$ Buch lesen	$s_2$ Verstehen	5	0.6
$s_2$ Verstehen	-	$s_0$ Lernen	0	0.5
$s_2$ Verstehen	-	$s_3$ Anwenden	10	0.5
$s_1$ Verwirrt sein	-	$s_2$ Lernen	0	0.1
$s_1$ Verwirrt sein	-	$s_1$ Verwirrt sein	-1	0.9

Mithilfe dieser Tabelle können nun die *state values* der Zustände, nach obiger Rechenvorschrift, bestimmt werden. Hierbei ist der *discount factor* noch unbestimmt.

$$v(s_0)^* = \max \begin{cases} a_0 : & 1(0 + \gamma v(s_0)^*) \\ a_1 : & 0.7(0 + \gamma v(s_0)^*) + 0.3(4 + \gamma v(s_2)^*) \\ a_2 : & 0.2(0 + \gamma v(s_0)^*) + 0.2(-1 + \gamma v(s_1)^*) + 0.6(5 + \gamma v(s_2)^*) \end{cases}$$

$$v(s_1)^* = 0.1(0 + \gamma v(s_0)^*) + 0.9(-1 + \gamma v(s_1)^*)$$

$$v(s_2)^* = 0.5(0 + \gamma v(s_0)^*) + 0.5(10 + \gamma v(s_3)^*)$$

Um eine einfache Lösung des Problems zu erhalten, kann für  $\gamma$  null eingesetzt werden. Somit wäre ein kurzsichtiges Verhalten des *agents* zu erwarten, da somit nur die Aussicht auf schnelle Belohnung berücksichtigt wird. Die Gleichungen sind somit nicht mehr rekursiv und besitzen von Anfang an einen konstanten Wert.

**Tabelle 2.6:** Lösen von MDP mit  $\gamma = 0$

$v^*$	t	t+x	t+y
$v(s_0)^*$	$a_1 \rightarrow 0.12$	$a_1 \rightarrow 0.12$	$a_1 \rightarrow 0.12$
$v(s_1)^*$	-0.9	-0.9	-0.9
$v(s_2)^*$	5	5	5

Aus dieser Rechnung kann der Schluss gezogen werden, dass es immer die Aktion ‚Testen‘ zu wählen gilt, wenn nur schnelle Belohnungen gewünscht sind. An dieser Stelle sei angemerkt, dass die Aktion  $a_1$  knapp die beste Wahl darstellt, Aktion  $a_2$  würde einen *state value* von 0.1 ergeben. Würde man versuchen das System für  $\gamma \neq 0$  zu lösen, würde man ungleich feststellen, dass die rekursive Form der Gleichungen zu Problemen führt. Daher werden hierfür numerische Algorithmen (Abb. 2.41) benötigt.

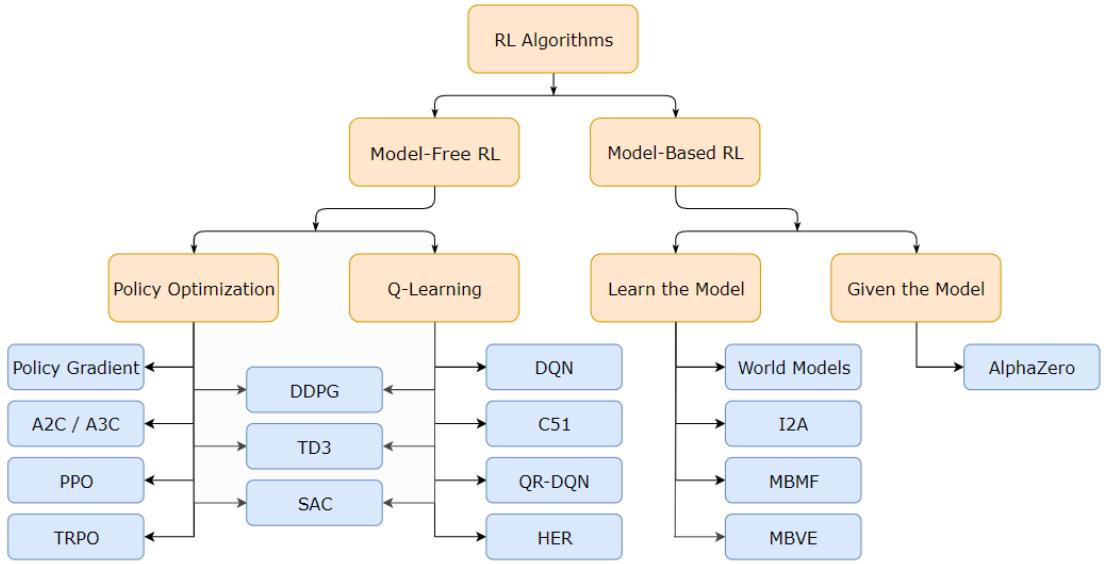


Abbildung 2.41: Algorithmen für RL [30]

#### 2.6.4.6 Temporal Difference Learning

Temporal-Difference Learning (TDL) [14, Kap. 4] kombiniert die Vorteile von *dynamic programming* und *Monte Carlo simulation*. Hierfür wird *bootstrapping* aus *dynamic programming* und *sample-based* aus der *Monte Carlo*-Methode verwendet. Glg. 2.54 zeigt das Schema für das Update der *state-values*.

$$V(s) \stackrel{!}{=} V(s) + \alpha [R + \gamma V(s') - V(s)] \quad (2.54)$$

Der momentane Schätzwert für die kumulierten Belohnungen, resultierend aus dem Zustand  $s$ , ergibt sich nun durch *bootstrapping* aus dem Schätzwert für den nachfolgenden Zustand  $s'$ , gewonnen durch mehrere Durchläufe (*sample-based*). Würde man den Term  $R + \gamma V(s')$  mit  $G_t$  (Glg. 2.44) ersetzen, hätte man einen reinen *Monte Carlo*-Ansatz verfolgt. Das Update verschiebt den geschätzten Wert von  $V(s)$  zu  $V(s) + \alpha \delta_t$  wobei  $\delta_t$  dem *temporal difference error* (Glg. 2.55) entspricht.

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \quad (2.55)$$

### 2.6.4.7 Deep Q-Learning

Deep-Q Learning (DQL) [14, Kap. 6] ist eine *model-free*, *off-policy*, *temporal-difference* Lernmethode im Bereich RL. Diese Methode beinhaltet einen *agent*, welcher einer gewissen *policy* folgt und Erfahrungen sammelt. Diese bestehen aus einem Tupel der Form: (*current state*, *action taken*, *resulted reward*, *next state*, *end of game?* *J/N*). Der *agent* verwendet anschließend diese Erfahrungen und die Bellman-Gleichung, um in einer iterativen Schleife die optimale *policy* zu bestimmen, welche die *value function* jeden Zustandes maximiert. Die Optimierung erfolgt durch die Update-Regel gegeben durch Glg. 2.56. Die Entnahme von Erfahrungen, gespeichert im sogenannten *replay-buffer*, kann auf verschiedene Weisen erfolgen. Meist wählt man einfachheitshalber eine Normalverteilung für die Auswahl, jedoch könnten diese auch priorisiert werden [31].

$$Q(s, a) \stackrel{!}{=} Q(s, a) + \alpha \left[ R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \quad (2.56)$$

Um diese Methode für *environments* mit großer Anzahl an Zuständen anwenden zu können, wird der *q-value* einer *policy* durch eine Funktionsapproximation (*deep learning*) geschätzt  $\tilde{q}(s, a, w) \approx q_\pi(s, a)$ . Somit ist der *q-value* von den Gewichtungsfaktoren abhängig und man spricht vom DQL. Unter Berücksichtigung linearer Approximation und dem BGD-Verfahren aus Kapitel 2.6.1 kann nun eine Update-Vorschrift für die Gewichtungsfaktoren bestimmt werden (Glg. 2.57).

$$w_{t+1} = w_t + \alpha \frac{1}{n} \sum_{i=1}^n \left[ r_i + \gamma \max_{a_i} \tilde{q}(s'_i, a'_i, w^{target}) - \tilde{q}(s_i, a_i, w_t^{online}) \right] \nabla_w \tilde{q}(s_i, a_i, w_t^{online}) \quad (2.57)$$

Index  $t$  deutet hierbei auf den Optimierungsschritt hin und Index  $i$  auf die Position im *batch*. Die Gewichtungsfaktoren  $w^{target}$  bleiben aus numerischen Gründen intervallweise konstant und wird am Intervallende mit  $w_t^{online}$  überschrieben.

### Double-Deep-Q-Learning

Aus Gleichung 2.57 ergibt sich, dass für die Bestimmung der maximierenden *action* und für die Bestimmung des *q-values* das selbe Netzwerk verwendet wird. Vereinfacht angeschrieben entspricht dies:

$$Y^{DQN} = r + \gamma \max_{a'} \tilde{q}(s', a', w_t^{target})$$

Die Gleichung kann auch äquivalent angegeben werden zu:

$$Y^{DQN} = r + \gamma \tilde{q}(s', \operatorname{argmax}_{a'} \tilde{q}(s', a', w_t^{\text{target}}), w_t^{\text{target}})$$

Nun nehmen wir zuerst die maximierende *action* und bestimmen anschließend den *q-value*. Dies ist äquivalent mit der zuvor gewählten Vorgehensweise, direkt den maximalen *q-value* zu bestimmen. Nun erkennt man, dass für die Bestimmung der besten *action* und für die Berechnung des *q-values* die gleichen Gewichtungsfaktoren verwendet werden. Dies führt zum sogenannten *maximization bias*, welches von [32] untersucht wurde. Die Autoren des Papers empfehlen eine Vorgehensweise, die sie *double DQN* nennen, wobei die Gewichte für die Bestimmung der besten *action*  $\operatorname{argmax}_{a'} \tilde{q}(s', a')$  mit  $w_t^{\text{online}}$  gewählt werden und die Gewichte für die Bestimmung des *q-values* mit  $w_t^{\text{target}}$ .

$$Y^{DQN} = r + \gamma \tilde{q}(s', \operatorname{argmax}_{a'} \tilde{q}(s', a', w_t^{\text{online}}), w_t^{\text{target}})$$

### Dueling-Double-Deep-Q-Learning

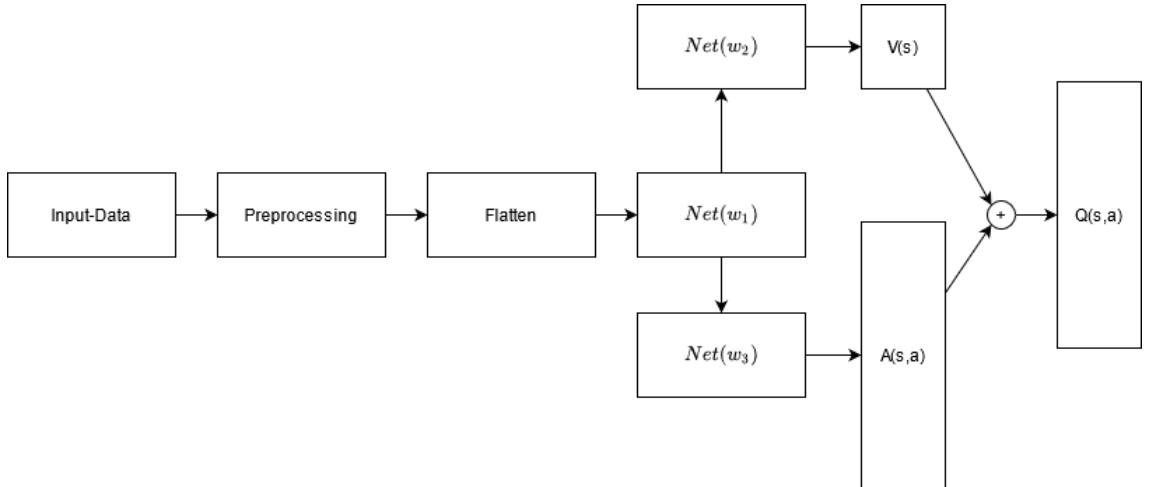
Bis jetzt nahmen die Netzwerke immer einen Zustand als Input und gaben *q-values*  $Q(s, \mathbf{a})$  für alle möglichen Aktionen  $\mathbf{a}$  zurück. Oft haben manche Aktionen in einem bestimmten Zustand jedoch keine wesentlichen Auswirkungen. Man stelle sich ein Auto auf einer leeren Fahrbahn vor, solange keine Hindernisse auftauchen, ist eine kleine Änderung der Richtung oder der Geschwindigkeit bedeutungslos. Somit würden alle diese Aktionen in ähnliche *q-values* resultieren. Nun stellt sich die Frage, ist es möglich, den Durchschnittswert eines Zustandes und den Vorteil einer bestimmten Aktion über diesen Durchschnittswert zu trennen? Diese Vorgehensweise wurde von [33] untersucht. Die Autoren zeigten, dass sich das Verhalten des Netzwerks durch eine solche Vorgehensweise wesentlich verbessern lässt, besonders wenn die Anzahl an möglichen Aktionen steigt. Hierfür wurde ein neuer Parameter  $A$  eingeführt, der sogenannte *advantage* (Glg. 2.58).

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \quad (2.58)$$

$V_\pi(s)$  misst den Wert in einem gewissen Zustand zu sein,  $Q_\pi(s, a)$  gibt den Wert an, eine bestimmte Aktion in diesem Zustand zu wählen. Die Differenz der beiden Werte kann nun als Vorteil interpretiert werden. Die Autoren des Papers entwickelten ein Netzwerk, welches den Zustand am Eingang in zwei Datenflüsse aufteilt.

Einer zur Bestimmung der *state-values*  $V(s)$  und der andere zur Bestimmung der *advantages*  $A(s, a)$ . Abschließend werden beide Datenflüsse wieder zusammengeführt um die *q-values* zu erhalten. Für eine verbesserte Stabilität des Verfahrens haben die Autoren noch den Mittelwert von *advantage* von den Endknoten  $Q(s, a)$  abgezogen (Glg. 2.59). Eine Veranschaulichung des Datenflusses für dieses Netzwerk wird in Abb. 2.42 dargestellt. Im Block ‚Preprocessing‘ verbergen sich Operationen die durch *convolution-layers* oder *pooling-layers* implementiert werden. Als ‚Input-Data‘ könnten hierbei z. B. *raw pixel* dienen.

$$\hat{Q}(s, a, w_1, w_2, w_3) = \hat{V}(s, w_1, w_2) + \left( \hat{A}(s, a, w_1, w_3) - \text{mean}(\hat{A}) \right) \quad (2.59)$$



**Abbildung 2.42:** Dueling-Double-Deep-Q Network (DQN)

## 3 Konzeptentwicklung

Für die Realisierung der Zielsetzungen wird ein minimalistisches Tool für die Anlagenkonfiguration in Python entwickelt. Hierfür wird im Speziellen die in Python standardmäßig inkludierte Bibliothek TKINTER herangezogen. Mithilfe dieser Bibliothek soll eine Interaktion mit dem Anwender stattfinden, um dem Programm die nötigen Konfigurationsparameter zu übergeben. Des Weiteren soll die Software eine Simulation beinhalten, um den konfigurierten Prozess zu veranschaulichen. In dieser Simulation soll auch die Funktionsweise des erstellten neuronalen Netzwerks ersichtlich sein, da die Umgebung speziell für diesen Grund entwickelt wird.

Zu Beginn wird der *workflow* aus Sicht des Bauteiles vorgestellt, woraus resultierend sich Klassen ableitet lassen, die eine entsprechende Realisierung des Prozesses ermöglichen. Im Kontext zu Ausfallsicherheit und Flexibilität werden im darauf folgendem Schritt die entsprechenden Attribute, Methoden und Zustände für die jeweiligen Klassen definiert. Des Weiteren wird eine *superclass* entwickelt, die Attribute und Methoden beinhaltet, welche von allen Klassen benötigt werden.

Im Abschnitt 3.4 wird die Vorgehensweise zur Entwicklung des Parkettierprozesses abgewickelt. Hierfür wird der Anwendungsbereich auf zylindrische Bauteile und Bauteile in Form von Tetrominos [34] eingegrenzt. Tetrominos sind aus vier Quadranten zusammenfügbar Geometrien, wie sie manchen evtl. aus dem Spiele-Klassiker TETRIS bekannt sind.

Im letzten Abschnitt wird das Konzept für die Steuerungsinteraktion vorgestellt. Die erstellte Software soll mit einer SPS von SIEMENS (S7) kommunizieren können, hierfür wird das im Kapitel 2.3 vorgestellte Kommunikationsprotokoll OPC UA verwendet. Für die Validierung des Konzepts steht eine Trainingszelle von KUKA zur Verfügung. Da das System Trainingszelle primär nur aus einem Drehtisch und einem Roboter besteht, wird für die Validierung eine vereinfachte Version der Software angefertigt. Es wird davon ausgegangen, dass sich die Resultate anschließend auf die Vollversion übertragen lassen.

### 3.1 Workflow

Um einen Überblick der Komponenten und deren Attribute und Methoden zu erhalten, wird ein Bauteil gedanklich durch den Anlagenprozess geschickt, wie es auch im Aktivitätsdiagramm (Abb. 2.20) aus Kapitel 2.4.7 angedeutet wurde. Eine Systemgrenze stellt dabei die Beladung des Fördersystems dar, in diesem Fall ein Förder-

### 3 Konzeptentwicklung

---

band. Zu Beginn muss das Bauteil am Förderband erkannt werden, hierfür benötigt man eine Scanvorrichtung, die eventuell mit Bildverarbeitung und Quick Response (QR)-Scannung das Bauteil identifiziert und dessen Lage im Raum vollständig beschreibt. Im Idealfall lässt sich dieser Identifizierungsprozess in die Fördereinheit integrieren. Somit müsste die Fördereinheit nur die Information zur Verfügung stellen, welche Position vom Roboter angefahren werden soll und um welches Bauteil es sich handelt. Die Position hat im allgemeinen sechs Freiheitsgrade und lässt sich mit einem *struct* abbilden. Das Bauteil müsste mehrere Informationen mitbringen, vor allem aber eine Greiftaktik und den dazu benötigten Greifer. Als Mensch erkennt man eine geeignete Greiftaktik intuitiv, dies ist jedoch bei Maschinen nur sehr schwer zu realisieren und bildet ein eigenes Forschungsthema. Neben der Greiftaktik soll das Bauteil noch eine eindeutige ID und eine Masse zur Verfügung stellen. Die ID dient zur Identifizierung im Logistiksystem und die Masse kann verwendet werden, um nach dem Greifen des Bauteils festzustellen, ob dies auch erfolgreich durchgeführt wurde.

Das gedanklich verfolgte Bauteil steht jetzt für die Abholung bereit. Die für die Abholung benötigten Informationen werden vom Fördersystem und indirekt vom Bauteil selbst zur Verfügung gestellt. Fordert eine Kontrolleinheit eine Bestückung von Bauteilen an, wird der Handlingsrobotik mitgeteilt, das vom Fördersystem zur Verfügung gestellte Bauteil abzuholen. Hierfür wird der Handlingsrobotik noch eine Zielposition mitgegeben. Es beginnt die Transition des Bauteils Richtung Prozessanlage oder zwischenzeitlich in Richtung einer Parkettierungszone. Die innerhalb des Systems zur Verfügung gestellten Informationen sind das Bauteil und dessen Zielposition. Um bei auftretenden Störungen, wie z. B. einem Stromausfall, das Anfahren der Zielposition zu erleichtern, sollten diese Informationen persistent gespeichert werden, um einen etwaigen Informationsverlust und somit einen unkontrollierten Zustand zu vermeiden.

Die Zielposition wird durch ein vordefiniertes Bestückungslayout, bzw. einer *policy* bestimmt, die im Leitprozess hinterlegt sind. Das Bauteil wird anschließend direkt oder auf Umwegen in der Prozessanlage abgelegt und wartet auf den Start des Prozesses. Der Prozess wird von der Anlage vollständig überwacht und dessen Parameter werden falls benötigt in einer Datenbank aufgezeichnet. Anschließend wird über die Bauteil-ID ein Link zu diesem Eintrag der Datenbank erstellt. Nach Abschluss des Prozesses wird das Bauteil wieder aus der Prozessanlage entfernt und entweder wieder direkt oder mit Zwischenschritt auf ein Fördersystem für den Abtransport gelegt. Hierbei sollten abschließend nochmals alle Bauteile registriert werden, um sicherzustellen, dass der Ablauf erfolgreich durchgeführt werden konnte. Nun befindet sich das Bauteil an der zweiten Grenze des zu steuernden Systems.

## 3.2 Definition der Klassen

Anhand von Kap. 3.1 werden nun Klassen abgeleitet, die den Prozess abbilden können. Hierbei wird die Klassenbezeichnung ‚member‘ und ‚component‘ eingeführt. Der ‚member‘ entspricht einem aktiven Teilnehmer des Prozesses wie dem Förderband oder der Härteanlage. Die Bezeichnung ‚component‘ bezieht sich auf eine passive Komponente wie dem Bauteil oder der Chargierplatte. Die eingeführten Attribute und Methoden sind zur Dokumentation als UML-Klassendiagramme im Quell-Code hinterlegt. Es sei hier angemerkt, dass in die Definition der Attribute und der Methoden viel Aufwand geflossen ist, jedoch aus Platzgründen nicht in die Arbeit aufgenommen wurden. Stattdessen werden an dieser Stelle nur die wesentlichen Klassen vorgestellt. Die definierten Zustände können im Anhang (12.3) eingesehen werden.

- **Plant\_Component:** Die *superclass*, welche Attribute und Methoden enthält, die von sämtlichen Klassen benötigt werden. Hierzu zählt z. B. die Position und Visualisierung.
- **Plant:** Darstellung der Härteanlage. Zeichnet sich durch veränderbare Beladungsplätze aus, welche durch Stapelhöhe und Anzahl der Stapel vorgegeben werden.
- **Robot:** Die Handlingsrobotik, welche den Transport von Bauteilen und Chargierplatten übernimmt.
- **Conveyor:** Ein Förderband, welches Bauteile geradlinig bewegt. Auf dem Förderband befindet sich eine Lichtschranke, die bei Aktivierung das Förderband ausschaltet.
- **Storage:** Die Idee dieser Klasse stammt ursprünglich aus einem Initialisierungsproblem. Bei der Initialisierung müssen die Objekte der Klasse ‚Table‘ einer Raumposition zugeordnet werden. Des Weiteren kann diese Klasse dazu verwendet werden, einen Zwischenspeicher für Chargierplatten zu realisieren.
- **Rotator:** Eine siebte Achse, wie sie bei Robotersystemen häufig zum Einsatz kommt. Sie kann Chargierplatten aufnehmen und eine Rotation um die Hoch-Achse realisieren.
- **Intake:** Eine der Systemgrenzen. Diese Klasse bringt Bauteile in das System ein.
- **Outtake:** Realisiert die zweite Grenze des Systems und entfernt daher Bauteile aus dem System.

- **Table:** Die Chargierplatten, welche Bauteile aufnehmen können und für das Handling benötigt werden.
- **Part:** Das Bauteil, welches über die Systemgrenzen initialisiert und auch wieder gelöscht wird. Wie Eingangs erwähnt, wird im Zuge dieser Arbeit zwischen zylinderförmigen Bauteilen und Bauteilen in Form von Tetrominos unterscheiden. Hierfür wird zur Unterscheidung eine Bauteil-ID eingeführt (Tab. 3.1).

**Tabelle 3.1:** Bauteilklassen und deren ID

Bauteil	Bezeichnung [35]	ID
	Zylinder	1
	Smashboy	2
	Teewee	3
	Rhode Island Z	4
	Cleveland	5
	Hero	6
	Orange Ricky	7
	Blue Ricky	8

## 3.3 Architekturentwurf

In diesem Abschnitt wird der Aufbau der Gesamtapplikation entwickelt. Folgend wird das Modul für die Anlagenkomponenten, des User Interfaces (UI), der Bestückungs- und der Prozessvorschrift, gefolgt vom Modul der Kommunikation vorgestellt. Zuletzt wird aus den Einzelentwürfen ein UML-Diagramm der Gesamtapplikation angefertigt und daraus eine Verzeichnisstruktur für die Programmierung abgeleitet.

### 3.3.1 Modul für die Anlagenkomponenten

Um die in Kap. 3.2 vorgestellten Komponenten möglichst wartungsfreundlich zu gestalten, werden diese in einem Modul „simulation\_members“ zusammengefasst. Hierbei beinhaltet die Klasse „Plant\_Component“ alle Attribute und Methoden, die auch

### 3 Konzeptentwicklung

---

von allen anderen Klassen der Anlagenkomponenten benötigt werden. Die Position der jeweiligen Koordinatensysteme kann ebenfalls in eine eigenen Klasse implementiert werden. Für die Klassen ‚Plant‘, ‚Storage‘ und ‚Conveyor‘ wird noch ein Mechanismus benötigt, der bei Anfrage von einem Objekt der Klasse ‚Robot‘ eine Abholstelle zurückgibt und diese anschließend bis zur Abholung sperrt. Diese Abholstellen werden mit der Klasse ‚Component\_Slot‘ realisiert. Die Darstellung mittels UML ist in Abb. 3.1 ersichtlich.

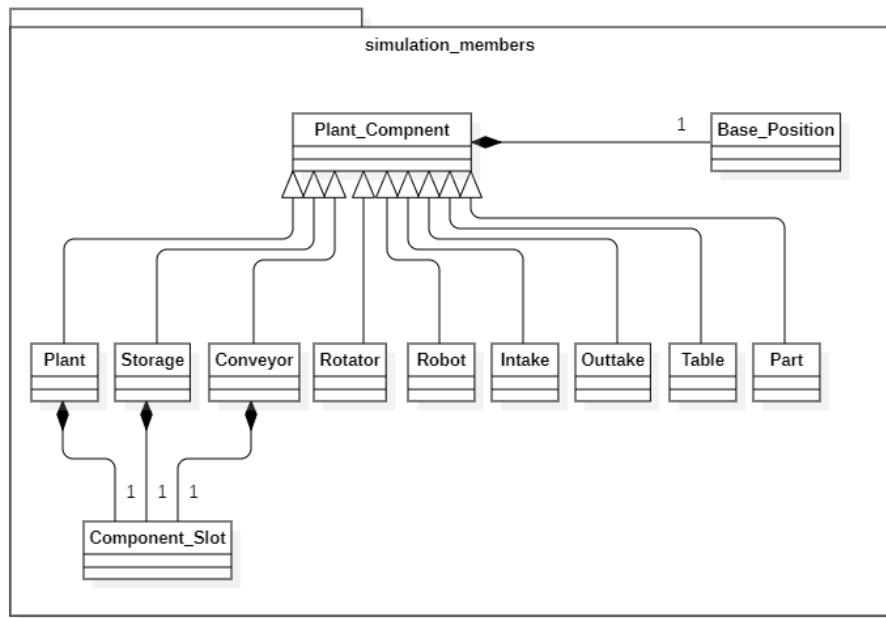


Abbildung 3.1: Modul ‚simulation\_members‘

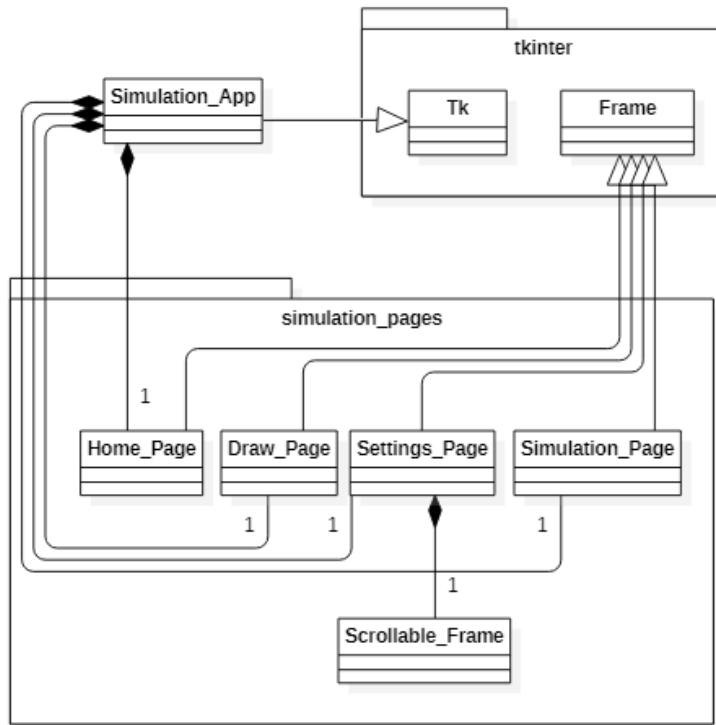
Das in STARUML erstellte Diagramm wird als Fragment exportiert, um zu einem späteren Zeitpunkt in andere Diagramme importiert werden zu können.

#### 3.3.2 Modul für das User-Interface

Für das UI wird die Bibliothek Tkinter verwendet, eine Standardbibliothek in Python. Hierfür wird eine Klasse ‚Simulation\_App‘ angelegt, die von der Klasse ‚tkinter.Tk‘ abgeleitet wird. Des Weiteren beinhaltet die Klasse ‚Simulation\_App‘ die anzuzeigenden Seiten der Anwendung. Diese dafür generierten Seiten werden in einem Modul ‚simulation\_members‘ zusammengefasst und bei der Instanziierung der Anwendung angelegt. Die Zusammenfassung in ein eigenes Modul soll eine mögliche Erweiterung der Applikation benutzerfreundlicher gestalten. Die in dem Modul befindlichen Klassen ‚Home\_Page‘, ‚Draw\_Page‘, ‚Settings\_Page‘ und ‚Simulation\_Page‘ werden von der Klasse ‚tkinter.Frame‘ abgeleitet und ermöglichen die

### 3 Konzeptentwicklung

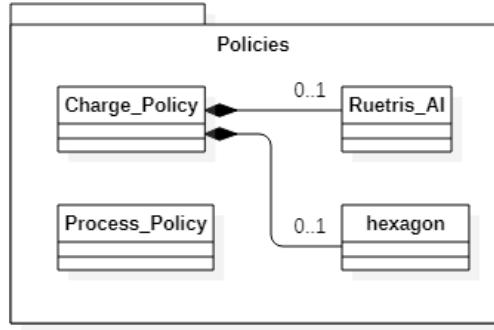
Interaktion mit den Objekten aus dem Modul „simulation\_members“. Das hierfür angelegte UML-Diagramm ist in Abb. 3.2 dargestellt.



**Abbildung 3.2:** Modul „simulation\_pages“

#### 3.3.3 Modul für die Parkettierungsvorschrift

Um den Prozess leiten und die Ablageposition der Bauteile bestimmen zu können, werden Vorschriften benötigt, die vom aktuellen Zustand der Parkettierung oder des Gesamtsystems abhängen. Die Vorschriften sind im Aufbau mit den *policies*, wie sie im Kap. 2.6.4.1 vorgestellt wurden, identisch. Hier wird auch der Kern dieser Arbeit zu einem späteren Zeitpunkt eingebunden, nämlich als „Charge\_Policy“, die Parkettierung von diversen Bauteilen. Diese „Charge\_Policy“ soll abhängig vom aktuellen Beladungszustand, eine Anzahl von erforderlichen Aktionen zurückgeben, welche aus Anweisungen für die drei Freiheitsgrade: Drehwinkel des „Rotators“ und xy-Koordinaten für die Handlingsrobotik, bestehen. Die Klasse „Process\_Policy“ dient zur Steuerung des Gesamtprozesses. Hierfür wird nur eine einzige Methode entwickelt, um zu zeigen, dass eine Steuerung des Gesamtprozesses durch eine vollständige Beschreibung des Systems möglich ist, welche durch die definierten Zustände realisiert wird. Die Darstellung mittels UML ist in Abb. 3.3 ersichtlich.



**Abbildung 3.3:** Modul ‚policies‘

### 3.3.4 Modul für die Kommunikationstreiber

Für die Kommunikation zwischen dem erstellten Programm und den realen Anlagenkomponenten werden Kommunikationstreiber benötigt, welche auch in einem eigenen Modul ‚communication‘ zusammengefasst werden. Da als Testanlage derweilen nur eine Trainingszelle zur Verfügung steht, wird auch nur hierfür ein Treiber vorgesehen. Die Trainingszelle, wie im Kap. 3.5.1 ersichtlich, besteht aus zwei Objekten: einem Roboter und einem Rotator. Der Aufbau der Kommunikation wird im Kapitel 3.5.2 erarbeitet.

### 3.3.5 Aufbau der Gesamtapplikation

In diesem Abschnitt werden die Einzelentwürfe zu einem Gesamtentwurf zusammengefasst, dessen Funktionsweise wie folgt beschrieben wird:

Beim Start des Programms soll ein Objekt der Klasse ‚Settings‘ angelegt werden, um Standardwerte zu laden und etwaige Werte ändern zu können. In diesem Objekt werden während der Laufzeit auch Objekte der als *dataclasses* notierten Klassen erzeugt, welche die Standardwerte für die jeweiligen Klassen enthalten. Anschließend soll die Hauptklasse des Programms instanziert werden, hierfür wird das Objekt der Klasse ‚Settings‘, neben der *superclass* ‚tkinter.Tk‘, mitübergeben. Das Objekt der Klasse ‚Simulation\_App‘ lädt in Folge das Modul ‚simulation\_pages‘ und instanziert jeweils ein Objekt der inbegriffenen Klassen. Das Objekt der Klasse ‚Home\_Page‘ verwaltet die Anzahl der benötigten Komponenten und hat daher eine Abhängigkeit zum Objekt der Klasse ‚Settings‘. Das Objekt der Klasse ‚Draw\_Page‘ instanziert aufgrund der Informationen von ‚Home\_Page‘, die in ‚Settings‘ gespeichert wurden, die jeweiligen Klassen aus dem Modul ‚simulation\_members‘. Mithilfe des Objekts der Klasse ‚Settings\_Page‘ kann auf die Parameter der erzeugten Member-Objekte,

### 3 Konzeptentwicklung

genauer formuliert: den betroffenen *dataclasses*, zugegriffen und beliebige Werte geändert werden. Die getätigten Änderungen können anschließend wieder im Objekt der Klasse ‚Draw\_Page‘ begutachtet werden. Um alle Parameter im Fenster unter zu bringen, wird noch eine Klasse ‚Scrollable\_Frame‘ instanziert, welche die Parameter beherbergt. Ist man mit dem Layout zufrieden, kann das Objekt der Klasse ‚Simulation\_Page‘ verwendet werden. In diesem Fenster wird das erstellte Layout geladen und die entsprechenden Einstellungen für die *policies* gewählt. Wurden die Einstellungen gewählt, wird jeweils ein Objekt der entsprechenden Klasse angelegt. Nun kann das System animiert werden. Des Weiteren soll während der Animation die Möglichkeit bestehen, die Zustände der Anlagenkomponenten mitzuverfolgen.

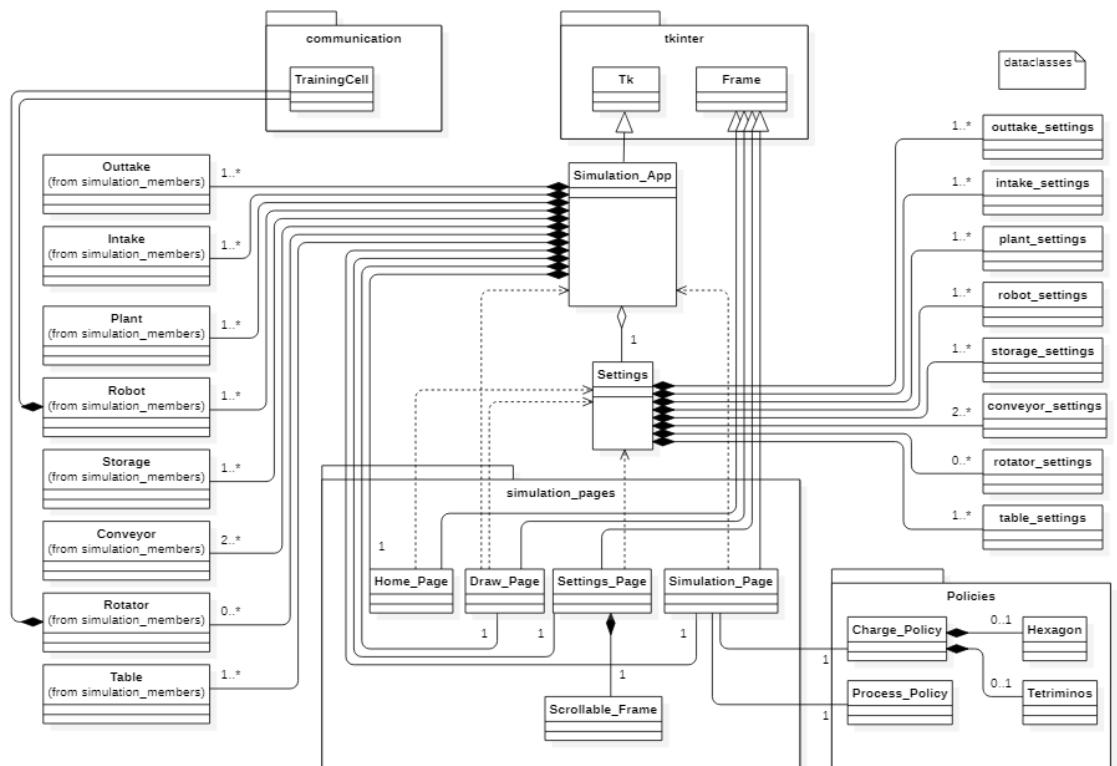


Abbildung 3.4: UML-Diagramm der Gesamtapplikation

Es sei noch erwähnt, dass das Modul ‚simulation\_members‘ als Fragment importiert wurde, daher ist der Modul-*frame*, sowie die Abhängigkeiten innerhalb des Moduls nicht sichtbar. Dies wurde aus Gründen der Übersichtlichkeit so gewählt.

#### 3.3.6 Codegenerierung

Die Diagramme wurden mithilfe der Evaluierungsversion von Staruml generiert. Diese Software bietet die Möglichkeit, ein Code-Skelett und eine Verzeichnisstruktur für

### 3 Konzeptentwicklung

das Projekt zu erstellen (Abb. 3.5). Die Software erzeugt jedoch auch eigene Files für die als *dataclasses* notierten Klassen. Hierbei handelt es sich jedoch um ein Python-Äquivalent zu einem *struct*, daher werden diese Files gelöscht und in das File ‚Setting.py‘ aufgenommen. Des Weiteren wird dem Hauptverzeichnis noch ein File ‚main.py‘ hinzugefügt und der Ordner ‚tkinter‘ gelöscht, da es als externe Bibliothek einfach importiert werden kann. Die somit erhaltene Programmstruktur ist in Abb. 3.6 ersichtlich.

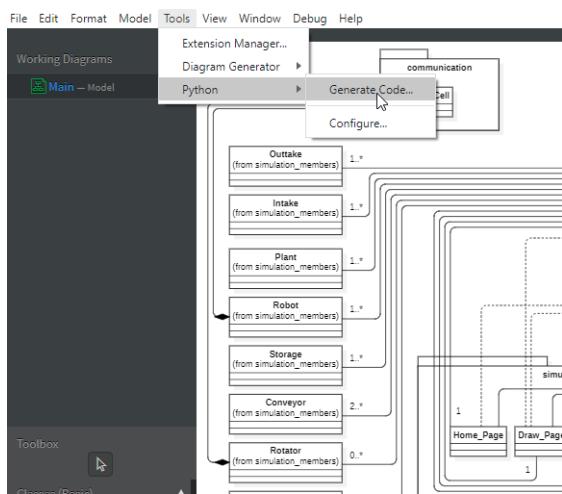


Abbildung 3.5: Codegenerierung UML

```
C:\TEMP\Downloads\code_gen\Model>tree /F /A
Aufzählung der Ordnerpfade für Volume OS
Volumenseriennummer : 8700-7F23
C:
|   Settings.py
|   Simulation_App.py
+---communication
|       TrainingCell.py
|       __init__.py
+---Policies
|       Hexagon.py
|       Process_Policy.py
|       Tetrominos.py
|       __init__.py
+---simulation_members
|       Component_Slot.py
|       Conveyor.py
|       Intake.py
|       Outtake.py
|       Part.py
|       Plant.py
|       Plant_Component.py
|       Robot.py
|       Rotator.py
|       Storage.py
|       Table.py
|       __init__.py
+---simulation_pages
|       Draw_Page.py
|       Home_Page.py
|       Scrollable_Frame.py
|       Settings_Page.py
|       Simulation_Page.py
|       __init__.py
```

Abbildung 3.6: Verzeichnisstruktur

## 3.4 Parkettierungsstrategie

In diesem Kapitel wird auf die Parkettierung der Chargierplatten eingegangen. Prinzipiell wird zwischen zwei Arten von Bauteilen und deren Parkettierungen unterschieden: zylindrische und tetromino-förmige Bauteile. Begonnen wird mit der Betrachtung von zylindrischen Bauteilen. Unter der Bedingung, dass nur gleichartige Bauteile auf eine Chargierplatte abgelegt werden, kann diese Parkettierung analytisch beschrieben werden. Anschließend werden die tetromino-förmigen Bauteile betrachtet. Da sich hier kein analytischer Zusammenhang erstellen lässt, oder zumindest dieser nicht bekannt ist, wird hierfür eine KI nach der RL-Methode angelernt. Hierfür wird eine entsprechendes *environment* benötigt und ein Programmcode für das Training.

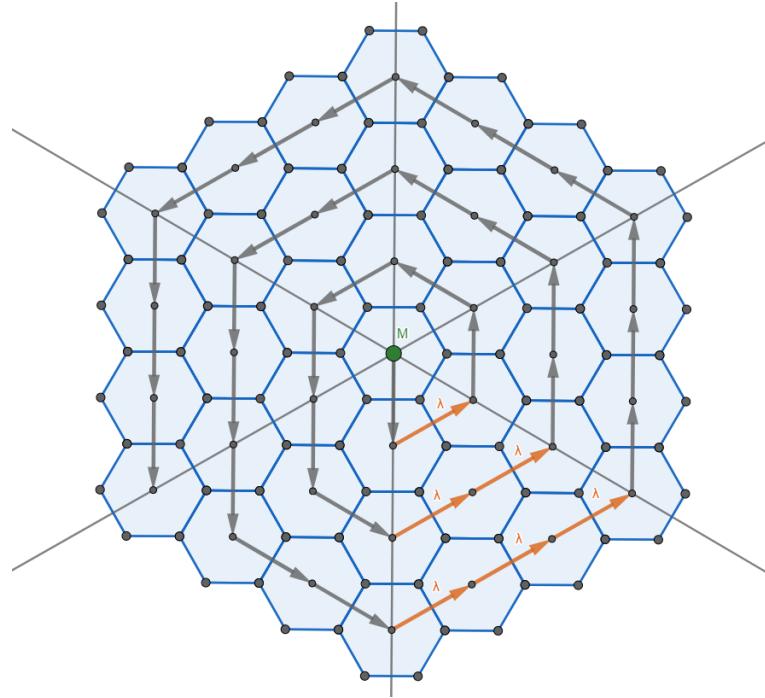
### 3.4.1 Parkettierung von zylindrischen Bauteilen

Für die Parkettierung von zylindrischen Bauteilen scheint die Hexagonform am geeignetsten (Kap. 2.5.2), daher wird nun eine mathematische Beschreibung für die

### 3 Konzeptentwicklung

---

Umsetzung gesucht. Ziel ist eine Funktion, die aufgrund des Tischdurchmessers und einer charakteristischen Länge des Hexagons, Positionen und deren Reihenfolge für die Belegung berechnet. Hierfür wird zunächst das Parkettierungsmuster mit GEOGEBRA (Abb. 3.7) gezeichnet und auf mögliche, mathematisch beschreibbare Muster untersucht.



**Abbildung 3.7:** Hexagon-Parkettierung mit Geogebra

Es fällt auf, dass eine Verbindung der Mittelpunkte wieder in einer Hexagonform mündet und die Anzahl dieser Punkte nach außen hin immer um sechs zunimmt (Abb. 3.9 und 3.10). Führt man nun Richtungsvektoren ein, um diese Eigenschaft zu verdeutlichen, erhält man zugleich die passende Reihenfolge für eine Belegung der benötigten Positionen. Man stellt außerdem fest, dass die Beschreibung der Mittelpunkte durch eine Summe von Richtungsvektoren ausgedrückt werden kann, deren Auswahl durch die sechs unterschiedlichen Richtungen begrenzt ist. Spielt man den Aufbau der ‚Hexagonblume‘ in Gedanken durch, kann zusätzlich festgestellt werden, dass die Anzahl und Richtung der Vektoren ein sich wiederholendes Muster ergeben (orange gekennzeichnet). Diese Informationen genügen, um einen Programmcode für den spiralförmigen Aufbau der Hexagone zu erstellen.

Programm 3.1, 3.2, 3.3 und 3.4 zeigen eine Implementierung mittels Python, welche in den folgenden Absätzen erläutert wird. Begonnen wird mit der Einbindung der benötigten Bibliotheken.

### 3 Konzeptentwicklung

---

#### Programm 3.1: Bibliotheken

```
1 import math
2 import numpy as np
3 import matplotlib.pyplot as plt
```

Für den Prototyp werden drei Bibliotheken verwendet: MATH um die Winkel-funktion zugänglich zu machen, die für die Beschreibung der Richtungsvektoren benötigt werden. NUMPY um die Rechenregeln für Vektoren verwenden zu können. MATPLOTLIB um den Programmcode zu debuggen und die Ergebnisse grafisch zu veranschaulichen. Nun kann die eigentliche Definition der Funktion stattfinden:

#### Programm 3.2: Hexagonfunktion

```
1 def calcHexagons(hexInnerDia, tableDia):
2
3     pointArray = []
4     p0 = np.array([0,0])
5     pointArray.append(p0)
6     k=1;
7     i=0;
8     history_k = 0;
9     notFinished = True
10
11    while notFinished:
12        while len(pointArray) <= k*6 + history_k:
13            for j in range(k):
14                direction = dirVector(i-history_k,k,hexInnerDia)
15                pointArray.append(pointArray[i] + direction)
16                i += 1
17            k += 1
18            history_k = len(pointArray)-1
19
20            direction = dirVector(i-history_k,k,hexInnerDia)
21            radius = (tableDia-hexInnerDia)/2
22            if np.linalg.norm(pointArray[i] + direction) > radius:
23                notFinished = False
24
25    return pointArray
```

Der Funktion „calcHexagons()“ werden zwei Übergabeparameter mitgegeben, der Innendurchmesser des Hexagons als charakteristische Abmessung und der Tisch-durchmesser, also die Größe der zu parkettierenden Kreisfläche. Anfangs wird der benötigte Rückgabewert der Funktion angelegt, nämlich die Liste von Punkten beschrieben im kartesischen Koordinatensystem. Anschließend wird der Ursprung des

### 3 Konzeptentwicklung

---

Koordinatensystems als erster Punkt ‚p0‘ angelegt und der Liste von Punkten übergeben. Im nächsten Schritt werden Laufvariablen und eine Abbruchbedingung angelegt, um den Programmaufbau übersichtlicher zu gestalten. Nach dem Anlegen dieser Variablen wird mit der äußersten Schleife begonnen. Diese Schleife wird so lange ausgeführt, bis die Abbruchbedingung erfüllt ist. Die Abbruchbedingung wird in der darunter liegenden Schleife überprüft. Sie evaluiert die Lage des nächsten zu generierenden Punktes und bricht die Programmschleife ab, falls sich dieser außerhalb der Kreisfläche befindet. Die zweite Schleife des Programms generiert die äußeren Mittelpunkte bei einer gedanklichen Umrundung mittels den Richtungsvektoren. Innerhalb dieser Schleife befindet sich schließlich der *for-loop*, der die Zunahme der Richtungsvektoren pro Zyklus abbildet. Um die Funktion übersichtlich zu halten, wird das Problem der Auswahl des Richtungsvektors in einer anderen Funktion beschrieben. Hierfür werden 3 Parameter übergeben: Als erstes ein Index gültig für den momentanen Umlauf, der immer einen Wert zwischen Null und der Anzahl an äußeren Hexagone (der aktuellen Instanz) annimmt (Abb. 3.9, 3.10). Der Parameter k, welcher die aktuelle Instanz bestimmt und ‚hexInnerDia‘ welcher die charakteristische Abmessung des Hexagons definiert.

**Programm 3.3:** Hilfsfunktion

```
1 def dirVector(i,k,hexInnerDia):
2
3     if i == 0:
4         return np.array([hexInnerDia*math.cos(3*math.pi/2),
5                         hexInnerDia*math.sin(3*math.pi/2)])
6     if i != 0 and i // k == 0:
7         return np.array([hexInnerDia*math.cos(11*math.pi/6),
8                         hexInnerDia*math.sin(11*math.pi/6)])
9     if i // k == 1:
10        return np.array([hexInnerDia*math.cos(math.pi/6),
11                         hexInnerDia*math.sin(math.pi/6)])
12    if i // k == 2:
13        return np.array([hexInnerDia*math.cos(math.pi/2),
14                         hexInnerDia*math.sin(math.pi/2)])
15    if i // k == 3:
16        return np.array([hexInnerDia*math.cos(5*math.pi/6),
17                         hexInnerDia*math.sin(5*math.pi/6)])
18    if i // k == 4:
19        return np.array([hexInnerDia*math.cos(7*math.pi/6),
20                         hexInnerDia*math.sin(7*math.pi/6)])
21    if i // k == 5:
22        return np.array([hexInnerDia*math.cos(3*math.pi/2),
23                         hexInnerDia*math.sin(3*math.pi/2)])
```

### 3 Konzeptentwicklung

---

Im wesentlichen ist die Funktion ‚dirVector()‘ nur eine Fallunterscheidung mit nicht-trivialen Bedingungen. Die Fallunterscheidungen lassen sich wie folgt in Worte fassen: Falls der aktuelle Punkt der erste der neuen Instanz ist ( $i==0$ ), soll der Richtungsvektor in negative y-Richtung zurückgegeben werden. Wenn der aktuelle Punkt nicht der Erste ist, aber die Ganzzahldivision mit der Instanzzahl ‚k‘ gleich Null ist, dann soll der Richtungsvektor in  $\frac{11\pi}{6}$ -Richtung zurückgegeben werden. Dieses Vorgehen ist für die restlichen Bedingungen analog.

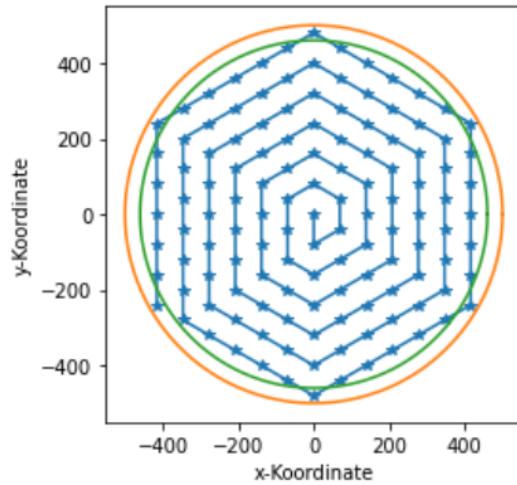
**Programm 3.4:** Testroutine

```
1 #testroutine
2 hexagonDia = 80
3 tableDia = 1000
4
5 #plot middlepoints of hexagons
6 a = np.array(calcHexagons(hexagonDia,tableDia))
7 #b = hexagonfilter(a, hexagonDia, tableDia)
8 x, y = a.T
9 plt.plot(x, y, '*-')
10
11 #plot table
12 angle = np.linspace( 0 , 2 * np.pi , 150 )
13 radius = tableDia/2
14 a = radius * np.cos( angle )
15 b = radius * np.sin( angle )
16 plt.plot(a,b)
17
18 #plot boundary
19 radius = (tableDia-hexagonDia)/2
20 a = radius * np.cos( angle )
21 b = radius * np.sin( angle )
22 plt.plot(a,b)
23
24 plt.xlabel('x-Koordinate')
25 plt.ylabel('y-Koordinate')
26 plt.gca().set_aspect('equal', adjustable='box')
27 plt.show()
```

In der Testroutine werden Werte für den Innendurchmesser der Hexagone und des Tischdurchmessers angelegt. Diese Werte werden anschließend der programmierten Funktion ‚calcHexagons()‘ übergeben und das Ergebnis auf der Variablen a gespeichert. Um die übliche Formatierung für einen Plot-Befehl zu erhalten, muss die Matrix noch transponiert werden. Anschließend werden die Punkte der Liste geplottet. Für die bessere Visualisierung werden die Punkte mit einem ‚\*‘ markiert

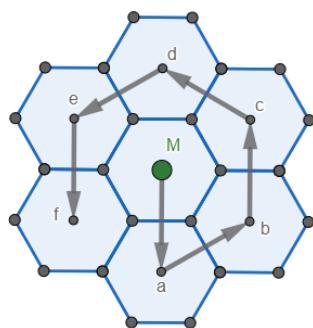
### 3 Konzeptentwicklung

und in der Reihenfolge der Generierung miteinander verbunden. Des Weiteren wird noch der Kreisumfang des Tisches und der Kreisumfang der Abbruchbedingung in den Plot mit aufgenommen. Mit dem Befehl „plt.show()“ wird die Grafik ausgegeben (Abb. 3.8).

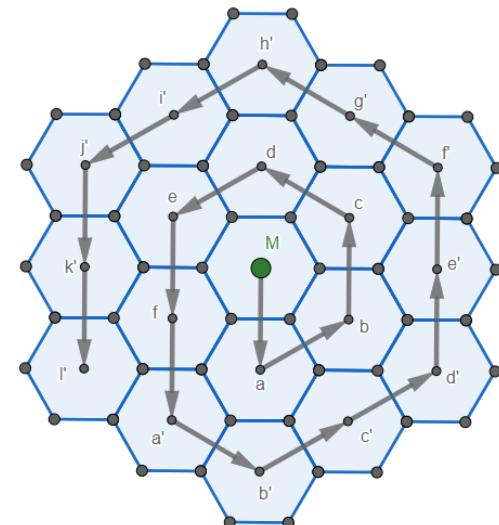


**Abbildung 3.8:** Ergebnis des Programmcodes für die Parkettierung

Das Ergebnis erfüllt die geforderten Bedingungen. Wie in der Hauptroutine ange deutet (auskommentiertes b) müssen noch Filter-Funktionen entwickelt werden, um die „Hexagonblume“ von zu weit entfernten Punkten und ausgewählten Positionen zu befreien. Die Filterung von ausgewählten Punkten wird benötigt, um Platz für Standfüße der darauf gestapelten Chargiergegestelle zu schaffen. Die Umsetzung einer solchen Filterfunktion wird im Kap. 3.4.5 vorgestellt.



**Abbildung 3.9:** Hexagone 1. Instanz



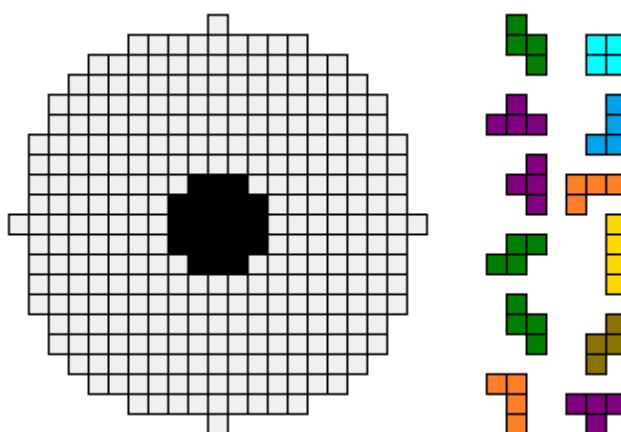
**Abbildung 3.10:** Hexagone 2. Instanz

### 3.4.2 Parkettierung von Tetrominos

Tetrominos [34] sind geometrische Formen, die aus vier Quadranten bestehen, wie sie auch aus dem Spiel Tetris bekannt sind. Für die Parkettierung von Tetrominos wird die Rechteckform der platonischen Parkettierung (Kap. 2.5.1) herangezogen. Auch hier wird wieder eine Vorgehensweise benötigt, um Positionen auf einer Ebene automatisiert generieren zu können. Für eine mögliche Umsetzung der Problematik wird der Einsatz einer KI, nach der DQL- und TDL-Methode aus Kap. 2.6.4.7 und 2.6.4.6, untersucht. Für den genannten Zweck muss eine Umgebung angefertigt werden, in der die KI mögliche Vorgehensweisen entwickeln kann. Hierfür wird zunächst eine minimalistischer Prototyp des *environments* mittels Python entwickelt, anhand dessen die nun folgenden Überlegungen angestellt wurden.

#### 3.4.2.1 Environment and Rewards

Begonnen wird mit der Diskretisierung der Chargierplatte anhand quaderförmiger Teilbereiche, wie sie aus der Platonischen-Parkettierung (Kap. 2.5.1) hervorgehen. Hierfür wird eine charakteristische Länge benötigt, welche für die Approximation der Bauteile geeignet erscheint. Abbildung 3.11 zeigt diese Diskretisierung von Chargierplatte und Bauteilen. Die schwarze Fläche im Zentrum stellt eine *deadzone* dar, auf welche die Bauteile nicht abgelegt werden dürfen. Solche *deadzones*, also Bereiche die in der Praxis frei bleiben müssen, können, wie bei der Parkettierung von zylindrischen Bauteilen, mittels geeigneten Filtern (Kap. 3.4.5) erzeugt werden. Ein jeder der Quader stellt dabei einen Eintrag in einer Matrix dar, welche mit der Numpy-Bibliothek angelegt werden kann. Für die Bewertung von abgelegten Tetrominos werden Kriterien benötigt, welche im Weiteren eine Punktevergabe ermöglichen. Diese Kriterien werden in den folgenden Absätzen erarbeitet.



**Abbildung 3.11:** Diskretisierung von Bauteilen und Chargierplatte

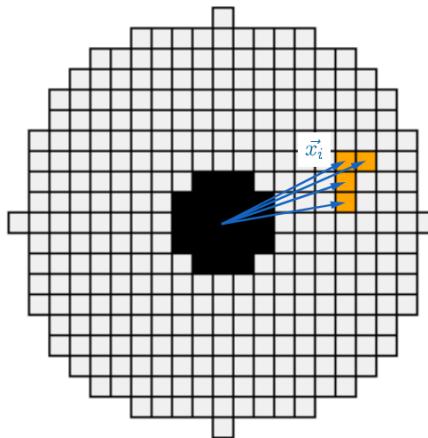
### 3 Konzeptentwicklung

---

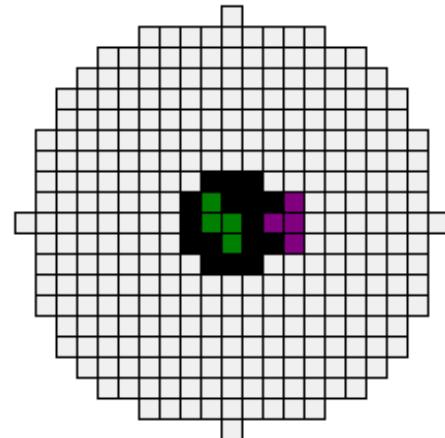
Bei der Chargierung ist es günstig, sich von innen nach außen zu arbeiten, somit erscheint der Abstand von Chargierplatte zum Flächenschwerpunkt der Bauteile als geeignetes Bewertungsmaß. Hierfür bestimmt man die Abstände zu jedem der Quader vom Mittelpunkt der Chargierplatte und bildet den Mittelwert (Glg. 3.1, Abb. 3.12).

$$\vec{x} = \frac{1}{4} \sum \vec{x}_i \quad (3.1)$$

Durch die Einführung von *deadzones*, kann auch die Schwebung von Bauteilen oder deren Ablage innerhalb dieser Zonen bewertet werden. Jedoch ist die Sinnhaftigkeit eines solchen Bewertungsmaßes von den Aktionsmöglichkeiten des *agents* abhängig, da diese Möglichkeiten auch eingeschränkt werden können, wie es in Kap. 3.4.2.2 angedeutet wird. Abbildung 3.13 zeigt eine Verletzung einer solchen *deadzone*-Regel.

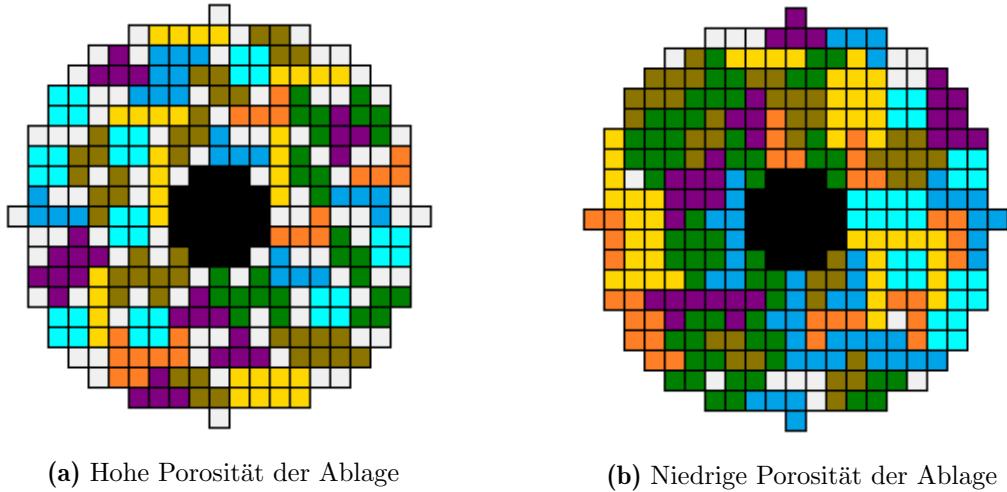


**Abbildung 3.12:** Flächenschwerpunkt



**Abbildung 3.13:** *Deadzone*-Regel

Ein weiteres Bewertungsmaß kann durch die Erhöhung eines Porosität-Wertes gewonnen werden. Um den Begriff Pososität in Bezug zu den Bewertungskriterien zu setzen wurden zwei Darstellungen (Abb. 3.14) angefertigt. Abbildung a zeigt die Ablage mit hoher Porosität wohingegen b eine Ablage mit niedriger Porosität zeigt. Außerdem zeigt b eine weitere Eigenschaft der Ablage: die unbelegten Felder rechts oben und mittig unten haben die Form eines Tetrominos und hätten somit belegt werden können, wenn das passende Bauteil erschienen wäre. Somit gelangt man zur Annahme, dass die Art der Verteilung, mit welcher die Bauteile auftreten, eine wesentliche Rolle spielt. Bei diesem Prototyp wurde eine Gleichverteilung verwendet, jedoch könnte man auch andere Verteilungsarten implementieren.



**Abbildung 3.14:** Porosität der Ablage von Tetrominos

Bis jetzt wurden also der Abstand vom Mittelpunkt  $\vec{x}$  und die Anzahl an verschwendeten Plätzen  $n_{wasted\_places}$  als potenzielle Bewertungskriterien gefunden. Eine weitere Möglichkeit ist das Zählen der anliegenden Quadrate, daher, es kann nach der Ablage gezählt werden, welche Seitenkanten unmittelbar einen Nachbarn besitzen. Somit ergibt sich ein Parameter  $n_{neighbours}$ . Was ab hier berücksichtigt werden sollte, ist, dass alle Tetrominos, bis auf *Smashboy* (Tab. 3.1), 10 Seitenkanten besitzen. Dies hat daher eine Benachteiligung für diese Form zur Folge.

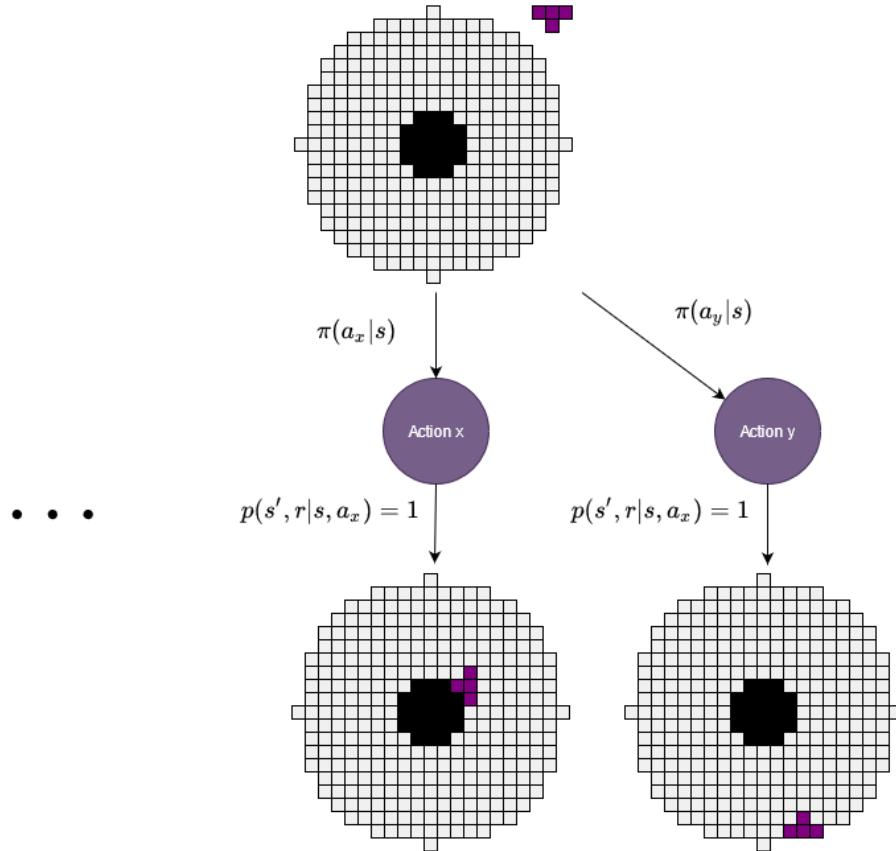
$$R_i = \frac{radius}{|\vec{x}_i|} * C_1 - n_{wasted\_places} * C_2 + n_{neighbours} * C_3 + C_4 \quad (3.2)$$

Gleichung 3.2 zeigt eine mögliche Belohnungsstrategie für das Setzten eines Bauteils. Die Konstanten  $C_1$  bis  $C_4$  müssen durch Versuche bestimmt werden. Somit kann man nun jedoch nur das Setzen an sich bewerten, es steht also noch die Frage nach den möglichen Aktionen des *agents* offen.

### 3.4.2.2 Agent

In diesem Abschnitt werden zwei Interaktionsschema des *agent* mit dem *environment* vorgestellt. In der ersten Variante lässt man den *agent* aus einem Pool aller möglichen Ablegepositionen auswählen, in der zweiten Variante aus einem Pool von Einzelaktionen. Eine Aktion nach dem ersten Schema wird durch eine Dreier-Tupel dargestellt. Dieses Tupel enthält die gewünschten x- und y-Koordinaten und die Anzahl der 90°-Rotationen (Glg. 3.3, Abb. 3.15). Da dieses Aktionsschema die direkte Ablage an einer Zielposition vorgibt, wird es im weiteren als ‚placement‘ bezeichnet.

$$a = (x - \text{Koordinate}, y - \text{Koordinate}, n - \text{Rotationen}) \quad (3.3)$$



**Abbildung 3.15:** Aktionsschema: Ablegepositionen

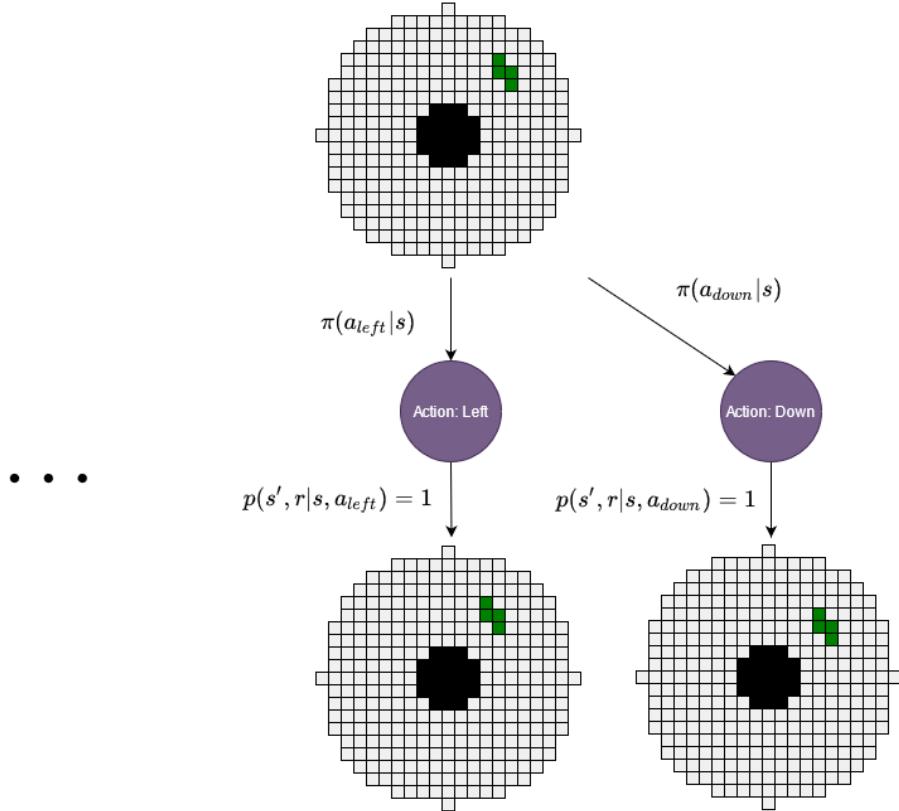
Bei dem zweiten Aktionsschema handelt es sich um Steuerung durch Einzelaktionen wie rauf, runter, links, rechts, Drehung nach links und Drehung nach rechts. Abbildung 3.16 zeigt eine Interaktion nach diesem Schema, welche von nun an als ‚controller‘ bezeichnet wird. Hierbei müssen jedoch auch die Einzelaktionen über Punkte in den Gesamt-*reward* einfließen.

Wie in den Abbildungen angeführt, spielt bei den vorgestellten Schemen die *transition dynamic* meist keine Rolle, da eine gewisse Aktion immer nur in einen daraus resultierenden Zustand mündet. Der größte Unterschied der beiden Schemen ist die Anzahl an Aktionen und die daraus resultierende Anzahl an Zuständen, die in einer *episode* auftreten können. Die Anzahl aller möglichen Zustände ist jedoch gleich, nämlich  $2^n$ . Dabei steht die Basis von 2 für eine binäre Codierung (ob ein Feld belegt ist oder nicht) und der Exponent  $n$  steht für die Anzahl an belegbaren Feldern (Vierer-Blöcke). Bei dieser Rechnung vernachlässigt man zwar die Abhängigkeit zwischen den vier Quadranten, dennoch bildet die Rechnung den Sachverhalt gut ab.

### 3 Konzeptentwicklung

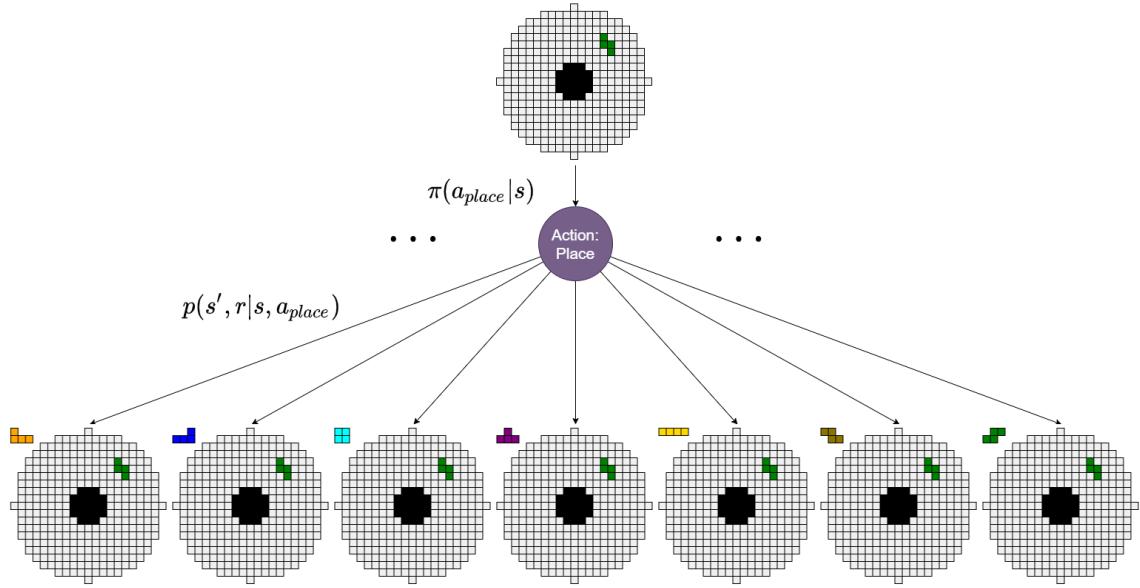
---

Auf dem hier verwendeten Prototyp befinden sich 296 belegbare Felder, daraus würden  $2^{\frac{296}{4}} \approx 1.9e22$  mögliche Zustände ergeben. Diese kleine Rechnung zeigt, welch umfangreiche Problemstellung gelöst werden muss.



**Abbildung 3.16:** Aktionsschema: Einzelaktionen

Im obigen Absatz wurde erwähnt, dass die *transision dynamic* meist keine Rolle spielt. Dies ist jedoch nur mit einer Ausnahme korrekt: dem Ablegen von Bauteilen. Werden nämlich Bauteile abgelegt, wird erst nach eben dieser Ablage ein neues Bauteil bestimmt. Abbildung 3.17 zeigt diese Verzweigung der Transition. Nun müssen auch die einzelnen Transitions-Wahrscheinlichkeiten beschrieben werden. Dies kann mithilfe eines Vektors, welcher diese Wahrscheinlichkeiten beinhaltet, ermöglicht werden (Glg. 3.4). Dadurch behält man sich die Möglichkeit, die Transitions-Wahrscheinlichkeiten problemabhängig vorzugeben, da eine Gleichverteilung die Realität meist nicht ausreichend genau abbildet. Weiterführende Gedanken dazu befinden sich im Kap. 3.4.3. Zu guter Letzt sei noch angemerkt, dass auch die Erscheinungsposition, welche in Abb. 3.17 immer links oben ist, aus Zufallszahlen generiert werden kann. Diese Zufälligkeit könnte das Training weiter beeinflussen.



**Abbildung 3.17:** Einzelaktion: Bauteil hinlegen

$$\mathbf{p}^T = \begin{bmatrix} p_0, p_1, p_2, p_3, p_4, p_5, p_6 \end{bmatrix} \quad (3.4)$$

### 3.4.2.3 Observation und Features

Für eine Lernroutine müssen Zustandsdarstellungen, die dem neuronalen Netzwerk als Input dienen, entworfen werden (Kap. 2.6.1.7). Als erste Darstellungsmöglichkeit bietet sich eine binäre Matrixnotation an. Hierfür wird die Umgebung in zwei Ebenen aufgeteilt: Eine Ebene stellt den Ablagebereich dar, die Andere die Bewegung des Bauteils. Beide Ebenen sind binäre Matrizen gleicher Dimension. Abbildung 3.18 zeigt eine solche Zustandsdarstellung für beide Matrizen. Diese Zustandsdarstellung wird im Folgenden als ‚full\_bool‘ bezeichnet.

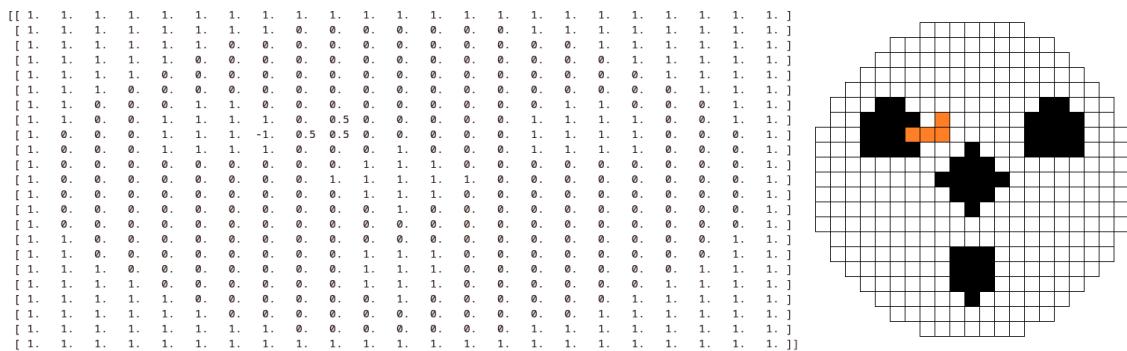
(a) Matrix des Ablagebereichs

(b) Matrix des Bauteils

**Abbildung 3.18:** Zustandsdarstellung mit binären Werten

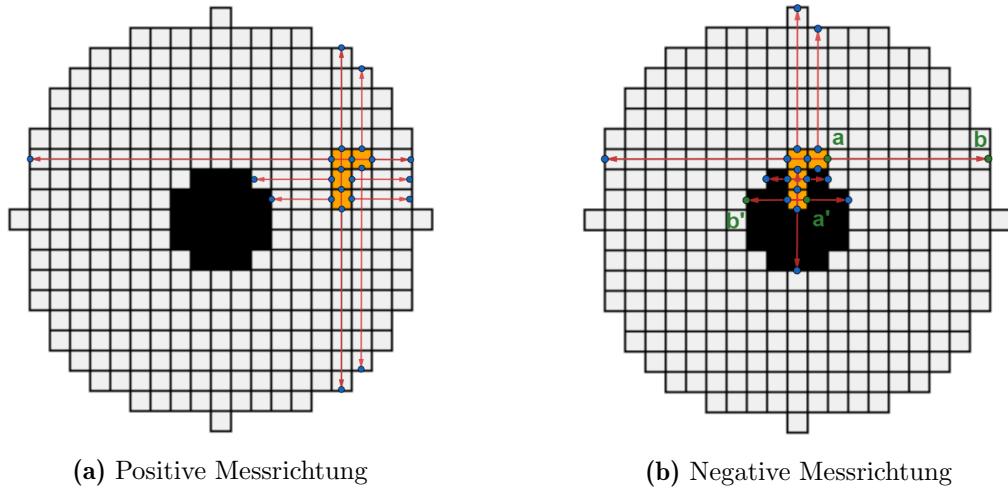
### 3 Konzeptentwicklung

Eine abgewandelte Form dieser Darstellung ist die Generierung einer einzelnen Ebene aus beiden Ebenen. Dies hat den Vorteil, dass der Input-Vektor in seiner Dimension um die Hälfte schrumpft. Nun werden jedoch Gleitkommazahlen für die Darstellung benötigt, da eine boolsche Beschreibung nicht mehr ausreicht. Behält man die Zahlenwerte für Belegt bzw. Unbelegt mit 0 und 1 bei, müssen Zahlenwerte definiert werden, falls das Bauteil über einem belegten Platz bzw. einem unbelegten Platz schwebt. Da die Schwebung über einen belegten Platz als ungünstig erscheint, wird dieser mit dem Wert -1 belegt. Die Schwebung über einen unbelegten Platz erscheint günstig und wird mit dem Wert 0.5 belegt. Somit wären die Zustände auf den Wertebereich  $[-1,1]$  begrenzt. Abbildung 3.19 zeigt einen solchen Zustand, der von nun an als ‚full\_float‘ bezeichnet wird.



**Abbildung 3.19:** Zustandsdarstellung mit Gleitkommazahlen

Eine weitere Möglichkeit einer Zustandsdarstellung ist die Sicht des Bauteils auf seine Umgebung, wie es z. B. bei selbst-fahrenden Autos praktiziert wird. Hierfür wird für alle Quader des Tetrominos der Abstand aller 4 Seitenflächen zu einem Übergang von *deadzone* auf Freifläche, oder vice versa, berechnet. Bei Nicht-Überlappung von Bauteil und *deadzone* wird positiv gezählt wie in Abb. 3.20 a dargestellt. Falls jedoch eine Überlappung vorliegt, wird in negative Richtung gezählt. Abbildung 3.20 b soll diesen Unterschied verdeutlichen. Betrachtet man den Vektor von Punkt *a* auf Punkt *b* wird die Distanz  $8 * \text{Blockgröße}$  gezählt. Betrachtet man dagegen den Vektor von Punkt *a'* auf Punkt *b'* wird die Distanz  $-3 * \text{Blockgröße}$  gezählt. Außerdem kann dieser Darstellung noch die Flächenschwerpunkte der Einzelblöcke hinzugefügt werden. Alle Werte müssen anschließend mittels Division durch einen Maximalwert noch auf den Wertebereich  $[-1,1]$  normiert werden. Da diese Zustandsbeschreibung am wenigsten Werte enthält, wird sie von nun an als ‚reduced‘ bezeichnet.



**Abbildung 3.20:** Messrichtungen für die Zustandsdarstellung „reduced“

#### 3.4.2.4 Neuronale Netzwerke

In diesem Kapitel wird auf die Entscheidungsfindung eingegangen. Hierfür werden grundlegend zwei verschiedene Vorgehensweisen vorgestellt. Die erste Vorgehensweise beschäftigt sich mit der Auswahl der Aktion anhand der geschätzten *state-values*, während sich die zweite Vorgehensweise mit der Aktionswahl nach dem DQL-Schema befasst.

##### Bestimmung state-value

Da die Anzahl an möglichen Zuständen nach einer Transition eher gering sind, kann man alle aus den Aktionen resultierende Zustände vorab berechnen. Für diese Folgezustände kann der resultierende *state-value* über ein neuronales Netz bestimmt werden. Die Auswahl der Aktion geschieht anschließend durch die Wahl des Folgezustandes mit dem höchsten, geschätzten *state-value*. Abbildung 3.21 zeigt auf der linken Seite ein Set an möglichen Folgezuständen, die durch die Wahl der dazugehörigen Aktion zustandekommen würden. Geschickt durch eine neuronales Netz, lassen sich die dazugehörigen *state-values* bestimmen. Es sei angemerkt, dass diese Methode für die Approximation der *reward*-Funktion (3.2) mit  $\gamma = 0$  herangezogen werden kann.

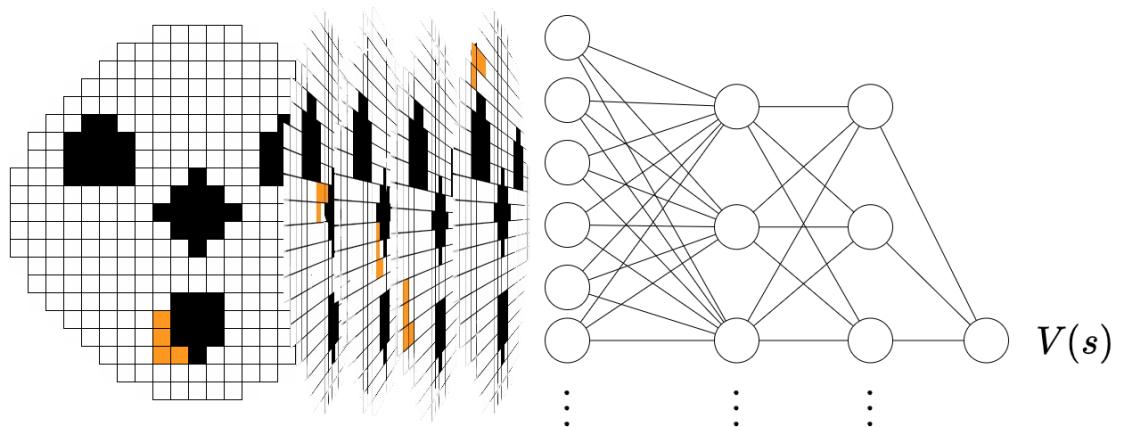


Abbildung 3.21: Schätzer für *state value*

### Bestimmung q-value

Da die Anzahl an möglichen Aktionen bei dem Aktionsschema ‚controller‘ überschaubar sind, ist die Vorgehensweise nach der DQL-Methode denkbar. Hierfür werden die *q-values* für alle möglichen Aktionen durch ein neuronales Netz geschätzt (3.22), mithilfe dessen die bestmögliche Aktion gewählt werden könnte. Hierbei kann man die verschiedenen vorgestellten Netztypen aus Kap. 2.6.4.7 verwenden. Vorteil dieser Methode ist unter anderem, dass nicht alle möglichen Folgezustände bestimmt werden müssten. Außerdem wäre somit auch eine Bewegung von Startposition zur Endposition mitbestimmt.

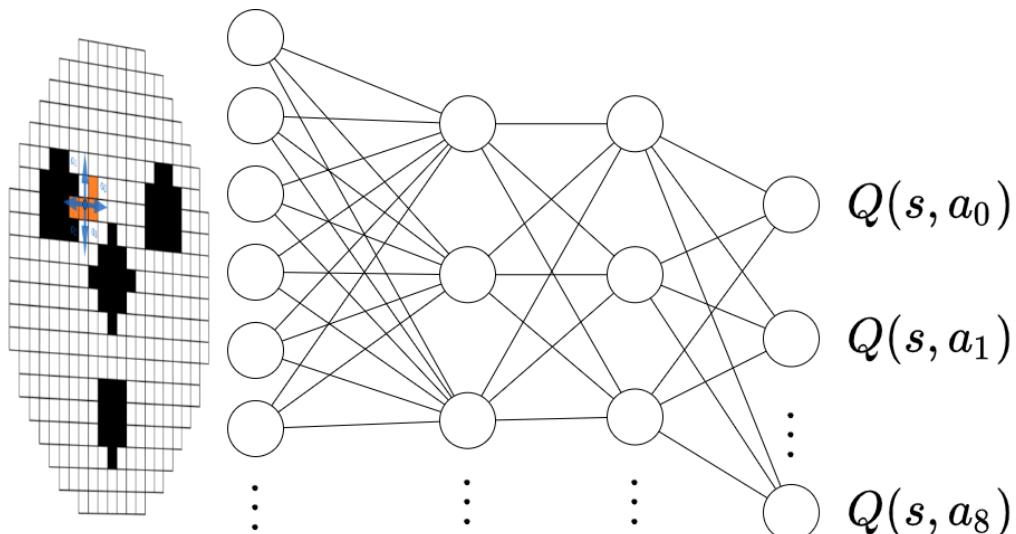


Abbildung 3.22: Schätzer für *q-value*

### 3.4.2.5 Benchmark

Es fehlt noch eine Methode für die Bewertung der KI, nach Abschluss der Trainingsroutine. Daher muss eine *benchmark* gefunden werden, welche zeigt, ob die KI gut oder schlecht performt. Hierfür werden die Konstanten der *reward*-Funktion (Glg. 3.2) in Versuchen solange verändert, bis die Wahl der Aktion mit den höchsten *reward*, der ja für alle möglichen Ablagepositionen bestimmt werden kann, ein zufriedenstellendes Ergebnis liefert. Bleibt die Rechenzeit gering, könnte diese Methode auch Einzug in die erstellte Software halten. Außerdem ist der Erfolg dieser Methode Voraussetzung, um alle Vorgehensweisen, wie sie in diesem Kapitel vorgestellt wurden, anzuwenden.

### 3.4.3 Bestimmung der Transitions-Wahrscheinlichkeiten

Im Kapitel 3.4.2.2 wurde auf die Problematik der *transition dynamics* hingewiesen, welche bei der Ablage von Bauteilen auftritt (Abb. 3.17). Für die Beschreibung dieses Ablagevorganges wurden die Transitions-Wahrscheinlichkeiten in einen Vektor zusammengefasst (Glg. 3.4), welcher im *environment* implementiert werden kann. Jedoch steht nun die Frage nach der Findung der konkreten Wahrscheinlichkeiten offen. Hierfür könnte man eventuell die zu produzierenden Bauteile eines Geschäftsjahrs, mithilfe von einer Klassifizierungs-KI (Kap. 2.6.3) in die jeweiligen Bauteilklassen unterteilen. Wurden die Bauteile erfolgreich klassifiziert, kann man mithilfe eines Histogramms die übergeordnete Verteilung für die Transitions-Wahrscheinlichkeiten verwenden. Abbildung 3.23 zeigt zwei mögliche Ergebnisse eines solchen Vorganges. Bei den dargestellten Wahrscheinlichkeitsverteilungen handelt es sich zum einen um eine Gleichverteilung (blau) und zum anderen um eine Normalverteilung (orange).

Die Transitions-Wahrscheinlichkeiten müssen sich natürlich zu 1 summieren.

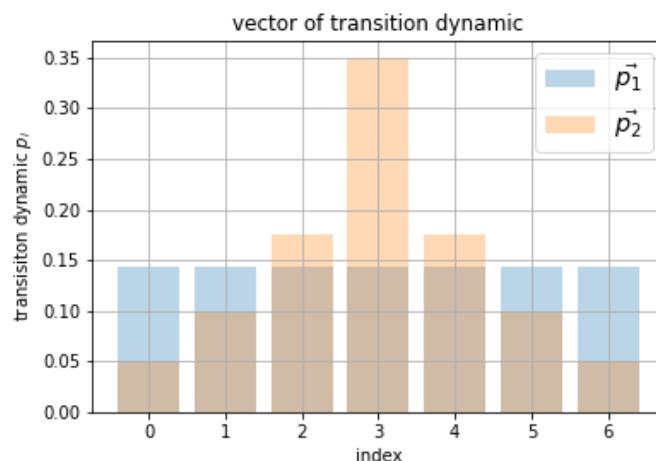


Abbildung 3.23: Plot Transitions-Wahrscheinlichkeitsvektor

### 3.4.4 Parkettierung durch Layoutvorgabe

Da nicht immer der Wunsch besteht, die Parkettierung vollautomatisch zu bestimmen, soll nun auch eine Schnittstelle für eine externe Parkettierungsstrategie entwickelt werden. Dies ist dem Sachverhalt geschuldet, dass Ablagepositionen oftmals vom fachkundigen Personal der Prozesstechnik vorgegeben werden. Als Input-Format wurde zu Gunsten der csv-Datei entschieden, welches die xy-Koordinaten der Ablageposition beinhaltet. Der *workflow* für die csv-Generierung ist in Abb. 3.24 ersichtlich. Zunächst wird eine Chargierplatte im Unternehmen verwendeten CAD-Programm (**SOLIDEDGE**) gezeichnet. Anschließend wird eine 2D-Abwicklung angefertigt, durch welche die Positionen als Bohrungstabelle ausgegeben werden können. Diese Bohrungstabelle kann abschließend über eine Excel-Tabelle als csv-Datei exportiert werden. Um zu zeigen, dass auch die zusätzliche siebte Achse zur Prozessoptimierung herangezogen werden kann, wird bei der Implementierung noch eine kleine Optimierungsroutine geschrieben, die den Transitionsweg von Bauteilaufnahme zu Bauteileablage im Zweidimensionalen minimiert (Kap 4.1.6).

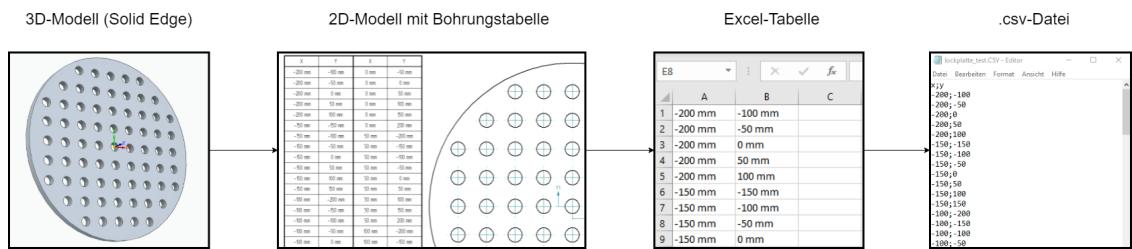


Abbildung 3.24: Workflow der .csv-Generierung

### 3.4.5 Parkettierungsfilter

Sowohl für das Hexagonmuster als auch für das *environment* der KI werden Filter benötigt, welche eine Freihaltung bestimmter Bereiche sicherstellen sollen. Diese Filter sollen parametrisiert aufgebaut sein, um eine möglichst hohe Flexibilität bieten zu können. Zum einen sollen Positionen innerhalb eines Innenkreises herausgefiltert werden, um die Gaszirkulation sicherzustellen, zum anderen werden Freiflächen für die Standbeine der Parkettiergestelle benötigt. Hierfür kann man, wie in Abb. 3.25 dargestellt, Relativvektoren  $\xi$  von den jeweiligen Mittelpunkten der *deadzone*-Flächen zur Position selbst bilden. Stellt man nun diese Vektoren in Polarkoordinaten dar, reicht eine einfache Vergleichsoperation zwischen der ersten Koordinate der Relativvektoren und den halben Durchmessern ‚middle\_dia‘ und ‚dead\_dia‘ der *deadzone*-Flächen.

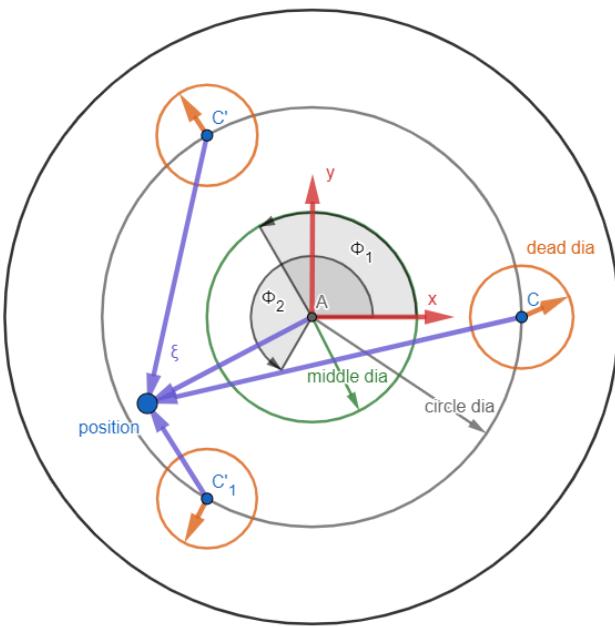


Abbildung 3.25: Filterung von Positionen

Programm 3.5: Parkettierungsfilter

```

1 def filter_deadzone(self, middle_dia, circle_dia, phis, dead_dia):
2     #filter everything inside middle_dia-deadzone
3     for position in self.pointArray:
4         polar = cart2pol(position)
5         if polar[0] < middle_dia/2:
6             self.pointArray.remove(position)
7
8     #filter everything inside standfoot-deadzones
9     for position in self.pointArray:
10        for phi in phis:
11            p0 = np.array([circle_dia/2, phi])
12            p1 = cart2pol(position)
13            xi = p1 - p0
14            if xi[0] <= dead_dia/2:
15                pointArray.remove(position)

```

## 3.5 Steuerung der Trainingszelle

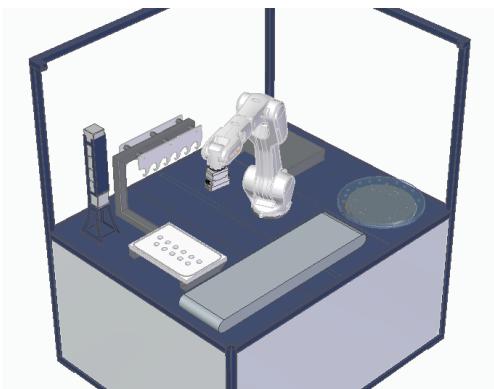
Bis jetzt wurden nur Konzepte für die Simulation und den Chargierprozess entwickelt, nun soll jedoch auch noch eine Methode entwickelt werden, um diese Konzepte auf realen Anlagen anwenden zu können. Hierfür wird eine Server-Client-Architektur

### 3 Konzeptentwicklung

genauer untersucht. Bei dem Server handelt es sich um einen OPC UA-Server, der auf einer Siemens-Steuerung angelegt wird. Dieser Server soll bestimmte Parameter dem Client zugänglich machen, um diesem die Möglichkeit zu gewähren, die Parameter zu überschreiben. Als *Proof-Of-Concept* wird eine einfache *Pick&Place*-Anwendung implementiert, welche mithilfe einer reduzierten Version des erstellten Programms gesteuert wird.

#### **3.5.1 Aufbau der Trainingszelle**

Das Kernstück der Trainingszelle stellt ein 6-Arm-Roboter von Kuka dar, es handelt sich dabei um das Modell AGILUS. Im Zuge der Inbetriebnahme wird eine SIMATIC S7 1500 von Siemens installiert, mithilfe derer Funktionen der MXAUTOMATION Bibliothek verwendet werden können [36]. Für eine Interaktion mit der SPS über OPC UA wird anfänglich eine RASPBERRY PI Modell 4B verwendet, um die nötige Kommunikationslogik zu erstellen und in eine entsprechende Klasse zu verpacken, welche anschließend in die Simulationssoftware eingebettet wird.



**Abbildung 3.26:** CAD-Modell



**Abbildung 3.27:** Realmodell

#### **3.5.2 OPC UA Datenmodell**

Die Kommunikation mit der Trainingszelle geschieht über eine Ansammlung von Variablen auf der SPS, die über OPC UA zugänglich gemacht werden. Siemens bietet hierfür ausreichend Dokumentation und etwaige Beispiele [37] [38]. Zunächst wird ein neuer Datenbaustein im Projektbaum der SPS angelegt. Dieser beinhaltet die benötigten Variablen, welche anschließend über die OPC UA-Einstellungen zur Verfügung gestellt werden. Die Definition der Schnittstelle wurde von einem externen Programmierer angefertigt und benannt. Dabei wurde der Begriff Mass-Flow Controller (MFC) als Bezeichnung für die zu erstellende Software eingeführt. Tabelle 3.2

### 3 Konzeptentwicklung

---

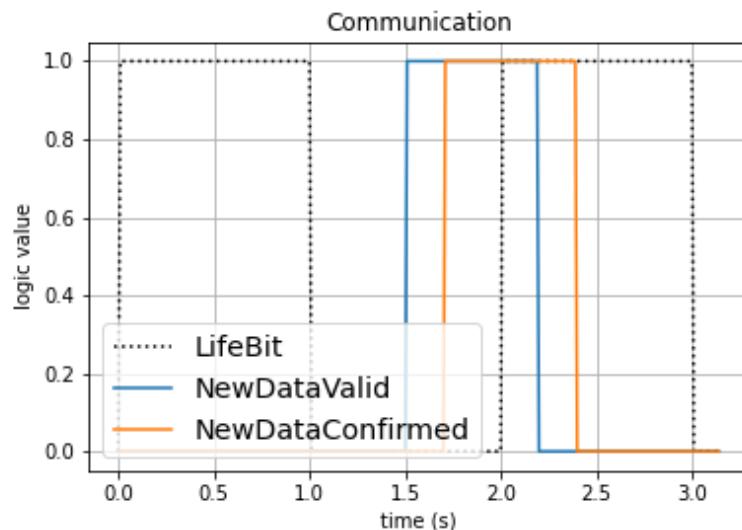
zeigt den dafür angelegten Datenbaustein, die Unterstrukturen sind in den Tabellen 12.3, 12.4, 12.5, 12.6 im Anhang beschrieben.

**Tabelle 3.2:** Datenbaustein: DB\_Interface\_MFC

Node	Datentyp	Beschreibung
MFC_Alife	Boolean	Logisch 1, wenn MFC online
MFC_to_Robot	Struct	Datenpaket: Übermittlung von MFC auf SPS
Robot_to_MFC	Struct	Datenpaket: Übermittlung von SPS auf MFC

### 3.5.3 Kommunikationsverlauf

Ein wesentlicher Punkt der Kommunikation ist der zu erstellende ‚Herzschlag‘, welcher der SPS mitteilt, ob der Treiber noch ordnungsgemäß läuft. Hierfür wird das Wechselintervall des vorgesehenen bool-Wertes gemessen. Falls der Timer die Fünf-Sekunden-Marke überschreitet, wird die Kommunikation als unterbrochen angenommen. Außerdem kommt in der Kommunikation ein *handshake* zum Einsatz. Werden von einem Teilnehmer Daten gesendet, wird als letztes das entsprechende Bit: ‚NewDataValid‘ auf logisch 1 gesetzt. Anschließend kopiert der Kommunikationspartner den Wert und setzt selbst das entsprechende ‚NewDataConfirmed‘-Bit (Tab. 12.3, 12.4). Zu guter Letzt werden beide Werte wieder auf logisch 0 zurückgesetzt, wobei die selbe Reihenfolge eingehalten wird. Abbildung 3.28 zeigt diese Vorgehensweise anhand einer groben Skizze.



**Abbildung 3.28:** Kommunikationsschema

### 3 Konzeptentwicklung

---

Die Manipulation der Daten wird mithilfe der Python-Bibliothek ASYNCUA realisiert [39]. Programm 3.6 zeigt das wohl einfachste Beispiel, die Manipulation eines boolschen Wertes. Hierbei wird auf das UA TCP für unverschlüsselte Kommunikation zurückgegriffen (Kap. 2.3.6). Für den Datenaustausch der Variablen unter der Struktur „Robot\_to\_MFC“ (Tab. 12.4) wird eine *subscription* (Kap. 2.3.3) angelegt.

**Programm 3.6:** Standard-NodeID Siemens S7 1500

```
1 from asyncua import Client, ua
2
3 url = 'opc.tcp://10.0.0.11:4840'
4
5 #Verbindungsaufbau mittels client
6 async with Client(url=url) as client:
7     #Lesen des gewünschten Nodes mittels Node-ID
8     node_id = "ns=3;s=\"DB_Interface_MFC\".\"MFC_Alife\""
9     variable = client.get_node(node_id)
10    #Auslesen des aktuellen Werts
11    current_value = await variable.read_value()
12    #Definition eines neuen, beliebigen Wertes
13    new_value = ua.Variant(True, ua.VariantType.Boolean)
14    new_data = ua.DataValue(new_value)
15    #Setzen des neuen Werts
16    await variable.set_value(new_data)
```

# 4 Implementierung

In diesem Kapitel wird die Implementierung der vorgestellten Konzepte (Kap. 3) erläutert. Nach einer kurzen Vorstellung des Simulationsprogramms, sowie deren Adaptierung für die Trainingszelle, wird auf das Training der KI eingegangen. Hierfür werden die erzielten Punkte per *episode* mittels der Bibliothek *TensorboardX* aufgezeichnet und visualisiert. Für die verwendete Soft- sowie Hardware wird an dieser Stelle auf den Anhang (Kap. 12.2) verwiesen.

## 4.1 Simulationsprogramm

Begonnen wird mit dem Aufbau und einer kleinen Funktionsbeschreibung des Simulationsprogramms aus Kap. 3.3.5. Anschließend werden verschiedene Konfigurationsmöglichkeiten für die Layout-Erzeugung gezeigt. Darauffolgend werden ein paar Implementierungen ausgewählt und diese näher beschrieben. Für die Dokumentation wurde mithilfe der Bibliotheken PYREVERSE und GRAPHVIZ Klassendiagramme generiert, die jene Attribute und Funktionen beinhalten, die bei der Implementierung angelegt wurden. Diese Klassendiagramme befinden sich im Quellcode.

### 4.1.1 Aufbau des Simulationsprogramms

Das Simulationsprogramm besteht aus vier Fenstern, was Objekten der vier Klassen aus dem Modul ‚simulation\_pages‘ entspricht (Kap. 3.3.1). Bei Starten erscheint als erstes die *home page* (Abb. 4.1), in der man die Anzahl der zu initialisierenden Objekte der Klassen aus dem Modul ‚simulation\_members‘ festlegt. Außerdem definiert man den Simulationsbereich, bzw. dessen Abmessung. Die einzige von der *home page* erreichbare Seite ist die *draw page* (Abb. 4.2). Auf dieser Seite kann durch drücken des *draw buttons* die Initialisierung der zuvor festgelegten Objekte beginnen. Durch Klicken in den grauen Bereich wird eine Routine ausgelöst und ein zusätzliches Fenster erscheint, welches die Änderung der Klassenparameter ermöglicht. Nach Beendigung der Einstellungen wird das entsprechende Objekt in der Instanz von ‚Simulation\_App‘ angelegt.

Will man etwaige Einstellungen ändern, kann man auf die *settings page* wechseln (Abb. 4.3). Diese Seite enthält alle Parameter der angelegten Objekte, die für die Prozessabwicklung von Interesse sind. Ist man mit dem Layout zufrieden, kann auf die *simulation page* (Abb. 4.4) gewechselt werden.

## 4 Implementierung

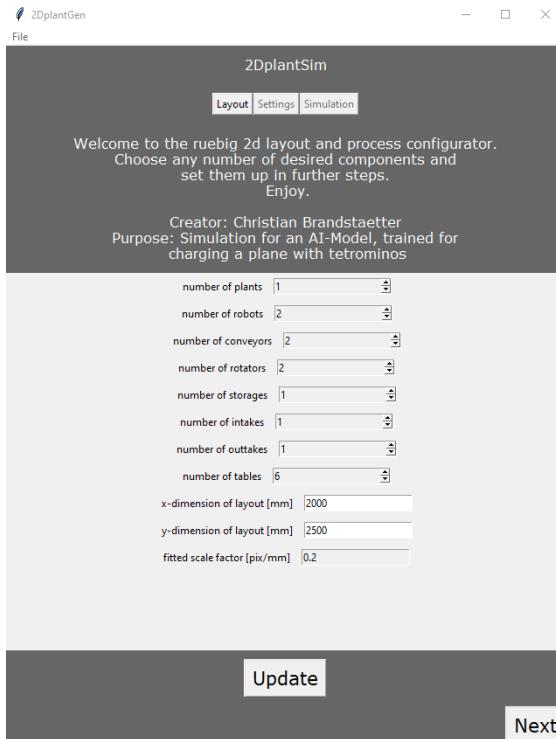


Abbildung 4.1: Home page

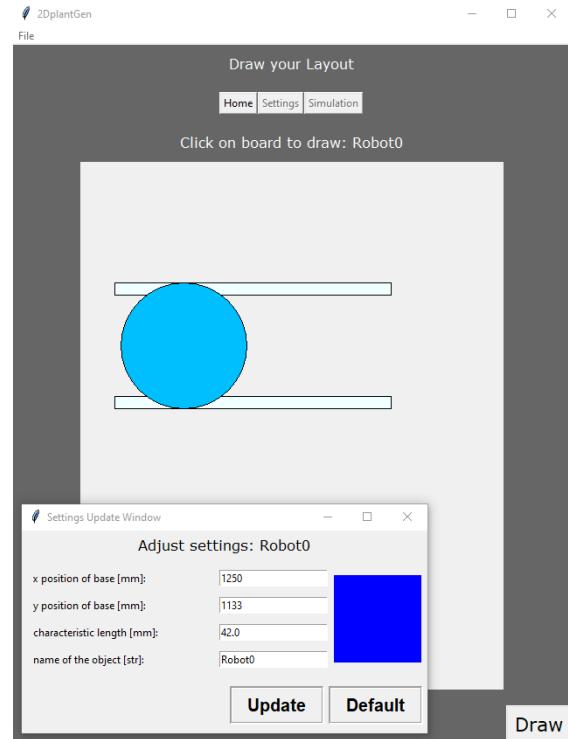


Abbildung 4.2: Draw page

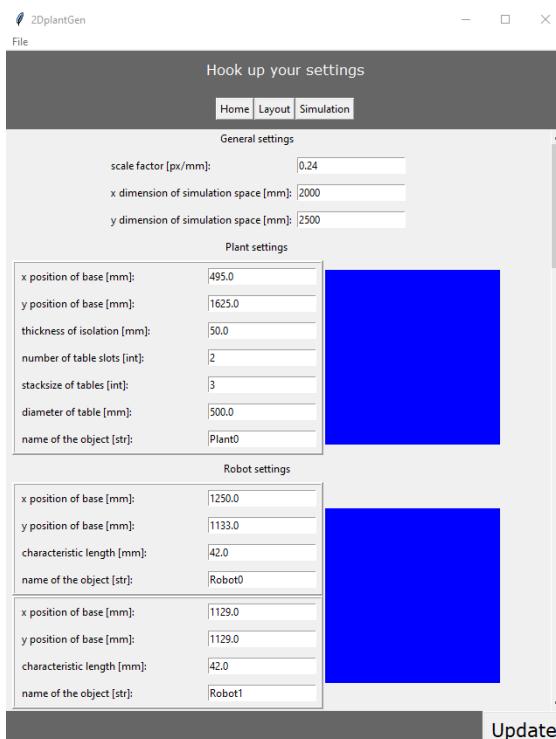


Abbildung 4.3: Settings page

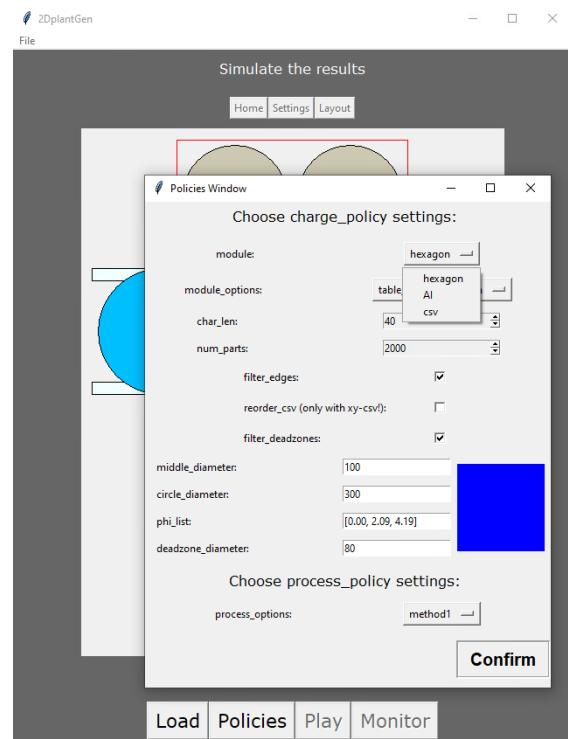


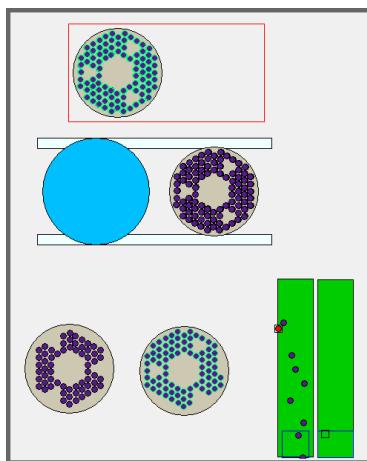
Abbildung 4.4: Simulation page

Auf dieser Seite wird zunächst das Layout geladen, anschließend können die *policies* eingestellt werden, welche das Verhalten des Systems steuern. Im Fenster „Policies“ kann man unter anderem die Positionen der Roboter und Tische sowie deren Größe und Form festlegen.

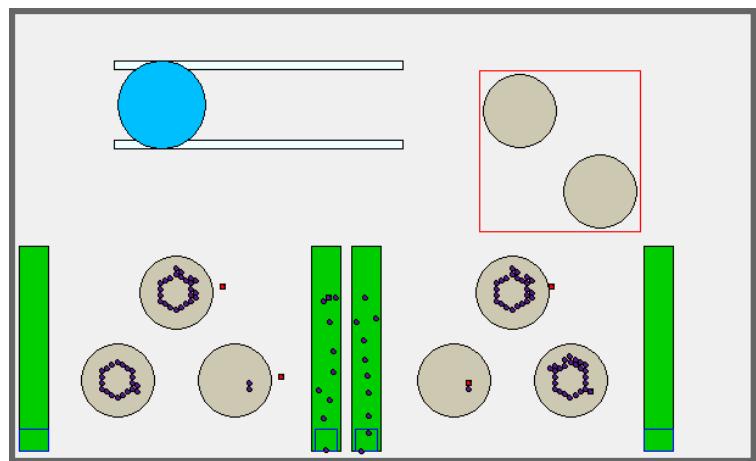
licies‘ werden unter anderem auch die Parameter für die Parkettierungsfilter eingegeben. Anschließend kann das System durch Drücken auf den *play button* simuliert werden. Des Weiteren kann während der Simulation ein Monitor aufgerufen werden, welcher die momentanen Zustände der einzelnen Anlagenkomponenten anzeigt.

### 4.1.2 Layoutkonfiguration

Nun soll noch auf die Konfigurationsmöglichkeiten bzw. auf die Skalierbarkeit des Systems eingegangen werden. So wie die Parkettierungsfilter aus Kap. 3.4.5 eine Änderung der zu parkettierenden Ebene bewirken, um die KI auch auf veränderbare environments trainieren zu können, so soll auch die Layouterstellung ein gewisses Maß an Varianz bieten. Dies wäre bei einer Weiterführung der Thematik von Interesse, da man auch für die Steuerung einer gesamten Produktionsstätte eine KI entwickeln könnte. An dieser Stelle sei jedoch angemerkt, dass hierfür das gesamte Programm neu überdacht und geschrieben werden müsste, da diese Implementierung bestenfalls als Vorschlag zu sehen ist. Abbildung 4.5 zeigt eine Konfiguration mit einer Tandem-Härteanlage mit zwei Plätzen und einer Stapelhöhe von Drei. Des Weiteren sind sechs Parkettiergestelle, zwei Förderbänder, ein Lagerplatz und zwei Rotatoren im Einsatz. Die Abbildung zeigt einen Entlade- und einen Beladevorgang. Das zweite Layout (Abb. 4.6) zeigt eine Konfiguration mit einer Tandem-Härteanlage mit drei Plätzen und einer Stapelhöhe von Sechs. Des Weiteren sind hier sechs Rotatoren, vier Förderbänder und 18 Parkettiergestelle im Einsatz. Hier befinden sich Einlässe auf den mittleren Förderbändern und Auslässe auf den Außen. Beide Konfigurationen werden mit der „process\_option: method1“ (Abb. 4.4) gesteuert.



**Abbildung 4.5:** Layout 1



**Abbildung 4.6:** Layout 2

### 4.1.3 Lineare und rotatorische Bewegung

Für die Bewegungen wird dem Objekt eine Geschwindigkeit übergeben, durch Integration dieser Geschwindigkeit über die Zeit (genauer den Aufrufzyklen) ergibt sich eine neue Position des Objekts (Glg. 4.1). Die neue Position wird anschließend erneut diskretisiert und in Pixel-Koordinaten umgewandelt. Das Update der Objekte in der Simulationsumgebung geschieht mit der Differenz des Zeitschrittes in Pixel. Es sei hier angemerkt, da keine harte Echtzeit bei den Aufrufen realisiert wurde, schwankt die Bewegungsgeschwindigkeit abhängig vom Aufrufintervall, vorgegeben durch das Betriebssystem. Abbildung 4.7 veranschaulicht diese Vorgehensweise für die Bewegung von A zu B.

$$v(t) = \frac{dx}{dt} \implies x(t) = x_0 + \int_{t_0}^{t_1} v(t) dt \quad (4.1)$$

**Programm 4.1:** Lineare Bewegung

```

1 def move(self, x_layout, y_layout, velo_max=8, epsilon=0.1):
2     #define lin. path:
3     dir_Vec = np.array([x_layout-self.x_layout,
4                         y_layout-self.y_layout])
5     dir_Vec_norm = np.linalg.norm(dir_Vec) #length
6     if dir_Vec_norm <= epsilon: #reached?
7         return True
8     elif dir_Vec_norm < velo_max: #close?
9         self.dx = dir_Vec[0]
10        self.dy = dir_Vec[1]
11    else:
12        dir_Vec_unit = dir_Vec / dir_Vec_norm
13        velo_Vec = velo_max*dir_Vec_unit
14        self.dx = velo_Vec[0]
15        self.dy = velo_Vec[1]
16
17        self.x_layout += self.dx
18        self.y_layout += self.dy
19        self.coord_trans()
20        self.dx_pixel = int(self.scale*self.x_tkinter) - self.x_pixel
21        self.dy_pixel = int(self.scale*self.y_tkinter) - self.y_pixel
22        self.canvas.move(self.id, self.dx_pixel, self.dy_pixel)
23        self.x_pixel += self.dx_pixel
24        self.y_pixel += self.dy_pixel
25
26    return False

```

## 4 Implementierung

Ähnlich der linearen Bewegung wird auch die rotatorische Bewegung über die passende, physikalische Differenzialgleichung (Glg. 4.2) implementiert. Für das Positionsupdate sind diskrete Werte für  $\varphi$  gegeben, die Bibliothek Tkinter lässt jedoch nur lineare Positionsupdates zu, daher wird der Kreis durch ein Polygon approximiert (Abb. 4.8).

$$\omega(t) = \frac{d\varphi}{dt} \implies \varphi(t) = \varphi_0 + \int_{t_0}^{t_1} \omega(t) dt \quad (4.2)$$

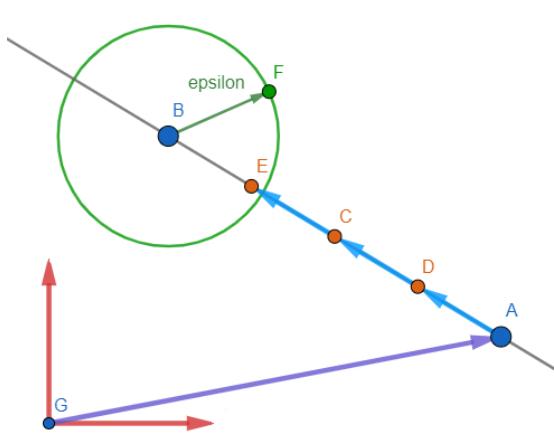


Abbildung 4.7: Lineare Bewegung

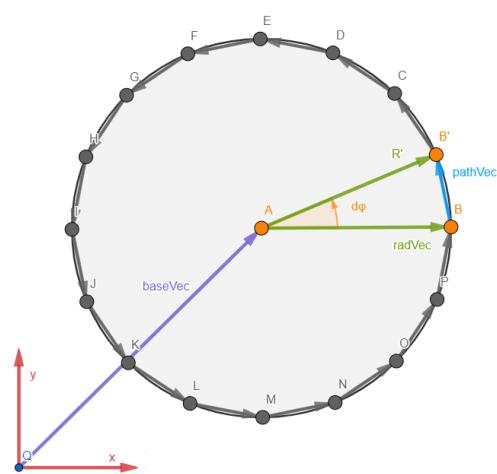


Abbildung 4.8: Rotation

### Programm 4.2: Rotatorische Bewegung

```

1 def rotate(self, phi, n=128, epsilon=0.01):
2     #Rotation as polygon(n) => angle
3     d_phi = np.pi/n
4     phi_rel = phi - self.phi
5     if np.abs(phi_rel) <= epsilon:
6         return True #reached?
7     elif np.abs(phi_rel) < d_phi:
8         delta_phi = phi_rel #close?
9     elif np.abs(phi_rel) >= d_phi and phi_rel < 0:
10        delta_phi = -d_phi
11    else:
12        delta_phi = d_phi
13
14    for part in self.list_of_parts:
15        partVec = np.array([part.x_layout, part.y_layout])
16        baseVec = np.array([self.x_layout, self.y_layout])
17        radVec = partVec - baseVec
18        r = np.linalg.norm(radVec)

```

## 4 Implementierung

---

```
19     phi = np.arctan2(radVec[1], radVec[0])
20     pathVec = r*np.array([np.cos(phi+delta_phi),
21                           np.sin(phi+delta_phi)])-radVec
22     part.dx += pathVec[0]
23     part.dy += pathVec[1]
24     #<<position update like before>>
25     self.phi += delta_phi
26     return False
```

### 4.1.4 Subscriptions

Da zeitgleich mehrere Objekte der Klasse ‚Robot‘ auf die selbe Teilkomponente der Gesamtanlage zugreifen können, um z. B. Bauteile abzuholen, musste eine Methode implementiert werden, die dies verwaltet. Hierfür wird der Vorgang anhand der Klasse ‚Conveyor‘ erklärt, die eine solche Methode beinhaltet. Die Methode wird von einem Objekt der Klasse ‚Robot‘ aufgerufen. Als Erstes wird überprüft, ob diese Instanz bereits eine Bestellung aufgegeben hat, wenn nicht, wird im zweiten Schritt diesem eine Bestellung zugeteilt. Hierbei wird unterschieden, ob das Objekt etwas abholen oder aufgeben möchte. Wird in weiterer Folge eine Methode für die Abholung oder Aufgabe aufgerufen, wird das Objekt (der Besteller) wieder aus der Zuteilungsliste entfernt.

**Programm 4.3:** *Subscriptions*

```
1 def reserve_slot_for(self, intend, component_type, subscriber):
2     if subscriber in self.subscriptions.keys():
3         return self.subscriptions[subscriber]
4
5     elif intend == 'put' and component_type == 'part':
6         slot = self.get_slot_to_put_for(component_type)
7         if slot is None:
8             print(self.name, ': no more slots available (put)')
9             return
10        self.update_state()
11        self.subscriptions[subscriber] = slot
12        return slot
13
14    elif intend == 'take' and component_type == 'part':
15        slot = self.get_slot_to_take_from(component_type)
16        if slot is None:
17            print(self.name, ': no more slots available (take)')
18            return
19        self.update_state()
20        self.subscriptions[subscriber] = slot
```

## 4 Implementierung

---

```

21     return slot
22 else:
23     raise Exception('Invalid Intent')

```

### 4.1.5 Komplexe Rotation

Nun soll noch die Rotation mittels komplexer Zahlen gezeigt werden. Hierfür kann man sich der exponentiellen Darstellung komplexer Zahlen bedienen (Glg. 4.3), um somit die Rotation über eine einfache Multiplikation darstellen zu können (Glg. 4.4). Eine Skizze für die Rotation in der komplexen Ebene wird in Abb. 4.9 gezeigt. Hierbei handelt es sich um eine besonders elegante und kompakte Vorgehensweise.

$$\underline{z} = a + ib = |\underline{z}| e^{i \arg(\underline{z})} \quad (4.3)$$

$$\underline{z}_1 \underline{z}_2 = |\underline{z}_1| |\underline{z}_2| e^{i * (\arg(\underline{z}_1) + \arg(\underline{z}_2))} \quad (4.4)$$

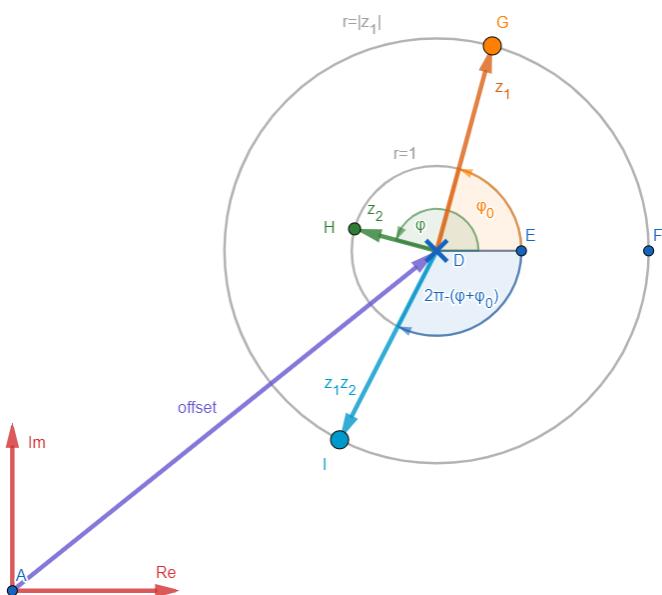


Abbildung 4.9: Rotation mittels komplexer Zahlen

**Programm 4.4:** Komplexe Rotation

```

1 def rotate_part(self, phi):
2     z2 = math.e**(-phi*1j) #gorgeous
3     self.update_points()
4     offset = complex(self.x_pixel, self.y_pixel)
5     new_points = []

```

## 4 Implementierung

---

```
6   for x,y in self.points:
7     z1 = (complex(x,y)-offset)
8     v = z2*z1 + offset
9     new_points.append(v.real)
10    new_points.append(v.imag)
11    self.canvas.coords(self.id, *new_points)
12    self.canvas.tag_raise(self.id)
```

### 4.1.6 Zweidimensionale Wegoptimierung

Um die Verfahrwege innerhalb der Simulation (und zum Teil auch an der Realanlage) gering zu halten, wird eine kleine Optimierung der Transitionsbewegung durchgeführt. Hierfür wird eine Minimierungsaufgabe für den Distanzvektor  $\vec{d}$  formuliert und gelöst (Abb. 4.10). Dabei werden die Positionen aus einer csv-Datei entnommen, welche durch die Vorgehensweise aus Kap. 3.4.4 gewonnen wurde. Die Koordinaten der csv-Datei sind dabei auf die Basis  $(v,w)$  bezogen, wie in Abb. 4.11 gezeigt wird. Um die Minimierungsaufgabe der Form:  $\min_{\varphi} |\vec{d}|$  zu formulieren, wird der Betrag des Distanzvektors bestimmt:

$$\begin{aligned}\vec{x}_g(\varphi) &= \vec{x}_b + \vec{\psi}(\varphi) \\ \vec{d}(\varphi) &= \vec{x}_s - \vec{x}_g(\varphi) = \vec{x}_s - \vec{x}_b - \vec{\psi}(\varphi) \\ |\vec{d}(\varphi)| &= \sqrt{(x_s - x_b - |\vec{\psi}| \cos(\varphi))^2 + (y_s - y_b - |\vec{\psi}| \sin(\varphi))^2}\end{aligned}$$

Die Aufgabe wurde mit Python-Bibliothek SCIPY gelöst. Der erstellte Programmcode ist im Prog. 4.5 ersichtlich. Diese Routine gibt eine *policy* zurück, wie sie im Simulationsprogramm verwendet wird.

**Programm 4.5:** Wegminimierung

```
1 import numpy as np
2 from scipy import optimize
3
4 pointArray = np.loadtxt('policies/data.csv', delimiter=';',
5                         skiprows=1)
6
7 def policy(**kwargs): #state as dictionary
8     idx = kwargs['parts_on_table']
9     pos = pointArray[idx]
10    vi = pos[0]
11    wi = pos[1]
12    beta = np.arctan2(wi, vi)
```

## 4 Implementierung

```

13     psi = np.sqrt(v**2 + w**2)
14     xs = kwargs['x_layout_robot']
15     ys = kwargs['y_layout_robot']
16     xb = kwargs['x_layout_table']
17     yb = kwargs['y_layout_table']
18
19     def F(phi):
20         return (x0-xb-psi*np.cos(phi))**2 + \
21                (y0-yb-psi*np.sin(phi))**2
22
23
24     res = optimize.minimize_scalar(F, bounds=(-np.pi, np.pi),
25                                    method='bounded')
26
27     phi = res.x
28     x = psi*np.cos(phi)
29     y = psi*np.sin(phi)
30
31
32     is_last = True if idx == len(pointArray)-1 else False
33     return {'phi': [beta-phi], 'x': [x],
34             'y': [y], 'is_last': is_last}

```

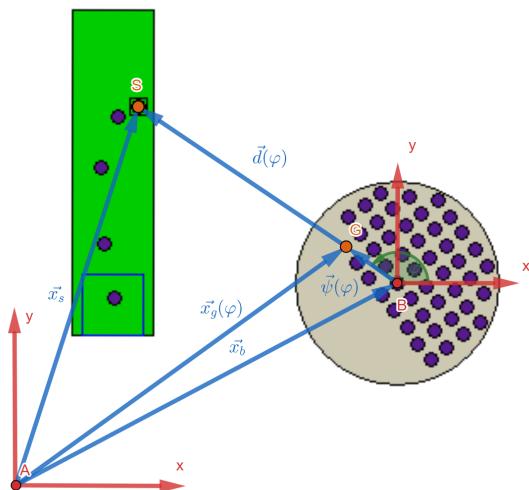


Abbildung 4.10: Wegvektor

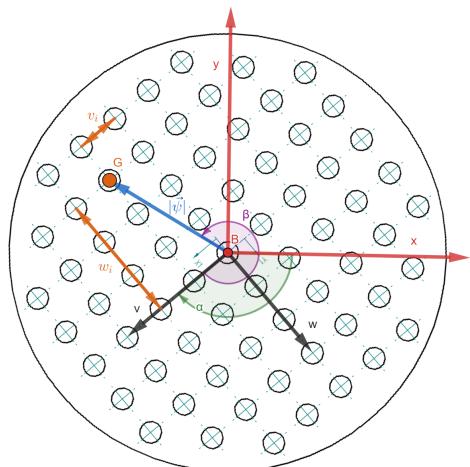


Abbildung 4.11: Koordinatenangabe

## 4.2 Anpassung an Trainingszelle

Es soll auch eine abgewandelte Form des Simulationsprogramms erstellt werden, welche den erstellten Kommunikationstreiber aus Kap. 3.5.1 beinhaltet. Das Programm wird nach erfolgreicher Anpassung an die Trainingszelle auf eine virtuelle Maschine (VM) (ähnlich der Entwicklungsumgebung) geladen, welche über ein virtuelles Netzwerk, Zugang zur Netzwerkkarte der SPS erhält. Somit kann nach erfolgreicher

Testung des Programms jederzeit verwendet werden, um dessen Funktionalitäten vorzuführen, ohne sich mit der Umgebungseinrichtung beschäftigen zu müssen. Im Kontext zur Automatisierungsspyramide (Kap. 2.1) befindet sich das Programm in der Prozessleitebene.

### 4.2.1 Layout

Es werden alle ‚simulation\_members‘, bis auf einen ‚Conveyor‘, einen ‚Rotator‘, einen ‚Table‘ und einen ‚Robot‘ entfernt. Die Anordnung ist durch Abb. 4.12 vorgegeben. Da Bauteile weder in das System eingebracht noch abgeführt werden, ist die Verwendung von Systemgrenzen obsolet. Daher wird eine neue ‚process\_policy‘ erstellt, welche die Bauteile zwischen B1 und B2 abwechselnd transferiert. Der Transfer wird invertiert, falls das gesamte Layout belegt ist, oder im Speicher keine Bauteile mehr vorhanden sind.

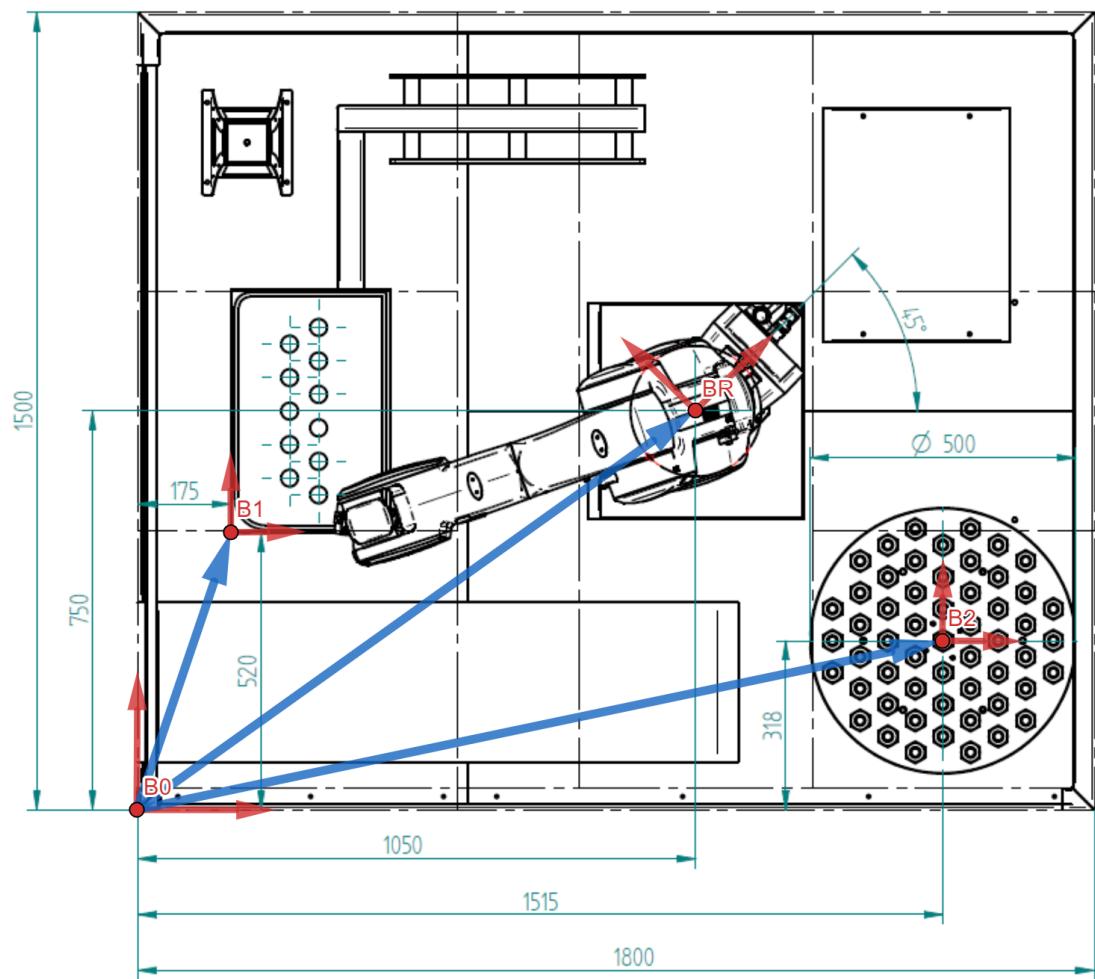


Abbildung 4.12: Layoutkonfiguration der Trainingszelle

### 4.2.2 Simulation und Steuerung

Die Steuerung der Trainingszelle wird über die neue „process\_policy“ abgewickelt, hierfür kann nun der Treiber für die Kommunikation dazugeschaltet werden (Abb. 4.14). Diese Zuschaltung sollte erst nach einer erfolgreichen Simulation erfolgen. Nach der Initialisierung befinden sich die zylindrischen Bauteile an den dafür vorgesehenen Positionen (Abb. 4.13). Da die Realanlage bei 100% *overdrive* sehr schnell agiert, wurde die Simulationsgeschwindigkeit auf 100 Pixel pro Aufrufzyklus erhöht (Prog. 4.1) und der Teiler für das Winkelinkrement der Rotation auf 16 reduziert (Prog. 4.2). Ein Video zur Steuerung befindet sich in einer YOUTUBE-Wiedergabeliste [40]. Die großen Verzögerungen ergeben sich zum einen durch die Datenübertragung und zum anderen durch einen Funktionsbaustein für die Koordinatenumrechnung.

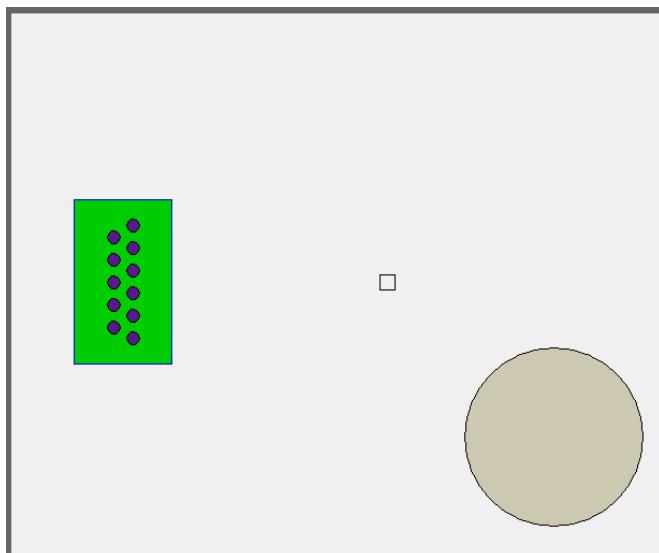


Abbildung 4.13: Simulationslayout

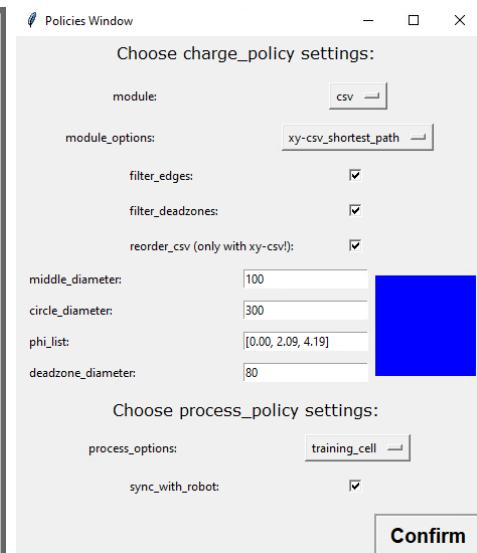


Abbildung 4.14: Policies

### 4.3 KI-Implementierung

Der Code für den *agent*, des *environments* und der *networks* sowie der Trainings und Testroutinen wurden in Python entwickelt. Der gesamte Quellcode ist auf GITHUB [41] frei zugänglich, daher wird auf die Implementierung des *environment* und des *agent* hier nicht weiter eingegangen. Es sei hier angemerkt, dass sich der Aufbau des Codes an Beispielen von der Pytorch-Webseite und an *open-source*-Projekten orientiert [42] [43] [44]. In diesem Kapitel werden jedoch die Trainingsroutine, und die neuronalen Netze präsentiert, welche für den Lernerfolg wesentlich sind. Im nächsten

## 4 Implementierung

---

Kapitel werden die Ergebnisse, abhängig von den gewählten Parametern, dargestellt und interpretiert.

### 4.3.1 Lernroutine

Nun wird der Aufbau der Lernroutine betrachtet. Zu Beginn werden die benötigten Bibliotheken geladen (Prog. 4.6). Der ‚Controller\_Agent‘ realisiert die entsprechende, im Kap. 3.4.2.2 vorgestellte, Methode. Der ‚Placement\_Agent‘ die erste Methode des selben Kapitels. Der ‚Benchmark\_Agent‘ (Kap. 3.4.2.5) ist ein *agent* ohne neuronales Netz und wurde zum einen für das Feintuning der *reward*-Funktion (Glg. 3.2) herangezogen, und zum anderen für die automatische Generierung von (s, s', r, d)-Tupel für die Anwendung von SL.

**Programm 4.6:** Trainingroutine: *Libraries*

```
1 import argparse
2 import json
3 from tensorboardX import SummaryWriter
4 from torchinfo import summary
5 from src.agent import Controller_Agent
6 from src.agent import Placement_Agent
7 from src.agent import Benchmark_Agent
8 from src.environment import Ruebris
```

Als nächstes werden die einzustellenden Parameter und *hyperparameter* benötigt. Diese werden mithilfe einer Funktion (Prog. 4.7) angelegt. Um den Platzbedarf zu kürzen, wurden die Hilfstexte der Parameter gelöscht, diesbezüglich wird auf den *source code* verwiesen [41]. Die meisten Bezeichnungen sollten sich selbst erklären, auf einige wird nun eingegangen. Der Parameter ‚-state\_report‘ bestimmt die Darstellungsart, welche dem neuronalen Netz übergeben wird. Es handelt sich dabei um die im Kap. 3.4.2.3 vorgestellten Zustandsdarstellungen. ‚-action\_method‘ bestimmt, neben ‚-reward\_action\_dict‘, den zu verwendeten *agent*, welcher vorgegeben durch den Parameter ‚-net\_type‘ den entsprechenden Netztyp (Kap. 3.4.2.4) verwendet. Im gesamten Versuch werden nur Netzwerke mit zwei Schichten verwendet, die Anzahl der Neuronen dieser Schichten kann jedoch mit den Parametern ‚-statescale‘ verändert werden. Bei den restlichen Parametern handelt es sich im Wesentlichen um die in Kapitel 2.6 vorgestellten *hyperparameter*.

**Programm 4.7:** Trainingroutine: *Arguments*

```
1 def get_args():
2     p = argparse.ArgumentParser(
3         '''Implementation of RL-Learning-Agent playing Ruebris'''
```

## 4 Implementierung

---

```
4 #environment variables:  
5 p.add_argument('--table_dia', type=int, default=420)  
6 p.add_argument('--block_size', type=int, default=20)  
7 p.add_argument('--state_report', type=str, default='full_float')  
8 p.add_argument('--action_method', type=str, default='placement')  
9 p.add_argument('--rnd_pos', type=bool, default=False)  
10 p.add_argument('--rnd_rot', type=bool, default=False)  
11 p.add_argument('--reward_action_dict', type=bool, default=False)  
12 p.add_argument('--c1', type=int, default=2)  
13 p.add_argument('--c2', type=int, default=50)  
14 p.add_argument('--c3', type=int, default=3)  
15 p.add_argument('--c4', type=int, default=5)  
16 #agent and network variables  
17 p.add_argument('--net_type', type=str, default='d-q-net')  
18 p.add_argument('--statescale_11', type=int, default=10)  
19 p.add_argument('--statescale_12', type=int, default=6)  
20 p.add_argument('--batch_size', type=int, default=128)  
21 p.add_argument('--lr', type=float, default=0.01)  
22 p.add_argument('--lr_decay', type=float, default=1.0)  
23 p.add_argument('--final_lr_frac', type=float, default=0.001)  
24 p.add_argument('--gamma', type=float, default=0.2)  
25 p.add_argument('--initial_epsilon', type=float, default=1)  
26 p.add_argument('--final_epsilon', type=float, default=0.1)  
27 p.add_argument('--epsilon_decay', type=float, default=0.99999)  
28 p.add_argument('--num_episodes', type=int, default=15000)  
29 p.add_argument('--memory_size', type=int, default=100000)  
30 p.add_argument('--burnin', type=int, default=1000)  
31 p.add_argument('--learn_every', type=int, default=1)  
32 p.add_argument('--sync_every', type=int, default=100)  
33 p.add_argument('--log_path', type=str, default='tensorboard')  
34 p.add_argument('--save_dir', type=str, default='trained_models')  
35 p.add_argument('--save_interval', type=int, default=70000)  
36 p.add_argument('--render', type=bool, default=False)  
37 #arguments for further training and supervised data:  
38 p.add_argument('--load_path', type=str,  
39                 default='trained_models/model.chkpt')  
40 p.add_argument('--continue_training', type=bool, default=False)  
41 p.add_argument('--new_epsilon', type=bool, default=True)  
42 p.add_argument('--create_superv_data', type=bool, default=False)  
43 p.add_argument('--save_dir_supervised_data', type=str,  
44                 default='supervised_data/')  
45 p.add_argument('--load_file_supervised_data', type=str,  
46                 default='benchmark_X/training_data_benchmark.obj')  
47 p.add_argument('--learn_from_data', type=bool, default=True)  
48 p.add_argument('--supervised_steps', type=int, default=50000)  
49 p.add_argument('--print_state_2_cli', type=bool, default=False)
```

## 4 Implementierung

---

```
50  
51     args = parser.parse_args()  
52     return args
```

Nun wird eine Trainingsroutine definiert (Prog. 4.8). Zu Beginn wird aufgrund der übergebenen Parameter der passende *agent* und das *environment* initialisiert. Anschließend wird zu Dokumentationszwecken eine Zusammenfassung des Netzwerks (net\_summary.txt) generiert und ausgegeben. Als nächstes wird, falls gewünscht, ein Vorabtraining gestartet. Hierbei wird das Netzwerk auf bereits vorhandene Daten trainiert, welche entweder durch das Script ‚manual\_mode.py‘ oder autonom über den ‚Benchmark\_Agent‘ generiert wurden. Im nächsten Schritt wird das Training für die vorgegebene Episodenanzahl durchgeführt. Die Funktionsweise sollte anhand des Codes ersichtlich sein, daher wird auf weitere Kommentare diesbezüglich verzichtet.

**Programm 4.8:** Trainingroutine: *Training*

```
1 def train(opt):  
2     writer = SummaryWriter(opt.log_path)  
3     env = Ruetrис(opt)  
4  
5     if not opt.reward_action_dict:  
6         state_dim = env.state_dim  
7         action_dim = env.action_dim  
8         agent = Controller_Agent(state_dim, action_dim, opt) if \  
9             opt.action_method=='controller' else \  
10            Placement_Agent(state_dim, action_dim, opt)  
11  
12     #make a nice summary of the network:  
13     model_stats = summary(agent.net, input_size=\  
14                             (opt.batch_size, state_dim))  
15     summary_str = str(model_stats)  
16     with open('net_summary.txt', 'w') as f:  
17         f.write(summary_str)  
18  
19     else:  
20         agent = Benchmark_Agent(opt)  
21  
22     #fill memory of Placement_Agent  
23     if isinstance(agent, Placement_Agent) and opt.learn_from_data:  
24         agent.load_memory(opt.load_file_supervised_data)  
25         supervised_steps = opt.supervised_training_steps  
26         for step in range(supervised_steps):  
27             q, loss = agent.learn()  
28             if q is not None: writer.add_scalar('SL/Mean', q, step)  
29             if loss is not None: writer.add_scalar('SL/Loss', loss, step)
```

## 4 Implementierung

---

```
30     lr = agent.optimizer.param_groups[0]['lr']
31     writer.add_scalar('SL/LR', lr, step)
32     agent.learn_from_data = False
33     agent.save(0)
34
35     episodes = opt.num_episodes
36
37     #####----- Mainloop: training the model -----#####
38     for epoch in range(episodes):
39         state = env.reset()
40         if not isinstance(agent, Controller_Agent):
41             state_action_dict = env.get_state_action_dict()
42
43         while True:
44             #1. Run agent on the state
45             action = agent.take_action(state) if \
46                     isinstance(agent, Controller_Agent) else \
47                     agent.take_action(state_action_dict)
48
49             #2. Agent performs action
50             p_s, next_state_action_dict, next_state, reward, done = \
51                     env.step(action, render=opt.render)
52             if p_s is not None: state = p_s
53
54             #3. Remember
55             agent.cache(state, next_state, action, reward, done)
56
57             #4. Learn
58             q, loss = agent.learn(epoch)
59
60             #5. Update state
61             state = next_state
62             if not isinstance(agent, Controller_Agent):
63                 state_action_dict = next_state_action_dict
64
65             #6. Check if end of game and log important data
66             if done:
67                 writer.add_scalar('T/Score', env.score, epoch)
68                 writer.add_scalar('T/Tetro', env.parts_on_board, epoch)
69                 writer.add_scalar('T/Wasted', env.wasted_places, epoch)
70                 writer.add_scalar('T/Holes', env.holes, epoch)
71                 if not isinstance(agent, Benchmark_Agent):
72                     writer.add_scalar('T/Epsilon', agent.epsilon, epoch)
73                     lr = agent.optimizer.param_groups[0]['lr']
74                     writer.add_scalar('T/Learning_Rate', lr, epoch)
75                     if q is not None: writer.add_scalar('T/Mean', q, epoch)
```

## 4 Implementierung

---

```
76     if loss is not None: writer.add_scalar('T/L', loss, epoch)
77     break
```

Um die Routine vom Command Line Interface (CLI) aus starten zu können, wird noch eine *main*-Routine/Definition benötigt (Prog. 4.9). Diese Routine wird beim alleinigen Start des Programms ausgeführt. Sie erstellt die zu übergebende Parameterliste, welche als ‚train\_config.txt‘-File zusätzlich abgelegt wird und startet die Trainings-Routine. Die abgelegte Parameterliste wird anschließend für die Testroutine und für die Einbindung in das Simulationsprogramm herangezogen, um die Verwendung benutzerfreundlicher zu gestalten.

**Programm 4.9:** Trainingroutine: *Main*

```
1 if __name__ == '__main__':
2     opt = get_args()
3     # saves arguments to config.txt file
4     with open('train_config.txt', 'w') as f:
5         json.dump(opt.__dict__, f, indent=2)
6     train(opt)
```

### 4.3.2 Neuronale Netzwerke

In diesem Abschnitt wird der Aufbau der neuronalen Netzwerke betrachtet. Wird eine Änderung der Netzwerkarchitektur gewünscht, muss diese in dem entsprechenden Modul ‚networks‘ durchgeführt werden. Dies betrifft die Initialisierung der Gewichtungsfaktoren, der Anzahl an *layers* und weitere Neuronen pro *layer*. Es wird nun wieder mit der Einbindung der benötigten Bibliotheken begonnen (Prog. 4.10). Für das Netzwerk wird primär das ‚nn-Modul‘ (neural network) benötigt. Für die Bestimmung des Mittelwerts und zur Erstellung von Netzwerkkopien werden noch die entsprechenden Funktionen importiert.

**Programm 4.10:** Networks: *Libraries*

```
1 import torch.nn as nn
2 import torch.mean as mean
3 import copy
```

Als Erstes wird ein Netz für die Bestimmung von *q/state-values* erstellt. Dieses Netzwerk wird sowohl vom ‚Controller-Agent‘ als auch vom ‚Placement-Agent‘ verwendet. Außerdem wird es zum einen für DQL und zum anderen für Double-DQL verwendet, wie es im Kap. 2.6.4.7 dargelegt wurde. Bei der verwendeten Methode ‚kaiming\_normal()‘ handelt es sich um die vorgestellte He-Initialisierung (Kap. 2.6.1.4).

## 4 Implementierung

---

**Programm 4.11:** Networks: DQN- and DDQN-Class

```
1 class Policy_QN(nn.Module):
2     def __init__(self, i_dim, o_dim, opt):
3         super(Policy_DDQN, self).__init__()
4         s1 = opt.statescale_l1
5         nodes_l1 = int(i_dim*s1)
6         s2 = opt.statescale_l2
7         nodes_l2 = int(i_dim*s2)
8
9         self.online = nn.Sequential(\n10             nn.Linear(i_dim, nodes_l1), nn.ReLU(),\n11             nn.Linear(nodes_l1, nodes_l2), nn.ReLU(),\n12             nn.Linear(nodes_l2, o_dim))\n13
14     self._create_weights()\n15     self.target = copy.deepcopy(self.online)\n16
17     #freeze parameters of self.target\n18     for p in self.target.parameters():\n19         p.requires_grad = False\n20
21     def _create_weights(self):\n22         for m in self.modules():\n23             if isinstance(m, nn.Linear):\n24                 nn.init.kaiming_normal_(m.weight, mode='fan_in',\n25                                         nonlinearity='relu')\n26                 nn.init.constant_(m.bias, 0)\n27
28     def forward(self, state, model='online'):\n29         if model == 'online':\n30             return self.online(state)\n31         elif model == 'target':\n32             return self.target(state)
```

Als nächstes wird ein Netzwerk ausschließlich für die Bestimmung von *q-values* erstellt. Es handelt sich um die Architektur für das Dueling-Double-DQL (Kap. 2.6.4.7), wobei die *q-values* über *advantage* und *state-value* bestimmt werden. Dieses Netzwerk wird ausschließlich vom ‚Controller-Agent‘ verwendet. Die Methode zur Initialisierung der Gewichtungsfaktoren ist gleich aufgebaut wie in der Klasse ‚Policy\_QN‘ und wurde daher weggelassen.

**Programm 4.12:** Networks: DDDQN-Class

```
1 class Policy_DDDQN(nn.Module):\n2     def __init__(self, i_dim, o_dim, opt):\n3         super(Policy_DDDQN, self).__init__()
```

## 4 Implementierung

---

```
4     s1 = opt.statescale_l1
5     nodes_l1 = int(i_dim*s1)
6     s2 = opt.statescale_l2
7     nodes_l2 = int(i_dim*s2)
8
9     #definition online network
10    self.fc_online = nn.Sequential(\n11        nn.Linear(i_dim, nodes_l1), nn.ReLU(),\n12        nn.Linear(nodes_l1, nodes_l2), nn.ReLU())
13    self.value_online = nn.Sequential(\n14        nn.Linear(nodes_l2, 1))
15    self.adv_online = nn.Sequential(\n16        nn.Linear(nodes_l2, o_dim))
17
18    self._create_weights()
19
20    #creating target network
21    self.fc_target = copy.deepcopy(self.fc_online)
22    self.value_target = copy.deepcopy(self.value_online)
23    self.adv_target = copy.deepcopy(self.adv_online)
24
25    #freeze parameters of target network
26    for p in self.fc_target.parameters():
27        p.requires_grad = False
28    for p in self.value_target.parameters():
29        p.requires_grad = False
30    for p in self.adv_target.parameters():
31        p.requires_grad = False
32
33    def forward(self, state, model='online'):
34        if model == 'online':
35            x = self.fc_online(state)
36            v = self.value_online(x)
37            adv = self.adv_online(x)
38            adv_avg = mean(adv, dim=1, keepdims=True)
39            q = v + adv - adv_avg
40
41        if model == 'target':
42            x = self.fc_target(state)
43            v = self.value_target(x)
44            adv = self.adv_target(x)
45            adv_avg = mean(adv, dim=1, keepdims=True)
46            q = v + adv - adv_avg
47
48    return q
```

## 4.4 KI-Training

In diesem Kapitel werden nun einige Parametersätze gewählt, um damit die Trainingsroutine zu starten. Aus Platzgründen werden jedoch nur die veränderten Werte in den dafür vorgesehenen Tabellen angeführt. Die vollständigen Parametersätze, jedoch ohne lange Dateipfade, sind in den Tabellen 12.7, 12.8, 12.9 und 12.10 im Anhang ersichtlich und entsprechen dem Informationsgehalt aus den generierten „train\\_config.txt“-Files. Die Anzahl an freien Parametern des Netzwerks sind dem „net\\_summary.txt“-File zu entnehmen. Für mehr Details wird an dieser Stelle wieder auf das Github-*repository* [41] verwiesen. Für die anfängliche Parameterwahl wurden die Vorschlägen eines MATHWORKS-Dokuments [45] herangezogen. Als Optimierer wird Adam (Glg. 2.16) verwendet und die aus den Updates der Gewichtungsfaktoren resultierenden Werte per *episode* dargestellt. Diese Werte sind:

- **Epsilon:** Parameter für die Auswahl von zufälligen Aktionen nach der *epsilon-greedy-policy* (Kap. 2.6.4.2 , Glg. 2.45).
- **Holes:** Anzahl an unbelegten Plätzen nach Abschluss der *episode*.
- **Learning Rate:** Parameter  $\alpha$  aus Kap. 2.6.1.3.
- **Loss:** Ausgewertetes Kostenfunktional (Kap. 2.6.1.5) für einen *batch* aus dem *memory* nach Abschluss der *episode*.
- **Score:** Kumulierte *rewards* nach Abschluss der *episode* (Glg. 2.44, 3.2).
- **Tetrominos:** Anzahl an abgelegten Bauteilen nach Abschluss der *episode*.
- **Wasted Places:** Anzahl an Plätzen die durch ungünstiges Ablegen keine weitere Ablage mehr zulassen (Kap. 3.4.2.1).
- **Mean:** Der Durchschnittliche Wert des Netzwerksausganges (*state* bzw. *q-value*) für einen *batch*.

### 4.4.1 Placement Agent

Bei den ersten Versuchsdurchführungen wird der „Placement-Agent“ (Kap. 3.4.2.2) herangezogen. Die dabei verwendeten Parameter befinden sich in Tabelle 4.1 und 4.2. Bei dem Versuch mit der ID-Nummer 001 handelt es sich um den „Benchmark-Agent“ welcher verwendet wurde, um  $(s, s', r, d)$ -Tupel zu generieren. Aufgrund dessen sind die meisten Parameter hierbei auch nicht relevant. Diese Tupel wurden zu einem späteren Zeitpunkt für die Vortrainings-Phase verwendet (Prog. 4.8),

## 4 Implementierung

---

zunächst werden jedoch die Unterschiede und Auswirkungen der verschiedenen Zustandsdarstellungen (Kap. 3.4.2.3) betrachtet. Hierfür werden Trainingsparameter gewählt, welche sich nur in den *observations* unterscheiden (Tab. 4.1).

**Tabelle 4.1:** Parametersätze: Placement-Agent

Bezeichnung-ID	ID001	ID008	ID009	ID010
-state_report	-	full_float	reduced	full_bool
-statescale_l1	-	10	60	5
-statescale_l2	-	6	30	3
-batch_size	-	128	128	128
-lr $\alpha$	-	0.01	0.01	0.01
-lr_decay	-	1.0	1.0	1.0
-final_lr_frac	-	-	-	-
-gamma $\gamma$	-	0.2	0.2	0.2
-initial_epsilon	-	1.0	1.0	1.0
-final_epsilon	-	0.1	0.1	0.1
-epsilon_decay	-	0.99999	0.99999	0.99999
-supervised	-	False	False	False
Anzahl der Gewichtungsfaktoren	-	19 600 509	746 401	22 398 919

Die aus diesen Parametern resultierenden Kurven, welche in Prog. 4.8 definiert wurden, sind in Abb. 4.15 dargestellt. Diese Graphen zeigen einen klaren Favoriten: die Zustandsdarstellung ‚reduced‘. Dies lässt sich dahingehend erklären, dass sich der *reward* aus dieser Darstellung indirekt leichter bestimmen lässt als bei den anderen Darstellungsweisen, auch wenn die Information der Porosität (Kap. 3.4.2.1) völlig verloren geht. Des Weiteren ist es interessant, dass die Darstellung mit ausschließlich boolschen Variablen der Darstellung mittels Gleitkommadarstellung überlegen ist, obwohl die Anzahl an Gewichtungsfaktoren in etwa gleich sind. Das Verhalten dieser drei *agents* ist durchwegs sehr unterschiedlich und wurde zu Dokumentationszwecken der erstellten Youtube-Wiedergabeliste [40] hinzugefügt.

## 4 Implementierung

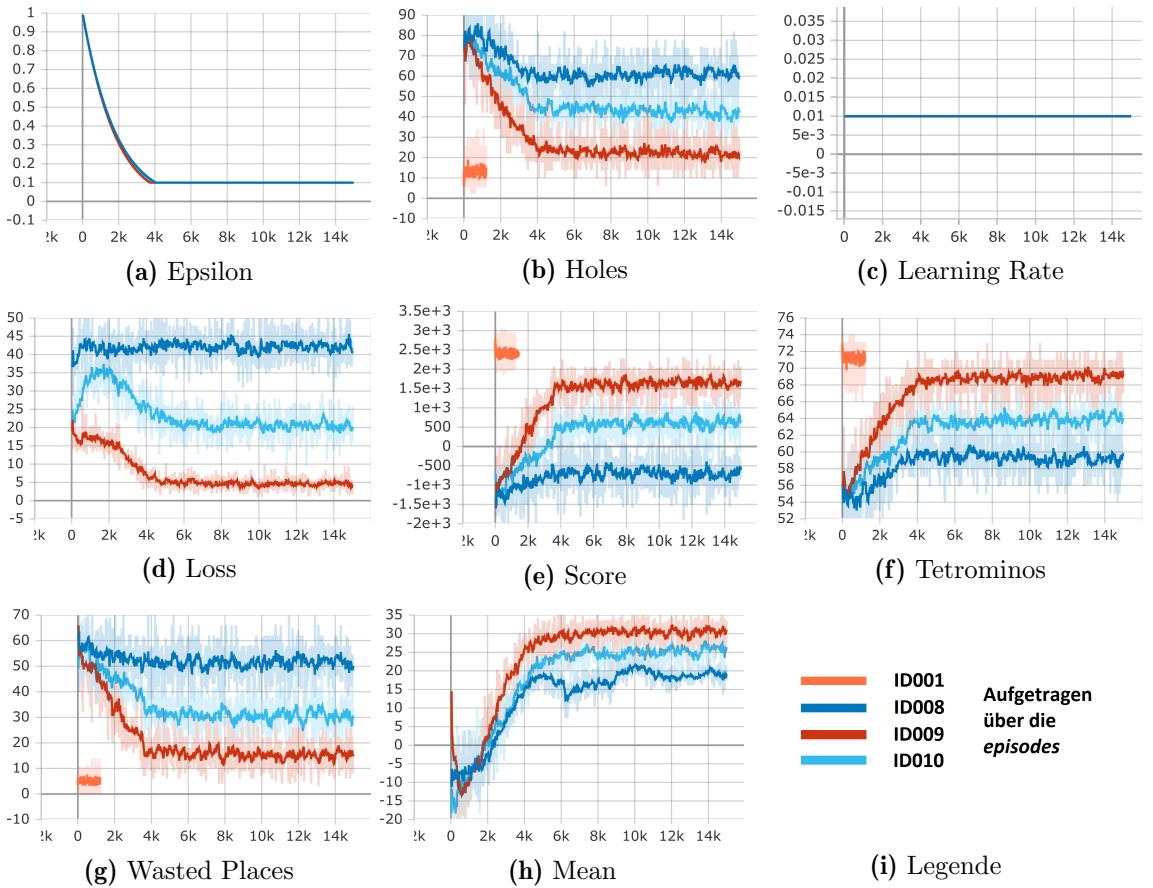


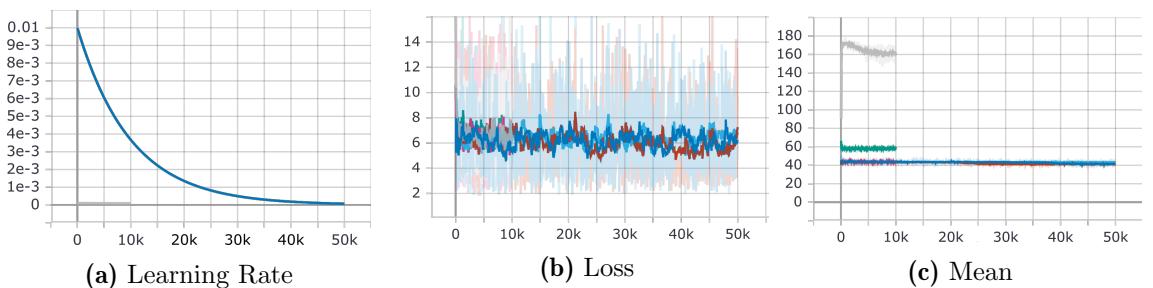
Abbildung 4.15: Haupttraining: Placement-Agent

Nun stellt sich auf natürliche Weise die Frage, wie die schlechte Performance, bedingt durch die Zustandsdarstellung ‚full\_float‘ verbessert werden kann. Hierfür wurde das Konzept der Vortrainings-Phase implementiert, welches für erste Versuche herangezogen wurde. Die Parameter dieser Versuchsdurchführungen befinden sich in Tab. 4.2. Die Varianz der Einstellungen soll weitere Einblicke in das Lernverhalten ermöglichen, wie die Auswirkung der Neuronendichte oder der *batch size*.

**Tabelle 4.2:** Parametersätze: Placement-Agent SL

Bezeichnung-ID	ID001	ID002	ID003	ID004	ID005	ID006	ID007
-statescale_l1	-	10	5	20	5	5	5
-statescale_l2	-	6	3	12	3	3	3
-batch_size	-	128	256	128	128	128	128
-lr $\alpha$	-	0.01	0.0001	0.01	0.01	0.01	0.01
-lr_decay	-	0.9999	0.99999	0.9999	0.9999	0.9999	0.9999
-final_lr_frac	-	0.001	0.01	0.001	0.001	0.01	0.01
-gamma $\gamma$	-	0.2	0.8	0.2	0.2	0.2	0.4
-initial_epsilon	-	0.01	0.5	0.01	0.01	0.5	0.4
-final_epsilon	-	0.001	0.02	0.001	0.001	0.01	0.03
-epsilon_decay	-	0.999999	0.999992	0.999999	0.999999	0.999995	0.99999
-supervised	-	True	True	True	True	True	True
-supervised_steps	-	10000	-	50000	50000	10000	10000

Durch die Vortrainings-Phase sind nun neben den bisherigen Graphen auch weitere verfügbar (Abb. 4.16), welche die *learning rate*, *loss* und den *mean value* der Zustände im *batch*, aufgetragen über die Gradientenaufrufe, enthalten. Der *loss*, welcher für die Updates der Gewichtungsfaktoren ausschlaggebend ist (Kap. 2.6.1.3), erreicht dabei innerhalb weniger Iterationen einen kleinen Wert. Dem neuronalen Netz, so zumindest die Idee, sollte nun das Verhalten des ‚Benchmark-Agents‘ eingeprägt worden sein.



**Abbildung 4.16:** Vorabtraining: Placement-Agent SL

## 4 Implementierung

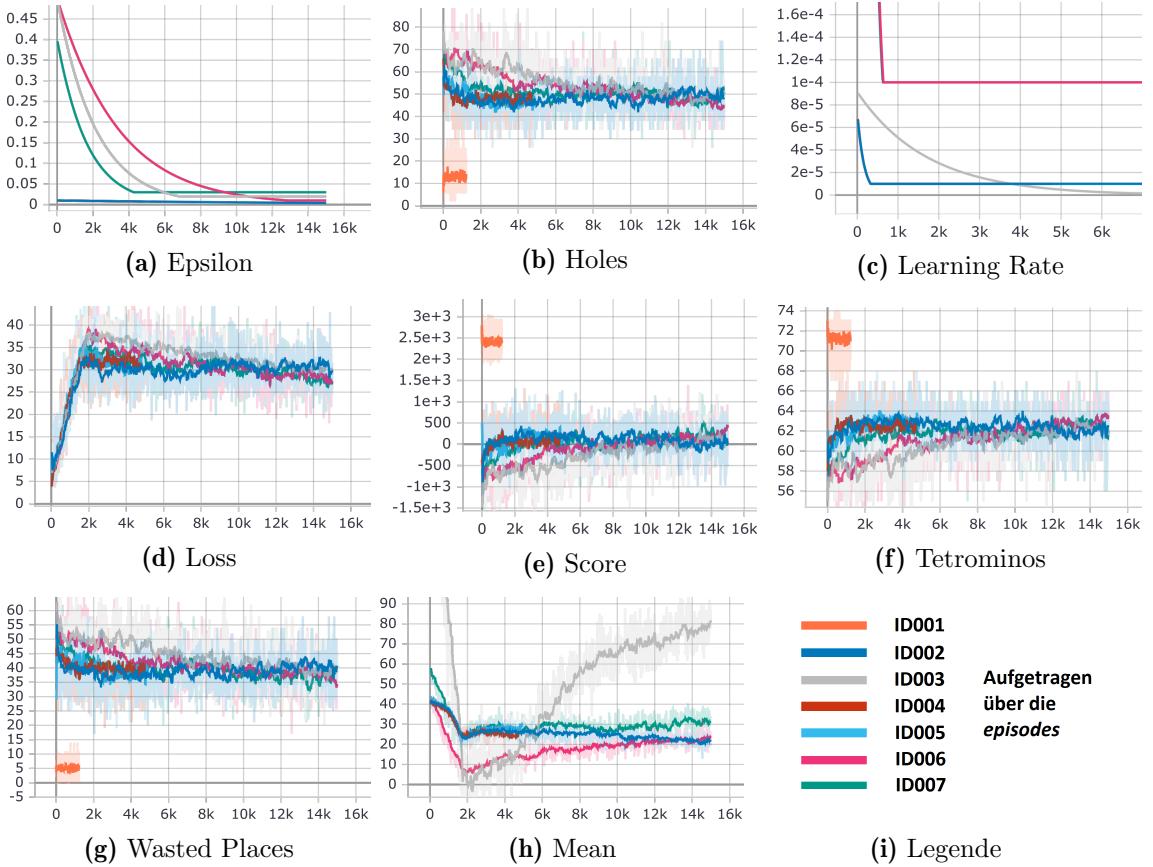


Abbildung 4.17: Haupttraining: Placement-Agent SL

Abbildung 4.17 zeigt die Ergebnisse des Haupttrainings nach Beendigung des Vortrainings. Zwar wurden die Erwartungen, dass durch SL das neuronale Netz verbessert wird, erfüllt, jedoch zeigen alle Einstellungen der Parameter sehr ähnliches Verhalten (Das Verhalten der Versuchsnummer 002 wurde der Wiedergabeliste hinzugefügt). Die Versuche mit den IDs 006, 007 und 003 können noch nicht als auskonvergiert betrachtet werden und würden eine weitere Versuchsdurchführung benötigen. Nichtsdestotrotz konnte der *loss* von rund 40 auf 30 verringert und der Score von rund -750 auf +100 verbessert werden. Für weitere Vergleiche wird an dieser Stelle auf die entsprechenden Graphen (Abb. 4.15 und 4.17) verwiesen.

### 4.4.2 Controller Agent

Für weitere Untersuchungen wird der ‚Controller-Agent‘ herangezogen. Für die entsprechende Interaktionsmethode (Kap. 3.4.2.2) mussten noch *rewards* für die einzelnen Aktionen dem *environment* hinzugefügt werden. Tabelle 4.3 ordnet die im *environment* definierten Aktionen den entsprechenden *rewards* zu. Der Ausdruck ‚overlapping‘ ist wahr, wenn das Bauteil nun über einen bereits belegten Platz schwebt.

## 4 Implementierung

---

Der Ausdruck ‚boundary‘ ist wahr, falls das Bauteil die Chargierfläche (Matrix) verlassen würde. Mit dem Ausdruck ‚OK‘ wird an dieser Stelle nur ausgedrückt, dass keine der beiden Bedingungen zutreffend ist. Für weitere Details wird wieder auf das Github-*repository* verwiesen [41].

**Tabelle 4.3:** *Rewards* der Einzelaktionen

Aktion	move_action()	rot_action()	table_action()	place_action()
<i>Reward</i> if overlapping	-2	-2	-	-500
<i>Reward</i> if boundary	-4	-4	-	-
<i>Reward</i> if OK	-1	-1	-3	$R_i$ (Glg. 3.2)

Mit den jeweiligen *rewards* der Einzelaktionen und den bisher bekannten Parametern kann die Trainingsroutine des ‚Controller-Agent‘ gestartet werden. Hierfür werden wieder einige Variationen der Einstellungen (Tab. 4.4) durchgerechnet. Als Erweiterung zur Trainingsroutine des ‚Placement-Agent‘ kann nun auch ein Netztyp zur Bestimmung der *q-values* (Kap. 2.6.4.7) und eine zufällige Startposition des Bauteils gewählt werden.

**Tabelle 4.4:** Parametersätze: Controller-Agent

Bezeichnung-ID	ID011	ID012	ID013	ID014	ID015	ID016
-state_report	reduced	reduced	full_bool	full_bool	full_float	reduced
-net_type	dd-q-net	ddd-q-net	ddd-q-net	ddd-q-net	ddd-q-net	ddd-q-net
-rnd_pos & rot	False	False	False	True	True	True
-statescale_l1	60	60	5	5	10	60
-statescale_l2	40	40	3	3	6	40
-batch_size	128	128	64	64	64	64
-lr_decay	1.0	1.0	0.999999	0.999999	1.0	1.0
-final_lr_frac	-	-	0.1	0.1	-	-
-gamma $\gamma$	0.9	0.9	0.8	0.8	0.5	0.5
-final_epsilon	0.01	0.01	0.1	0.08	0.08	0.08

## 4 Implementierung

Wie Abbildung 4.18 zeigt, kann mit den gewählten Einstellungen kein zufriedenstellendes Ergebnis erzielt werden. Auch bei dieser Versuchsdurchführung schneidet die Zustandsdarstellung ‚reduced‘ am besten ab, welche jedoch nach Beendigung des Trainings faszinierende Verhaltensmuster offenlegt, wie eine ewige Rotation des Bauteils: Smashboy (Tab. 3.1). Diesem Verhaltensmuster ist es auch geschuldet, dass die Netzwerke mit der ID 011 und 012 ausgiebig Minuspunkte sammeln. Wie nach dem Versuch mit dem ‚Placement-Agent‘ zu erwarten, ist hier die Zustandsdarstellung von ‚full\_bool‘ der von ‚full\_float‘ überlegen. An dieser Stelle sei noch angemerkt, dass der Versuch mit der ID 013, der in den Graphen leider etwas untergegangen ist, einen guten Eindruck macht, was die Lösungsstrategie der Aufgabe betrifft. Eine Auswahl an Netzwerken befindet sich in der Youtube-Wiedergabeliste [40].

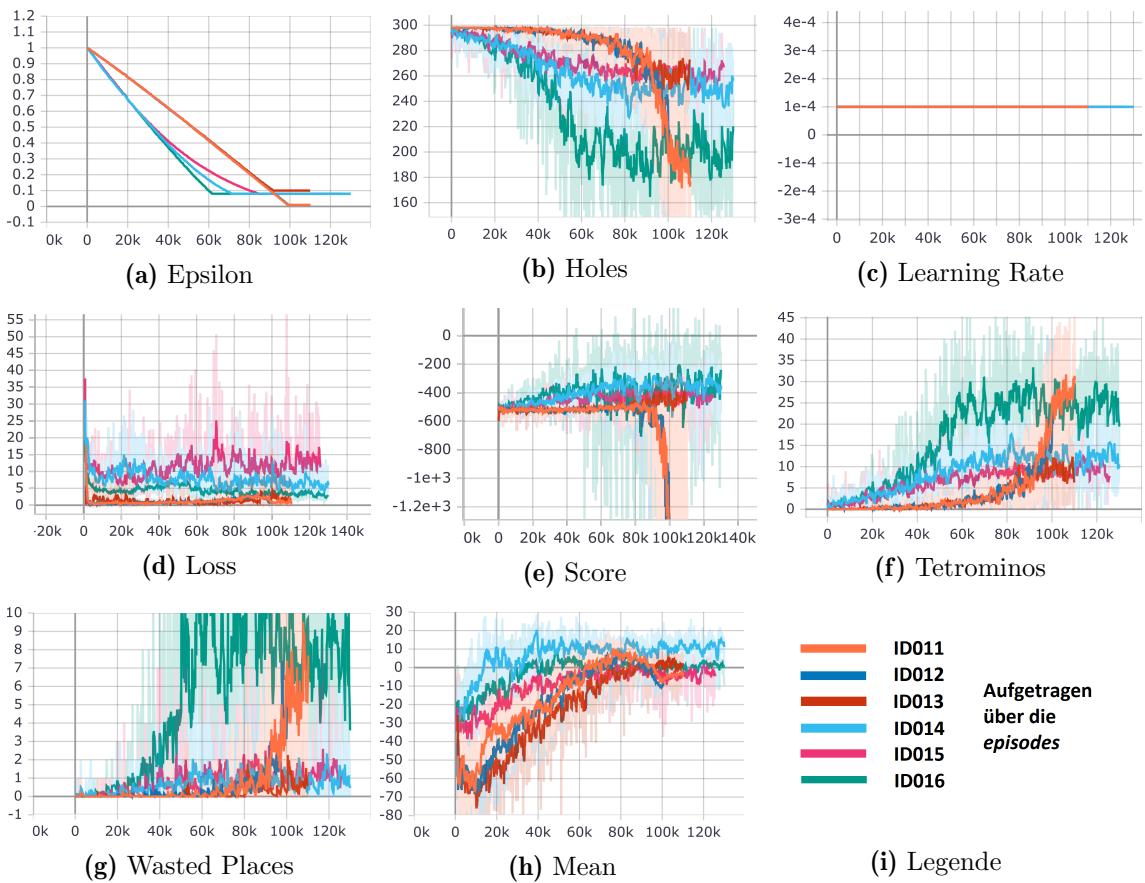


Abbildung 4.18: Haupttraining: Controller-Agent

### 4.4.3 Conclusio

Abschließend zu diesem Kapitel wird noch kurz auf die erzielten Ergebnisse und auf weitere Möglichkeiten für Versuche eingegangen. An dieser Stelle sei angemerkt, dass sich die in diesem Kapitel generierten Daten auch auf Github befinden, jedoch aus Speicherplatzgründen in einem eigenen *repository* [46] abgelegt wurden.

#### Zu den Ergebnissen

Auch wenn bei den Versuchen mit dem ‚Placement-Agent‘ ein relativ gutes Ergebnis mithilfe der Zustandsdarstellung ‚reduced‘ erzielt wurde, konnte der *highscore* des ‚Benchmark-Agents‘ nicht überboten werden. In den meisten Fällen konnte im manuellen Modus dies jedoch mit gewisser Taktik (Kap. 12.6) erreicht werden. Im Zuge der Versuchsdurchführung, besonders durch den ‚Controller-Agent‘, hat sich die Frage aufgetan, ob eine Identifizierung des Bauteils durch die Angabe der Flächenschwerpunkte für das Netzwerk möglich ist, oder weitere Informationen dieser Zustandsdarstellung hinzugefügt werden müssten. Zum Beispiel könnte man nicht nur Abstände der Flächennormalen, sondern auch die Diagonalabstände der Kanten verwenden. Ob dies das Verhalten verbessern würde, konnte im Zuge dieser Arbeit nicht mehr ausgetestet werden und gilt aus diesem Grund auch als fragwürdig. Jedenfalls bieten die erarbeiteten Ergebnisse eine Basis für weitere Nachforschungen.

#### Weitere bestehende Möglichkeiten

Zwar wurde die Möglichkeit implementiert, bereits gespeicherte Netzwerke zu laden und eine weitere Trainingsroutine durchzuführen, jedoch wurde darauf in dieser Arbeit verzichtet. Des Weiteren besteht die Möglichkeit, auch für andere Zustandsdarstellungen die vorgestellte SL-Implementierung anzuwenden. Hierfür müsste der ‚Benchmark-Agent‘ nur mit der dafür vorgesehenen Option einige Zeit laufen gelassen werden, um die dafür nötigen Daten zu generieren. Auch wäre es denkbar, einen solchen *agent* für die ‚controller‘-Methode anzulegen, indem die Ablageposition mithilfe der ‚placement‘-Methode vorgegeben wird und die Aktionen, die in diesen Zustand münden, berechnet werden. Eine weitere, bereits implementierte Möglichkeit für die Generierung dieser Daten wäre die Verwendung des Scripts ‚manual\_mode.py‘. Diese Script ermöglicht mit dem *environment* als Mensch zu interagieren und die Aktionen und Zustände aufzuzeichnen.

### Weitere zu implementierende Möglichkeiten

Im Kapitel 2.6.1.8 wurde die Bibliothek Raytune erwähnt um die Parameter des Netzwerks zu optimieren. Auf die Implementierung wurde dahingehend verzichtet, da ein Graphics Processing Unit (GPU)-Cluster benötigt werden würde, um die Optimierung effizient betreiben zu können. Ein solcher Cluster war nicht verfügbar, jedoch könnte man über Dienste wie die GOOGLE CLOUD eine rechen-starke VM beziehen. Ein weiterer Punkt für mögliche Implementierungen wäre die Erweiterung durch verschiedene *agents* und Netzwerkstrukturen. Hierfür wurden im Kap. 2.6.4.5 mehrere Möglichkeiten präsentiert (Abb. 2.41).

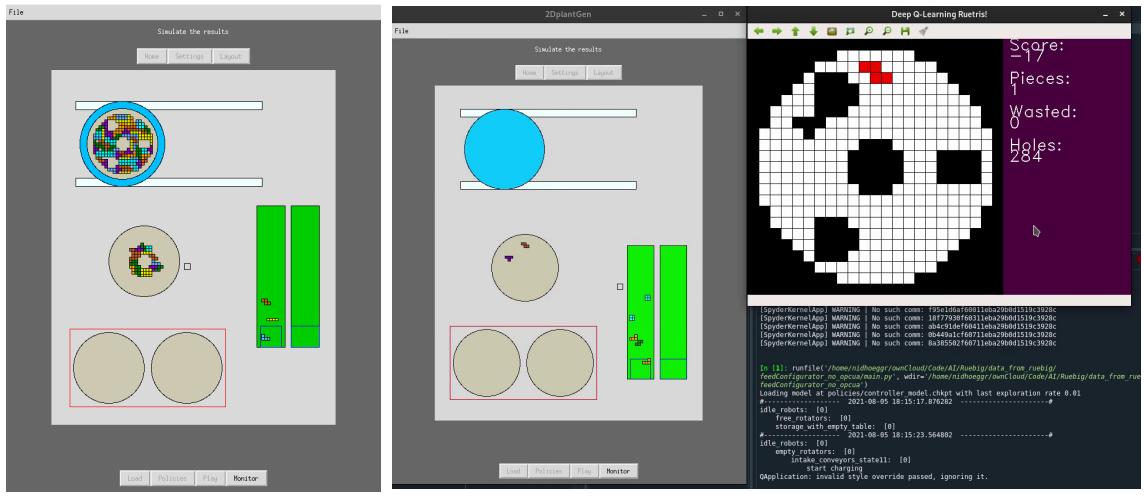
### Persönliche Meinung

In diesem Abschnitt möchte ich noch etwas von der eigenen Meinung miteinbringen. Zunächst möchte ich betonen, dass all die Themen dieser Arbeit für mich Neuland waren, besonders jedoch war die Erarbeitung des Kapitels ML und die nachfolgende Konzeptionierung einer Implementierung herausfordernd. Der Komplexität und den zeitlichen Ressourcen geschuldet, konnte ich leider dabei nicht alle Ideen ausprobieren. Vielleicht komme ich in Zukunft dazu, diese Kapitel weiter zu verfeinern, oder ein anderer Interessent findet Zeit, sich mit Verbesserungen auseinanderzusetzen. Zu den Stärken und Schwächen des RL möchte ich an dieser Stelle noch auf einen interessanten Blog-Post hinweisen [47].

## 4.5 Einbindung in das Simulationsprogramm

Um die Einbindung in das Simulations- und Steuerungsprogramm einfach zu halten, wird für die Implementierung das ‚train\\_config.txt‘-File herangezogen. Mit dessen Hilfe reduziert sich der Wechsel von neuronalen Netzen auf das Umbenennen dieser beiden Dateien und der Ablage in den dafür vorgesehenen *policies*-Ordner (Abb. 3.6). In den Programmrutinen wird bei der Ausführung das *environment*-Objekt entsprechend angelegt und der passende *agent* gewählt. In der ‚Placement-Agent‘-Klasse wird dabei Epsilon mit Null initialisiert und bei der ‚Controller-Agent‘ mit ungleich Null. Die Initialisierung mit ungleich Null wird aus den in Kap. 4.4.2 erwähnten Gründen so gewählt. Abbildung 4.19 zeigt die Verwendung der *agents* innerhalb der Simulationsumgebung, eine Aufzeichnung des ‚Placement-Agents‘ befindet sich in der Wiedergabeliste [40]. Bei der Verwendung des ‚Controller-Agents‘ wird das *environment* aufgerufen, welches vom *agent* gesteuert wird, bis das momentane Bauteil abgelegt wurde. Anschließend werden die Bewegungen in die Simulation übernommen.

## 4 Implementierung



**Abbildung 4.19:** Einbindung der *agents*

## **5 Zusammenfassung und Ausblick**

Ziel dieser Arbeit war das Erstellen eines benutzerfreundlichen Interaktionsschemas für die Chargierung von Härteanlagen und die dabei erforderlichen Datenmodelle für die optimale Chargierung selbst zu finden. Um diese Aufgabe, bezogen auf das Interaktionsschema, zu lösen, wurde eine Prozesskonfiguration und eine Steuerungslogik für dessen Steuerung entwickelt. Diese Lösung wurde in einem nächsten Schritt auf eine Roboter-Trainingszelle angewendet um deren Funktionalität zu beweisen. Für die Fragestellung nach geeigneten Datenmodellen für die optimale Chargierung wurden die Bauteile durch einfache Geometrien approximiert. Diese approximierten Bauteile wurden anschließend durch vorgegebene Positionen, oder über Umwege durch Auswertungen neuronaler Netze, auf die Chargiergestelle abgelegt. Für die Katalogisierung von komplexeren Bauteilen wurde dabei die Form von Tetrominos herangezogen.

Notwendige Programm Routinen für das Training der neuronalen Netze wurden erstellt und in einer ersten Iteration verwendet, welche nun auch online zur Verfügung stehen.

Die Neuheit dieser Arbeit besteht in der Anwendung von neuronalen Netzen, um das Parkettierungsproblem der Chargiergestelle von Härteanlagen automatisiert und in entsprechender Zeit zu lösen. In der ersten Versuchsdurchführung konnte bereits ein brauchbares Datenmodell für diese Aufgabe erstellt werden, welches jedoch viel Raum für weitere Verbesserungen bietet. Dieses Modell wurde in das erstellte Konfigurationsprogramm integriert und ließe sich somit auf Realanlagen übertragen.

Da die erstellten Programm Routinen noch einiges an Verbesserungspotenzial bieten, wäre, in einer weiteren Iteration, eine umfangreichere Implementierung innerhalb eines Projektteams sinnvoll. Die Erstimplementierung ist zwar nicht für die Endanwendung konzipiert, eignet sich jedoch als Demonstrator, um weitere Projekte daraus abzuleiten.

### **Prozess-Ablaufs-Optimierung**

Bei der Erstellung der Software wurde darauf geachtet, dass sich das Gesamtsystem mithilfe von Zuständen und deren Transitionen abbilden lässt. Somit wäre auch der Weg, eine KI den Prozessablauf anlernen zu lassen, geeignet. Die Idee wäre, ein Anlagenlayout zu erstellen und sie als Lernumgebung für die KI exportieren zu können. Ist der Lernprozess durchgeführt, könnte man die generierte *policy* zurück in die Umgebung laden und beurteilen. Durch diese Vorgehensweise erscheint eine Vollautomatisierung beliebiger Anlagenlayouts mittels KI durchaus realisierbar.

### **Greiftaktik**

Völlig unberücksichtigt in dieser Arbeit wurde das Thema des autonomen Greifens. Um den Prozess, wie in dieser Arbeit vorgestellt, durchführen zu können, wäre eine geeignete Strategie diesbezüglich erforderlich. Eine relativ einfache Möglichkeit wäre, einen Greifer mit adaptierbaren Saugnäpfen auszustatten, die sich der Form von Tetrominos anpassen.

### **Identifizierung weiterer Bauteilklassen**

In dieser Arbeit wurde das Konzept UL im Kap. 2.6.3 kurz vorgestellt, jedoch nicht weiter verwendet. Jedoch würde sich diese Methode eventuell dazu eignen, weitere Bauteilklassen, neben zylindrische und tetromino-förmige Bauteile, zu identifizieren. Des Weiteren wäre es auch möglich, dass Pentominos, Hexominos und auch weitere Polyominos [34] [48] für die Bauteilklassifizierung herangezogen werden. Auch wäre es denkbar, andere Grundgeometrien für die Art der Klassifizierung zu verwenden, die platonische Parkettierung liefert uns hierfür die passenden Teilgeometrien.

## 6 Akronymverzeichnis

<b>Adam</b>	Adaptive Momentum Estimation
<b>BGD</b>	Batch-Gradient Descent
<b>CLI</b>	Command Line Interface
<b>CNN</b>	Convolutional Neural Network
<b>CPU</b>	Central Processing Unit
<b>DQL</b>	Deep-Q Learning
<b>DQN</b>	Deep-Q Network
<b>FNN</b>	Feedforward Neural Network
<b>GPU</b>	Graphics Processing Unit
<b>HTTP</b>	Hypertext Transfer Protocol
<b>HTTPS</b>	Hypertext Transfer Protocol Secure
<b>ID</b>	Identifikator
<b>IEC</b>	International Electrotechnical Commission
<b>KI</b>	Künstliche Intelligenz
<b>MFC</b>	Mass-Flow Controller
<b>MGD</b>	Minibatch-Gradient Descent
<b>ML</b>	Maschine Learning
<b>MSE</b>	Mean Square Error
<b>OPC UA</b>	Open-Platform-Communication Unified-Architecture
<b>POE</b>	Programm-Organisationseinheit
<b>RL</b>	Reinforcement Learning
<b>RNN</b>	Recurrent Neural Network
<b>SGD</b>	Stochastic-Gradient Descent
<b>SL</b>	Supervised Learning
<b>SPS</b>	Speicherprogrammierbare Steuerung
<b>TCP</b>	Transmission Control Protocol
<b>TDL</b>	Temporal-Difference Learning
<b>UI</b>	User Interface
<b>UL</b>	Unsupervised Learning
<b>UML</b>	Unified Markup Language
<b>URI</b>	Uniform Resource Identifier
<b>URL</b>	Uniform Resource Locator
<b>VM</b>	Virtuelle Maschine

## 7 Abbildungsverzeichnis

1.1	V-Modell [1] . . . . .	3
2.1	Automatisierungsspyramide [2] . . . . .	5
2.2	Projektstruktur nach IEC 61131-3 . . . . .	6
2.3	POE-Struktur . . . . .	7
2.4	Definition der Datentypen . . . . .	8
2.5	Programmiersprachen der IEC 61131-3 [3] . . . . .	9
2.6	Browsing im Adressraum . . . . .	12
2.7	Browsing im Adressraum mit einer <i>view</i> . . . . .	12
2.8	<i>Node</i> -Modell von OPC UA [5] . . . . .	13
2.9	OPC UA-Protokolle und mögliche Kommunikationswege [4] . . . . .	14
2.10	Darstellung von Klassen in UML . . . . .	16
2.11	Darstellung von Klassen-Attributen in UML . . . . .	17
2.12	Darstellung von Assoziationen in UML . . . . .	18
2.13	Darstellung einer Zwischenklasse in UML . . . . .	18
2.14	Darstellung einer Komposition in UML . . . . .	18
2.15	Darstellung einer Aggregation in UML . . . . .	19
2.16	Darstellung einer Navigation in UML . . . . .	19
2.17	Darstellung von Assoziationsnamen in UML . . . . .	20
2.18	Darstellung eines abgeleiteten Attributes in UML . . . . .	20
2.19	Darstellung einer Vererbung in UML . . . . .	21
2.20	Beispiel eines Aktivitätsdiagramms in UML . . . . .	23
2.21	Platonische Parkettierung [10] . . . . .	24
2.22	Schnitt durch eine hexagonale Kugelpackung . . . . .	24
2.23	Teilbereiche des <i>machine learnings</i> . . . . .	25
2.24	Schematischer Aufbau eines Neurons . . . . .	26
2.25	<i>Identity-Activation</i> . . . . .	27
2.26	<i>Sigmoide-Activation</i> . . . . .	27
2.27	<i>TanH-Activation</i> . . . . .	27
2.28	<i>ReLU-Activation</i> . . . . .	27
2.29	<i>LeakyReLU-Activation</i> . . . . .	28
2.30	<i>Swish-Activation</i> . . . . .	28
2.31	FNN . . . . .	38
2.32	Einfacher Laplace-Filter-Kern . . . . .	40
2.33	Schichten eines CNNs . . . . .	43

2.34 Recurrent network . . . . .	44
2.35 Wahl einer Aktion mittels <i>policy</i> . . . . .	46
2.36 Backup diagram . . . . .	47
2.37 Beispiel einer <i>Markov chain</i> . . . . .	49
2.38 Beispiel <i>Markov reward</i> . . . . .	50
2.39 Beispiel <i>Markov process</i> . . . . .	51
2.40 Backup diagram für <i>state</i> und <i>action values</i> . . . . .	53
2.41 Algorithmen für RL [30] . . . . .	56
2.42 Dueling-Double-DQN . . . . .	59
3.1 Modul „simulation_members“ . . . . .	64
3.2 Modul „simulation_pages“ . . . . .	65
3.3 Modul „policies“ . . . . .	66
3.4 UML-Diagramm der Gesamtapplikation . . . . .	67
3.5 Codegenerierung UML . . . . .	68
3.6 Verzeichnisstruktur . . . . .	68
3.7 Hexagon-Parkettierung mit Geogebra . . . . .	69
3.8 Ergebnis des Programmcodes für die Parkettierung . . . . .	73
3.9 Hexagone 1. Instanz . . . . .	73
3.10 Hexagone 2. Instanz . . . . .	73
3.11 Diskretisierung von Bauteilen und Chargierplatte . . . . .	74
3.12 Flächenschwerpunkt . . . . .	75
3.13 Deadzone-Regel . . . . .	75
3.14 Porosität der Ablage von Tetrominos . . . . .	76
3.15 Aktionsschema: Ablegepositionen . . . . .	77
3.16 Aktionsschema: Einzelaktionen . . . . .	78
3.17 Einzelaktion: Bauteil hinlegen . . . . .	79
3.18 Zustandsdarstellung mit binären Werten . . . . .	79
3.19 Zustandsdarstellung mit Gleitkommazahlen . . . . .	80
3.20 Messrichtungen für die Zustandsdarstellung „reduced“ . . . . .	81
3.21 Schätzer für <i>state value</i> . . . . .	82
3.22 Schätzer für <i>q-value</i> . . . . .	82
3.23 Plot Transitions-Wahrscheinlichkeitsvektor . . . . .	83
3.24 Workflow der .csv-Generierung . . . . .	84
3.25 Filterung von Positionen . . . . .	85
3.26 CAD-Modell . . . . .	86
3.27 Realmodell . . . . .	86
3.28 Kommunikationsschema . . . . .	87

## 7 Abbildungsverzeichnis

---

4.1	<i>Home page</i>	90
4.2	<i>Draw page</i>	90
4.3	<i>Settings page</i>	90
4.4	<i>Simulation page</i>	90
4.5	Layout 1	91
4.6	Layout 2	91
4.7	Lineare Bewegung	93
4.8	Rotation	93
4.9	Rotation mittels komplexer Zahlen	95
4.10	Wegvektor	97
4.11	Koordinatenangabe	97
4.12	Layoutkonfiguration der Trainingszelle	98
4.13	Simulationslayout	99
4.14	<i>Policies</i>	99
4.15	Haupttraining: Placement-Agent	109
4.16	Vorabtraining: Placement-Agent SL	110
4.17	Haupttraining: Placement-Agent SL	111
4.18	Haupttraining: Controller-Agent	113
4.19	Einbindung der <i>agents</i>	116
12.1	Zustände der Klasse ‚Plant‘	132
12.2	Zustandsdiagramm der Klasse ‚Robot‘	133
12.3	Zustände der Klasse ‚Robot‘	133
12.4	Zustände der Klasse ‚Conveyor‘	133
12.5	Zustandsdiagramm der Klasse ‚Rotator‘	133
12.6	Zustände der Klasse ‚Rotator‘	134
12.7	Zustände der Klasse ‚Table‘	134
12.8	Zustandsdiagramm der Klasse ‚Table‘	134
12.9	Strategie für die Ablage	143

## 8 Tabellenverzeichnis

1.1	Ziele und Nicht-Ziele . . . . .	2
2.1	Abkürzung der <i>namespace</i> -URI mittels <i>lookup table</i> . . . . .	11
2.2	Möglichkeiten an Multiplizitäten . . . . .	20
2.3	Initialisierungen für Aktivierungsfunktionen . . . . .	31
2.4	Dimensionen der Matrizen aus diesem Kapitel . . . . .	36
2.5	Alle Möglichen (s,a,s',r)-Kombinationen aus Abb. 2.39 . . . . .	54
2.6	Lösen von MDP mit $\gamma = 0$ . . . . .	55
3.1	Bauteilklassen und deren ID . . . . .	63
3.2	Datenbaustein: DB_Interface_MFC . . . . .	87
4.1	Parametersätze: Placement-Agent . . . . .	108
4.2	Parametersätze: Placement-Agent SL . . . . .	110
4.3	<i>Rewards</i> der Einzelaktionen . . . . .	112
4.4	Parametersätze: Controller-Agent . . . . .	112
12.1	Setup der UI-Entwicklungsumgebung . . . . .	131
12.2	Setup der KI-Entwicklungsumgebung . . . . .	132
12.3	Struct: MFC_to_Robot . . . . .	135
12.4	Struct: Robot_to_MFC . . . . .	135
12.5	Struct: Job_cmd . . . . .	136
12.6	Struct: Job_state . . . . .	136
12.7	Parametereinstellungen A . . . . .	137
12.8	Parametereinstellungen B . . . . .	138
12.9	Parametereinstellungen C . . . . .	140
12.10	Parametereinstellungen D . . . . .	141

# 9 Formelverzeichnis

2.1:	Matrix-Notation für die gewichtete Summe . . . . .	26
2.2:	Aktivierungsfunktion <i>Identity</i> . . . . .	27
2.3:	Aktivierungsfunktion: Sigmoide . . . . .	27
2.4:	Aktivierungsfunktion: Tanh . . . . .	27
2.5:	Aktivierungsfunktion: ReLU . . . . .	27
2.6:	Aktivierungsfunktion: Leaky ReLU . . . . .	28
2.7:	Aktivierungsfunktion: Swish . . . . .	28
2.8:	Gradienten-Methode . . . . .	28
2.9:	Gradienten . . . . .	28
2.10:	Momentum-Methode I . . . . .	30
2.11:	Momentum-Methode II . . . . .	30
2.12:	RMSProp-Methode I . . . . .	30
2.13:	RMSProp-Methode II . . . . .	30
2.14:	Adam-Methode I . . . . .	31
2.15:	Adam-Methode II . . . . .	31
2.16:	Adam-Methode III . . . . .	31
2.17:	Abbildung von $J(\mathbf{w})$ . . . . .	32
2.18:	MSE-Kostenfunktional . . . . .	32
2.19:	<i>Cross entropy</i> -Kostenfunktional . . . . .	33
2.20:	<i>Huber loss</i> -Kostenfunktional . . . . .	33
2.21:	$l_1$ -regularization . . . . .	34
2.22:	$l_2$ -regularisation . . . . .	34
2.23:	Inputdaten in Matrix-Notation . . . . .	34
2.24:	Schreibweise von $\mathbf{z}$ durch (2.23) . . . . .	34
2.25:	Schreibweise von $f(\mathbf{z})$ durch (2.24) . . . . .	35
2.26:	Gewichtungsfaktoren in Matrix-Notation . . . . .	35
2.27:	<i>Bias</i> in Matrix-Notation . . . . .	35
2.28:	Gewichtete Summen in Matrix-Notation I . . . . .	35
2.29:	Gewichtete Summen in Matrix-Notation II . . . . .	35
2.30:	Übertragungsverhalten von Multilayer-Neuronal-Networks . . . . .	36
2.31:	Softmax-Funktion . . . . .	38
2.32:	Summe der Softmax-Funktionen . . . . .	38
2.33:	Anzahl an lernbaren Parametern pro Schicht . . . . .	39
2.34:	Summe aller lernbaren Parameter eines Netzwerks . . . . .	39
2.35:	Faltungsfilter . . . . .	40

2.35:	Anwendung <i>pooling</i> I . . . . .	42
2.37:	Anwendung <i>pooling</i> II . . . . .	42
2.38:	Dimensionsbestimmung <i>pooling</i> . . . . .	42
2.39:	Zustandsvariable von RNN's . . . . .	44
2.40:	Definition <i>policy</i> . . . . .	46
2.41:	<i>Stochastic policy</i> . . . . .	46
2.42:	<i>Deterministic policy</i> . . . . .	46
2.43:	Wahrscheinlichkeit für Eintritt von Zustand und Belohnung . . . . .	47
2.44:	<i>Reward discounting</i> . . . . .	47
2.45:	$\epsilon$ - <i>greedy policy</i> . . . . .	48
2.46:	Transitionswahrscheinlichkeit einer <i>Markov chain</i> . . . . .	49
2.47:	Transition-Wahrscheinlichkeit-Matrix einer <i>Markov chain</i> . . . . .	49
2.48:	<i>Value</i> eines Zustandes . . . . .	50
2.49:	<i>Value</i> eines Zustandes abhängig von der <i>policy</i> . . . . .	51
2.50:	<i>Action value</i> eines Zustandes abhängig von der <i>policy</i> . . . . .	51
2.51:	Bellman-Gleichung für $v_\pi(s)$ . . . . .	52
2.52:	Verknüpfung von $v_\pi(s)$ mit $q_\pi(s, a)$ . . . . .	53
2.53:	Bellman-Gleichung für $q_\pi(s, a)$ . . . . .	53
2.54:	Update-Gleichung für TDL . . . . .	56
2.55:	<i>Temporal difference error</i> . . . . .	56
2.56:	<i>Q learning</i> . . . . .	57
2.57:	Update der Gewichtungsfaktoren für DQL . . . . .	57
2.58:	Definition <i>advantage</i> . . . . .	58
2.59:	<i>Dueling-Double-DQN</i> . . . . .	59
3.1:	Flächenschwerpunkt Tetromino . . . . .	75
3.2:	Belohnungskonzept . . . . .	76
3.3:	Aktionstupel für den <i>agent</i> nach Schema 1 . . . . .	77
3.4:	Vektor der Transitionswahrscheinlichkeiten . . . . .	79
4.1:	Differenzialgleichung des Weges . . . . .	92
4.2:	Differenzialgleichung des Winkels . . . . .	93
4.3:	Komplexe Zahl . . . . .	95
4.4:	Multiplikation komplexer Zahlen . . . . .	95
12.1:	Definition des Erwartungswerts . . . . .	131
12.2:	Linearitätseigenschaft des Erwartungswerts . . . . .	131

# 10 Programmverzeichnis

2.1	OPC UA Node-ID . . . . .	11
2.2	Adam Klasse Pytorch . . . . .	31
2.3	Gewichts-Initialisierung Pytorch . . . . .	32
2.4	Kostenfunktionale Pytorch . . . . .	33
2.5	<i>Dropout</i> Pytorch . . . . .	34
2.6	<i>Search space</i> Raytune . . . . .	37
2.7	FNN Pytorch . . . . .	39
2.8	<i>Convolution layer</i> Pytorch . . . . .	41
2.9	<i>Pooling layer</i> Pytorch . . . . .	42
2.10	CNN Pytorch . . . . .	43
3.1	Bibliotheken . . . . .	69
3.2	Hexagonfunktion . . . . .	70
3.3	Hilfsfunktion . . . . .	71
3.4	Testroutine . . . . .	72
3.5	Parkettierungsfilter . . . . .	85
3.6	Standard-NodeID Siemens S7 1500 . . . . .	88
4.1	Lineare Bewegung . . . . .	92
4.2	Rotatorische Bewegung . . . . .	93
4.3	<i>Subscriptions</i> . . . . .	94
4.4	Komplexe Rotation . . . . .	95
4.5	Wegminimierung . . . . .	96
4.6	Trainingroutine: <i>Libraries</i> . . . . .	100
4.7	Trainingroutine: <i>Arguments</i> . . . . .	100
4.8	Trainingroutine: <i>Training</i> . . . . .	102
4.9	Trainingroutine: <i>Main</i> . . . . .	104
4.10	Networks: <i>Libraries</i> . . . . .	104
4.11	Networks: DQN- and DDQN-Class . . . . .	105
4.12	Networks: DDDQN-Class . . . . .	105

# 11 Literaturverzeichnis

- [1] Vajna, Sándor, Weber, Christian, Zeman, Klaus, Hohenberger, Peter, Gerhard, Detlef, und Wartzack, Sandro. *CAx für Ingenieure: Eine praxisbezogene Einführung (German Edition)*. Springer Vieweg, 2018.
- [2] Heinrich, Berthold, Linke, Petra, und Glöckler, Michael. *Grundlagen Automatisierung*. Springer-Verlag GmbH, 2020.
- [3] John. *IEC 61131-3: Programming industrial automation systems : concepts and programming languages, requirements for programming systems, decision-making aids*. Berlin New York: Springer, 2010.
- [4] Plenk, Valentin. *Angewandte Netzwerktechnik kompakt Dateiformate, Übertragungsprotokolle und ihre Nutzung in Java-Applikationen*. Wiesbaden: Springer Vieweg, 2017.
- [5] Wikipedia der OPCFoundation-Group [<http://wiki.opcfoundation.org>]. Aufgerufen am 01.04.2021.
- [6] Fabian Spitzer, MSc. *Framework für die Konfiguration von Intralogistiksystemen zur automatisierten Generierung der Kommunikationslogik*. 2019. Masterarbeit. FH-Wels.
- [7] Damm, Gappmeier, Zugfil, Plöb, Fiat, und Störkuhl. *Sicherheitsanalyse OPC UA*. Verfügbar von: <https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/OPCUA/OPCUA.pdf>. Technischer Bericht. Bundesamt für Sicherheit in der Informationstechnik. Aufgerufen am 24.06.2021.
- [8] Randen, Hendrik Jan van, Bercker, Christian, und Fiendl, Julian. *Einführung in UML*. Gabler, Betriebswirt.-Vlg, 2016.
- [9] Alsina, Claudi. *Perlen der Mathematik : 20 geometrische Figuren als Ausgangspunkte für mathematische Erkundungsreisen*. Berlin: Springer Spektrum, 2015.
- [10] Strick, Heinz Klaus. *Mathematik ist wunderschön*. Springer-Verlag GmbH, 2021.
- [11] Wikipedia. *Keplersche Vermutung* [[https://de.wikipedia.org/wiki/Keplersche\\_Vermutung](https://de.wikipedia.org/wiki/Keplersche_Vermutung)].
- [12] *The Kepler Conjecture*. Springer-Verlag GmbH, 2011.

## 11 Literaturverzeichnis

---

- [13] Michelucci, Umberto. *Applied Deep Learning*. APRESS L.P., 2018.
- [14] Sanghi, Nimish. *Deep Reinforcement Learning with Python*. Apress, 2021.
- [15] Géron, Aurélien. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly UK Ltd., 2019.
- [16] PyTorch. *PyTorch Documentation* [<https://pytorch.org/docs/stable/index.html>]. Aufgerufen am 24.06.2021.
- [17] *Aktivierungsfunktionen nn-Modul* [<https://pytorch.org/docs/stable/nn.html>]. Aufgerufen am 29.07.2021.
- [18] Ruder, Sebastian. An overview of gradient descent optimization algorithms. 2016 (). Verfügbar von arXiv: 1609.04747 [cs.LG].
- [19] Glorot, Xavier. Understanding the difficulty of training deep forward neural networks. In: *AISTATS*. 2010.
- [20] Kumar, Siddharth Krishna. On weight initialization in deep neural networks. 2017 (). Verfügbar von arXiv: 1704.08863 [cs.LG].
- [21] Boulila, Wadii, Driss, Maha, Al-Sarem, Mohamed, Saeed, Faisal, und Krichen, Moez. Weight Initialization Techniques for Deep Learning Algorithms in Remote Sensing: Recent Trends and Future Perspectives. 2021 (). Verfügbar von arXiv: 2102.07004 [cs.LG].
- [22] DiPietro, Rob. *A Friendly Introduction to Cross-Entropy Loss* [<https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/>]. 2016. Aufgerufen am 01.04.2021.
- [23] Cavazza, Jacopo, und Murino, Vittorio. Active Regression with Adaptive Huber Loss. 2016 (). Verfügbar von arXiv: 1606.01568 [cs.LG].
- [24] *Hyperparameter tuning tutorial* [[https://pytorch.org/tutorials/beginner/hyperparameter\\_tuning\\_tutorial.html](https://pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html)]. Aufgerufen am 19.07.2021.
- [25] Gao, Tianxiang, und Jojic, Vladimir. Degrees of Freedom in Deep Neural Networks. 2016 (). Verfügbar von arXiv: 1603.09260 [cs.LG].
- [26] Pattanayak, Santanu. *Pro Deep Learning with TensorFlow*. Apress, 2017.
- [27] *Torchvision models* [<https://pytorch.org/vision/stable/models.html>]. Aufgerufen am 15.08.2021.
- [28] rpatrik96. *Advantage Actor Critic* [<https://github.com/rpatrik96/pytorch-a2c/blob/master/src/model.py>]. 2019. Aufgerufen am 19.07.2021.

## 11 Literaturverzeichnis

---

- [29] Ohrn, Anders. *Image Clustering Implementation with PyTorch* [<https://towardsdatascience.com/image-clustering-implementation-with-pytorch-587af1d14123>]. Aufgerufen am 31.07.2021.
- [30] OpenAI. *Kind of RL Algorithms* [[https://spinningup.openai.com/en/latest/spinningup/rl\\_intro2.html](https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html)]. Aufgerufen am 24.06.2021.
- [31] Schaul, Tom, Quan, John, Antonoglou, Ioannis, und Silver, David. Prioritized Experience Replay. 2015 (). Verfügbar von arXiv: 1511.05952 [cs.LG].
- [32] Hasselt, Hado van, Guez, Arthur, und Silver, David. Deep Reinforcement Learning with Double Q-learning. 2015 (). Verfügbar von arXiv: 1509.06461 [cs.LG].
- [33] Wang, Ziyu, Schaul, Tom, Hessel, Matteo, Hasselt, Hado van, Lanctot, Marc, und Freitas, Nando de. Dueling Network Architectures for Deep Reinforcement Learning. 2015 (). Verfügbar von arXiv: 1511.06581 [cs.LG].
- [34] Golomb, Solomon. *Polyominoes : puzzles, patterns, problems, and packings*. Princeton, N.J: Princeton University Press, 1994.
- [35] *When a claim was made that each Tetris block has a name* [<http://www.8-bitcentral.com/blog/2019/tetrisBlockNames.html>]. Aufgerufen am 14.08.2021.
- [36] Siemens. *SIMATIC Robot Integration für KUKA* [<https://support.industry.siemens.com/cs/document/109482123>]. Aufgerufen am 23.06.2021.
- [37] Siemens. *OPC UA methods for the SIMATIC S7-1500* [<https://support.industry.siemens.com/cs/document/109756885>]. Aufgerufen am 23.06.2021.
- [38] Siemens. *Funktionshandbuch SIMATIC S7 1500 Kommunikation*. 2019.
- [39] Library, Free OPC-UA. *GitHub repository of asyncua* [<https://github.com/FreeOpcUa/opcua-asyncio>]. Aufgerufen am 03.08.2021.
- [40] Brandstaetter, Christian. *Videos of this work* [[https://www.youtube.com/channel/UChI56p6C45m\\_HcJ\\_eteWmgA/playlists?view=1&sort=dd&shelf\\_id=0](https://www.youtube.com/channel/UChI56p6C45m_HcJ_eteWmgA/playlists?view=1&sort=dd&shelf_id=0)]. Aufgerufen am 02.08.2021.
- [41] Brandstaetter, Christian. *GitHub repository of this work* [<https://github.com/codocalypse/ruetrис/>]. Aufgerufen am 30.07.2021.
- [42] *Reinforcement learning (DQN) tutorial* [[https://pytorch.org/tutorials/intermediate/reinforcement\\_q\\_learning.html](https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html)]. Aufgerufen am 23.06.2021.

## 11 Literaturverzeichnis

---

- [43] *Train a mario-playing RL agent (DDQN)* [[https://pytorch.org/tutorials/intermediate/mario\\_rl\\_tutorial.html](https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html)]. Aufgerufen am 23.06.2021.
- [44] Uvipen. *Deep Q-learning for playing Tetris* [<https://github.com/uvipen/Tetris-deep-Q-learning-pytorch>]. Aufgerufen am 23.06.2021.
- [45] MathWorks. *Reinforcement Learning Toolbox User's Guide* [[https://www.jiule.cn/wp-content/uploads/2020/05/reinforcement\\_user\\_guide.pdf](https://www.jiule.cn/wp-content/uploads/2020/05/reinforcement_user_guide.pdf)]. Aufgerufen am 30.07.2021.
- [46] Brandstaetter, Christian. *GitHub repository of results* [[https://github.com/codocalypse/ruetrис\\_results](https://github.com/codocalypse/ruetrис_results)]. Aufgerufen am 06.08.2021.
- [47] Irpan, Alex. *Deep Reinforcement Learning Doesn't Work Yet* [<https://www.alexirpan.com/2018/02/14/rl-hard.html>]. 2018.
- [48] Wikipedia. *Polyomino* [<https://de.wikipedia.org/wiki/Polyomino>].

# 12 Anhang

## 12.1 Grundlagen aus der Statistik

Definition Erwartungswert einer diskreten Zufallsvariablen  $X$ .

$$\mathbb{E}[X] = \sum_{k \in \mathbb{D}_x} k \mathbb{P}(X = k) \quad (12.1)$$

Linearitätseigenschaft des Erwartungswerts.

$$\mathbb{E}[\alpha X + \beta Y] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y] \quad (12.2)$$

## 12.2 Verwendete Hard- und Software

Die Konfiguration der Entwicklungsumgebungen ist den folgenden Tabellen zu entnehmen. Die Konfiguration der KI-Entwicklungsumgebung befindet sich als „env.yml“-File auf Github [41].

**Tabelle 12.1:** Setup der UI-Entwicklungsumgebung

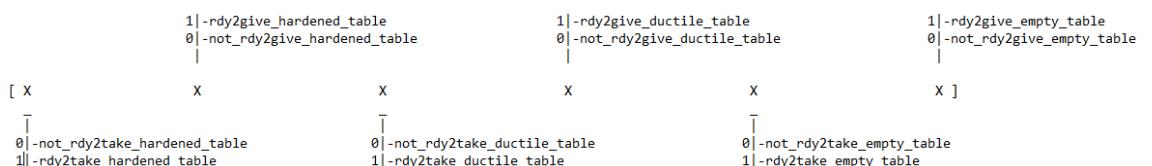
Framework/ Bibliothek/ OS	Version
Scipy	1.6.1
Tkinter	-
Numpy	1.19.3
Windows 10	10.0.18363
Spyder	4.2.3
Python	3.7.9
Async-OPCUA	0.9.90

**Tabelle 12.2:** Setup der KI-Entwicklungsumgebung

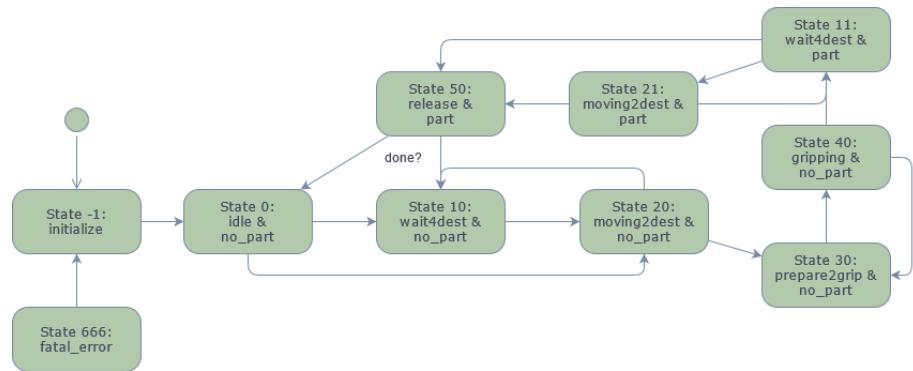
Framework/ Bibliothek/ OS/ IDE/ Hardware	Version
PyTorch	1.9.0
Conda	4.9.2
Python	3.6.12
TensorboardX	2.4.0
Linux-Manjaro Gnome	21.1.0
Linux Kernel	5.10.53-1
Numpy	1.19.5
OpenCV	4.1.0
Spyder	4.2.1
Motherboard	ASUS Prime
CPU (overclocked)	Intel Core i7-7800X 4.27GHz
GPU (overclocked)	Geforce RTX 2080 2.1GHz
GPU (first died)	Geforce RTX 3070 TI
RAM (overclocked)	Corsair 32GB 3.2GHz

## 12.3 Zustände der Simulation-Member-Klassen

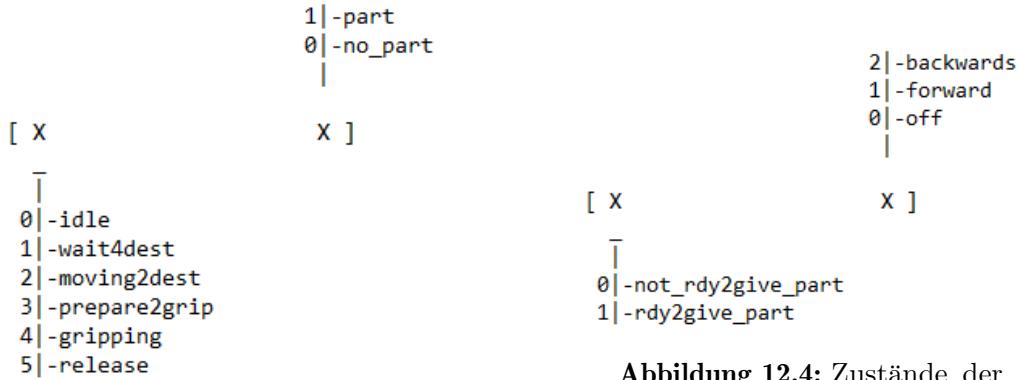
Im folgenden sind die Zustände, binär oder dezimal Codiert, angegeben, welche die jeweiligen Klassen-Objekte aus dem Modul „Simulation\_members“ annehmen können.



**Abbildung 12.1:** Zustände der Klasse „Plant“

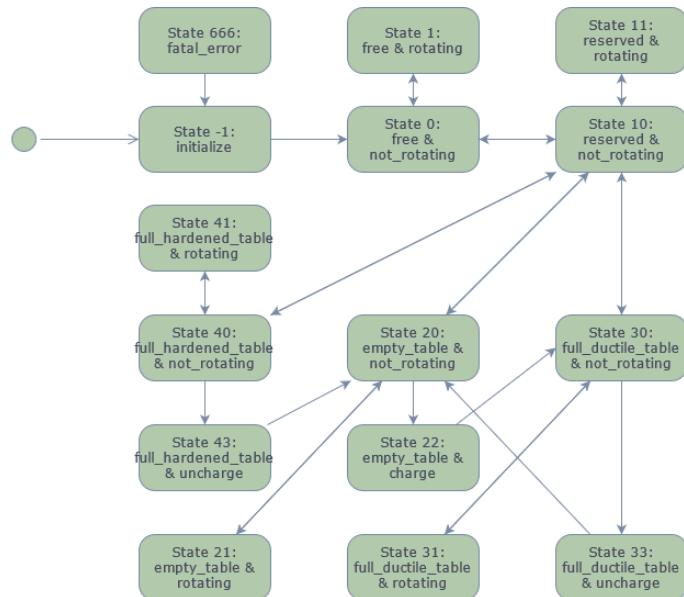


**Abbildung 12.2:** Zustandsdiagramm der Klasse ‚Robot‘



**Abbildung 12.3:** Zustände der Klasse „Robot“

**Abbildung 12.4:** Zustnde der Klasse ‘Conveyor‘



**Abbildung 12.5:** Zustandsdiagramm der Klasse „Rotator“

## 12 Anhang

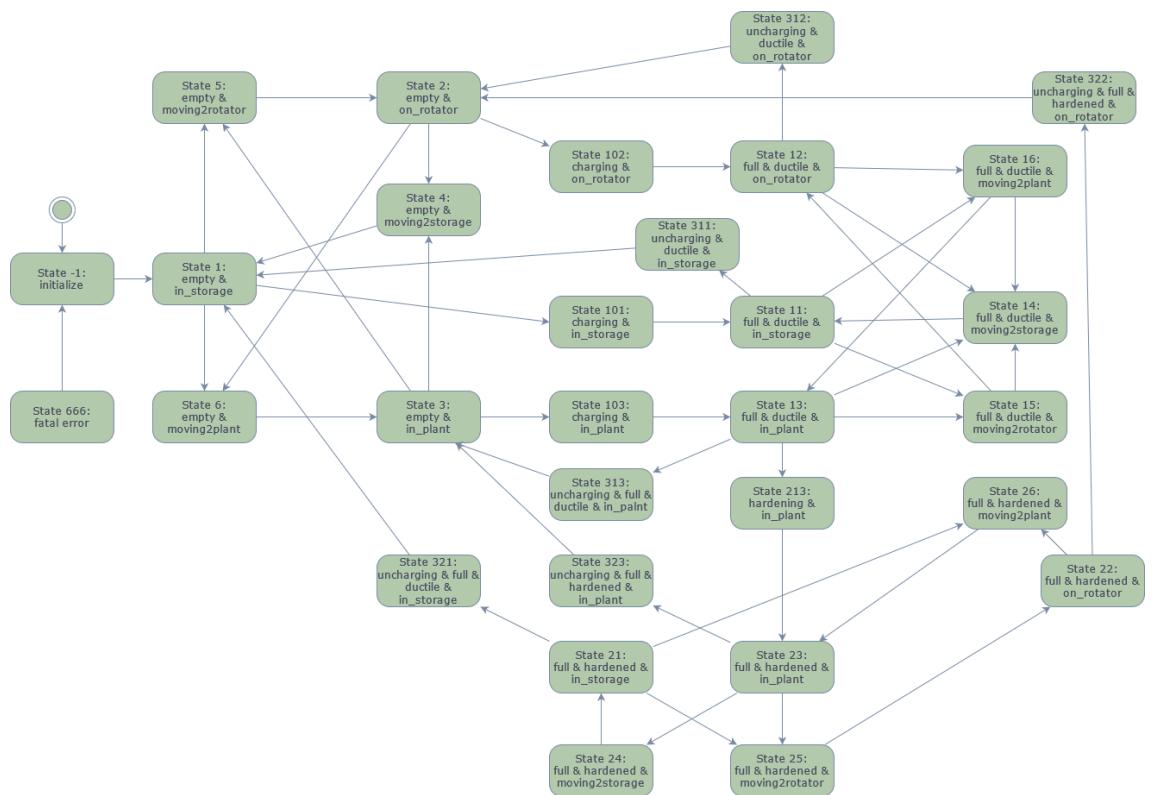
---

		3  -uncharge 2  -charge 1  -rotating 0  -not_rotating
[ X ]	X ]	
-		
0  -free 1  -reserved 2  -empty_table 3  -ductile_table 4  -hardened_table		

**Abbildung 12.6:** Zustände der Klasse „Rotator“

		2  -full_hardened 1  -full_ductile 0  -empty
[ X ]	X ]	
-		
0  -not_processing 1  -charging 2  -hardening 3  -uncharging		
[ X ]	X ]	
-		
1  -in_storage 2  -on_rotator 3  -in_plant 4  -moving2storage 5  -moving2rotator 6  -moving2plant		

**Abbildung 12.7:** Zustände der Klasse „Table“



**Abbildung 12.8:** Zustandsdiagramm der Klasse „Table“

## 12.4 OPC UA Datenmodell

Den folgenden Tabellen, ist der Aufbau der Kommunikation zur SPS der Trainingszelle zu entnehmen. Die Bezeichnung MFC entspricht dem erstellten Simulationsprogramm.

**Tabelle 12.3:** Struct: MFC\_to\_Robot

Node	Datentyp	Beschreibung
Job_cmd	Struct	Befehlsstruktur für Pick&Place-Anwendung
LifeBit	Boolean	Rechtecksignal als Lebenszeichen (min. 0.2 Hz)
NewDataConfirmed	Boolean	Bestätigungsbit bei Erhalt neuer Daten
NewDataValid	Boolean	Informationsbit über Sendung neuer Daten

**Tabelle 12.4:** Struct: Robot\_to\_MFC

Node	Datentyp	Beschreibung
Job_state	Struct	Informationsstruktur der Jobausführung
LifeBit	Boolean	Rechtecksignal als Lebenszeichen mit 0.5 Hz
NewDataConfirmed	Boolean	Bestätigungsbit bei Erhalt neuer Daten
NewDataValid	Boolean	Informationsbit über Sendung neuer Daten
Ready	Boolean	Bereit für eine Jobausführung

**Tabelle 12.5:** Struct: Job\_cmd

Node	Datentyp	Beschreibung
Target_Position	Struct	Ziel- bzw. Ablageposition
Source_Position	Struct	Start- bzw. Abholposition
Target_ID	Integer	Bezugssystem (ID) der Ablageposition
Source_ID	Integer	Bezugssystem (ID) der Abholposition
Job_Type	Integer	Aufgabendefinition: 1)Pick, 2)Place, 3)Pick&Place
Execute	Boolean	Befehlsbit zur Ausführung

**Tabelle 12.6:** Struct: Job\_state

Node	Datentyp	Beschreibung
Busy	Boolean	Logisch 1, wenn beschäftigt
Done	Boolean	Logisch 1, wenn Job abgearbeitet
Error	Boolean	Logisch 1, wenn Fehler auftraten
Gripper_Occupied	Boolean	Logisch 1, wenn Bauteil im Greifer
Job_Plausible	Boolean	Logisch 1, wenn übertragene Daten sinnvoll
Pick_Busy	Boolean	Logisch 1 bei Greifroutine
Pick_Done	Boolean	Logisch 1, wenn Greifroutine beendet
Pick_Error	Boolean	Logisch 1, bei Fehlern während der Greifroutine
Place_Busy	Boolean	Logisch 1, bei Ablegeroutine
Place_Done	Boolean	Logisch 1, wenn Ablegeroutine beendet
Place_Error	Boolean	Logisch 1, bei Fehlern während der Ablageroutine

## 12.5 Parameterlisten der Versuchsdurchführung

Es folgen die Parametereinstellungen der Versuchsdurchführung. Weitere Details sind dem Github-*repository* [46] zu entnehmen.

**Tabelle 12.7:** Parametereinstellungen A

Bezeichung-ID	ID001	ID002	ID003	ID004
-table_dia	420	420	420	420
-block_size	20	20	20	20
-state_report	'full_float'	'full_float'	'full_float'	'full_float'
-action_method	-	'placement'	'placement'	'placement'
-rnd_pos	-	False	False	False
-rnd_rot	-	False	False	False
-reward_action_dict	True	False	False	False
-c1	2	2	2	2
-c2	50	50	50	50
-c3	3	3	3	3
-c4	5	5	5	5
-net_type	-	'd-q-net'	'd-q-net'	d-q-net
-statescale_l1	-	10	10	20
-statescale_l2	-	6	6	12
-batch_size	-	128	128	128
-lr $\alpha$	-	0.01	0.01	0.01
-lr_decay	-	0.9999	1.0	0.9999
-final_lr_frac	-	0.001	-	0.001
-gamma $\gamma$	-	0.2	0.2	0.2
-initial_epsilon	-	0.01	1	0.01
-final_epsilon	-	0.001	0.1	0.001
-epsilon_decay	-	0.999999	0.99999	0.999999
-num_episodes	1.3e3	15e3	15e3	15e3
-memory_size	1e5	1e5	1e5	1e5

-burnin	-	1000	1000	1000
-learn_every	-	1	1	1
-sync_every	-	100	100	100
-save_interval	100	7e4	7e4	7e4
-render	True	False	False	False
-load_path	-	-	-	-
-continue_training	-	False	False	False
-new_epsilon	-	-	-	-
-create_supervised_data	True	False	False	False
-learn_from_benchmark	-	True	False	True
-supervised_training_steps	-	5e4	-	5e4
-print_state_2_cli	False	False	False	False

**Tabelle 12.8:** Parametereinstellungen B

Bezeichnung-ID	ID005	ID006	ID007	ID008
-table_dia	420	420	420	420
-block_size	20	20	20	20
-state_report	'full_float'	'full_float'	'full_float'	'full_float'
-action_method	'placement'	'placement'	'placement'	'placement'
-rnd_pos	False	False	False	False
-rnd_rot	False	False	False	False
-reward_action_dict	False	False	False	False
-c1	2	2	2	2
-c2	50	50	50	50
-c3	3	3	3	3
-c4	5	5	5	5

<code>-net_type</code>	'd-q-net'	'd-q-net'	'd-q-net'	d-q-net
<code>-statescale_l1</code>	5	5	5	10
<code>-statescale_l2</code>	3	3	3	6
<code>-batch_size</code>	128	128	128	128
<code>-lr <math>\alpha</math></code>	0.01	0.01	0.01	0.01
<code>-lr_decay</code>	0.9999	0.9999	0.9999	1.0
<code>-final_lr_frac</code>	0.001	0.01	0.01	-
<code>-gamma <math>\gamma</math></code>	0.2	0.2	0.4	0.2
<code>-initial_epsilon</code>	0.01	0.5	0.4	0.01
<code>-final_epsilon</code>	0.001	0.01	0.03	1.0
<code>-epsilon_decay</code>	0.999999	0.999995	0.99999	0.99999
<code>-num_episodes</code>	15e3	15e3	15e3	15e3
<code>-memory_size</code>	1e5	1e5	1e5	1e5
<code>-burnin</code>	1000	1000	1000	1000
<code>-learn_every</code>	1	1	1	1
<code>-sync_every</code>	100	100	100	100
<code>-save_interval</code>	7e4	7e4	7e4	7e4
<code>-render</code>	False	False	False	False
<code>-load_path</code>	-	-	-	-
<code>-continue_training</code>	False	False	False	False
<code>-new_epsilon</code>	-	-	-	-
<code>-create_supervised_data</code>	False	False	False	False
<code>-learn_from_benchmark</code>	True	True	True	False
<code>-supervised_training_steps</code>	5e4	1e4	1e4	-
<code>-print_state_2_cli</code>	False	False	False	False

**Tabelle 12.9:** Parametereinstellungen C

Bezeichung-ID	ID009	ID010	ID011	ID012
-table_dia	420	420	420	420
-block_size	20	20	20	20
-state_report	'reduced'	'full_bool'	'reduced'	'reduced'
-action_method	'placement'	'placement'	'controller'	'controller'
-rnd_pos	False	False	False	False
-rnd_rot	False	False	False	False
-reward_action_dict	False	False	False	False
-c1	2	2	2	2
-c2	50	50	50	50
-c3	3	3	3	3
-c4	5	5	5	5
-net_type	'd-q-net'	'd-q-net'	'dd-q-net'	ddd-q-net
-statescale_l1	60	5	60	60
-statescale_l2	30	3	40	40
-batch_size	128	128	128	128
-lr $\alpha$	0.01	0.01	1e-4	1e-4
-lr_decay	1.0	1.0	1.0	1.0
-final_lr_frac	-	-	-	-
-gamma $\gamma$	0.2	0.2	0.9	0.9
-initial_epsilon	1.0	1.0	1.0	1.0
-final_epsilon	0.1	0.1	0.01	0.01
-epsilon_decay	0.99999	0.99999	0.999999	0.999999
-num_episodes	15e3	15e3	4e5	1.1e5
-memory_size	1e5	1e5	1e5	1e5

-burnin	1000	1000	1e4	1e4
-learn_every	1	1	3	3
-sync_every	100	100	1e4	1e4
-save_interval	7e4	7e4	7e5	1e6
-render	False	False	False	False
-load_path	-	-	-	-
-continue_training	False	False	False	False
-new_epsilon	-	-	-	-
-create_supervised_data	False	False	False	False
-learn_from_benchmark	False	False	False	False
-supervised_training_steps	-	-	-	-
-print_state_2_cli	False	False	False	False

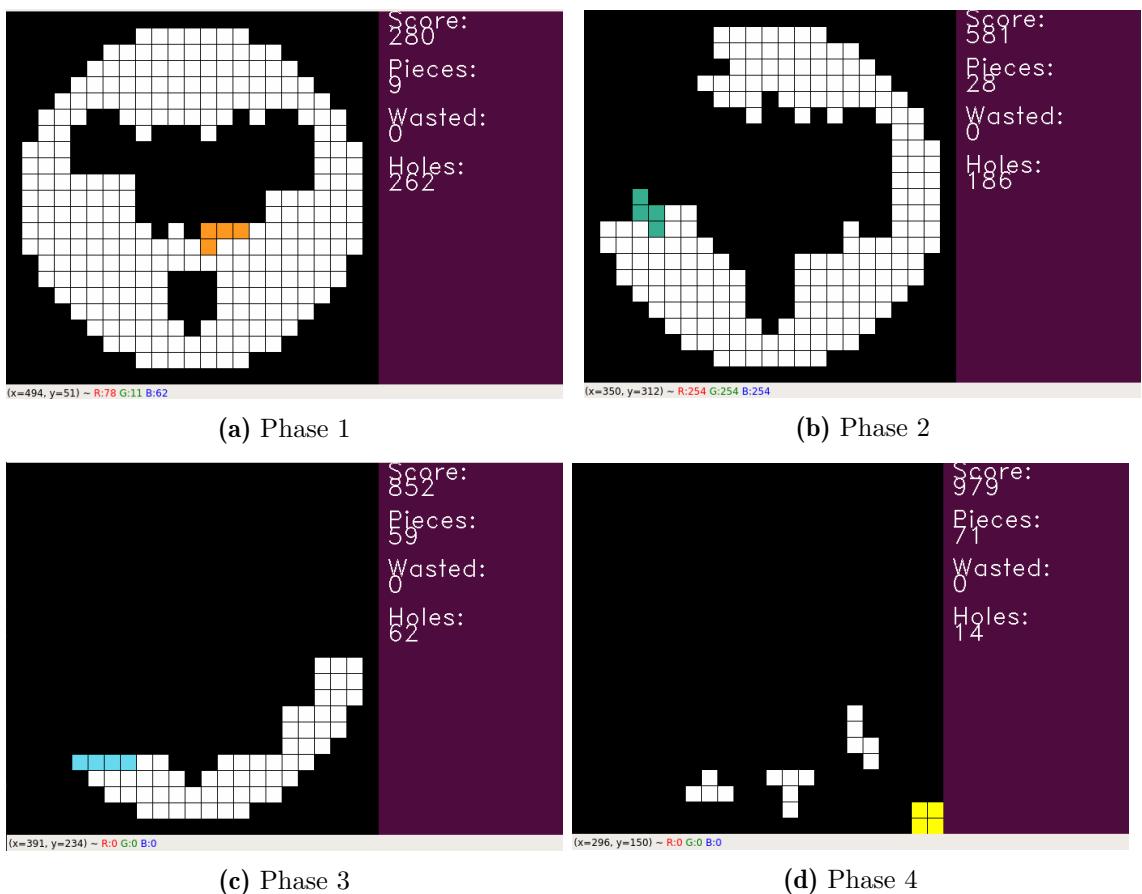
**Tabelle 12.10:** Parametereinstellungen D

Bezeichung-ID	ID013	ID014	ID015	ID016
-table_dia	420	420	420	420
-block_size	20	20	20	20
-state_report	'full_bool'	'full_bool'	'full_float'	'reduced'
-action_method	'controller'	'controller'	'controller'	'controller'
-rnd_pos	False	True	True	True
-rnd_rot	False	True	True	True
-reward_action_dict	False	False	False	False
-c1	2	2	2	3
-c2	50	50	50	50
-c3	3	3	3	3
-c4	5	5	5	5

<code>-net_type</code>	'ddd-q-net'	'ddd-q-net'	'ddd-q-net'	ddd-q-net
<code>-statescale_l1</code>	5	5	10	60
<code>-statescale_l2</code>	3	3	6	40
<code>-batch_size</code>	64	64	64	64
<code>-lr α</code>	1e-4	1e-4	1e-4	1e-4
<code>-lr_decay</code>	0.999999	0.999999	1.0	1.0
<code>-final_lr_frac</code>	0.1	0.1	-	-
<code>-gamma γ</code>	0.8	0.8	0.5	0.5
<code>-initial_epsilon</code>	1.0	1.0	1.0	1.0
<code>-final_epsilon</code>	0.1	0.08	0.08	0.08
<code>-epsilon_decay</code>	0.999999	0.999999	0.999999	0.999999
<code>-num_episodes</code>	1.1e5	1.3e5	1.3e5	1.3e5
<code>-memory_size</code>	1e5	1e5	1e5	1e5
<code>-burnin</code>	1e4	1e4	1e4	1e4
<code>-learn_every</code>	3	3	3	3
<code>-sync_every</code>	1e4	1e4	1e4	1e4
<code>-save_interval</code>	1e6	1e6	1e6	1e6
<code>-render</code>	False	False	False	False
<code>-load_path</code>	-	-	-	-
<code>-continue_training</code>	False	False	False	False
<code>-new_epsilon</code>	-	-	-	-
<code>-create_supervised_data</code>	False	False	False	False
<code>-learn_from_benchmark</code>	False	False	False	False
<code>-supervised_training_steps</code>	-	-	-	-
<code>-print_state_2_cli</code>	False	False	False	False

## 12.6 Strategie für die manuelle Interaktion

Da die manuelle Interaktion mit dem *environment* implementiert wurde, konnten auch eigene Strategien für die optimale Lösung erzeugt werden. Eine vielversprechende Strategie sei hier kurz vorgestellt (Abb. 12.9): Zu Beginn werden die Bauteile mittig abgelegt, um die zusätzlichen Punkte für den kürzeren Abstand zur Mitte, zu erhalten (a). Nach einigen Ablegungen wird begonnen alle Felder zwischen Mittelfeld und Rand an einer beliebigen Stelle vollständig auszufüllen (b). Anschließend versucht man, diesen Bereich weiter in den radialen Richtungen auszudehnen (c). Zu guter Letzt versucht man, unter Berücksichtigung der Transitions-Wahrscheinlichkeiten für die Ablage, geeignete Felder freizulassen, insofern sich diese nicht vermeiden lassen (d). Man wird feststellen, dass hier das Glück eine große Rolle spielt.



**Abbildung 12.9:** Strategie für die Ablage