

Technische Hochschule Ulm

Masterarbeit

zur Erlangung des akademischen Grades
Master of Engineering (M. Eng.)

Studiengang Systemengineering & Management Logistik

Simulationsbasiertes maschinelles Lernen für die agentenbasierte Steuerung von Materialflüssen – Konzeption und prototypische Umsetzung

Martin Schlump

Aufgabensteller: Prof. Dr. Sven Völker

Arbeit vorgelegt am: 06.08.2020

Immatrikulationsnummer: 3126594

Anschrift: Neumummen 18, 87509 Immenstadt

Kurzfassung

Der Trend zu hoch individualisierten Produkten und die damit einhergehende große Variantenvielfalt erfordern wandlungsfähige und flexible Logistik- und Produktionssysteme. Getrieben durch die digitale Transformation im Rahmen von Industrie 4.0 werden dezentrale, agentenbasierte Systemstrukturen und Steuerungsprinzipien im Fertigungs- und Logistikumfeld entwickelt, die selbstständig Transport- und Steuerungsentscheidungen in Materialflüssen treffen. Besonders die in den letzten Jahren stark weiterentwickelten Methoden des maschinellen Lernens sind treibende Kräfte für die Implementierung dezentraler Fertigungs- und Steuerungskonzepte. Diese müssen jedoch meist in digitalen Modellen abgebildet, entwickelt und getestet werden, bevor sie zum Einsatz kommen. Infolge dessen wurde die Zielstellung dieser Arbeit formuliert, ein Konzept zum simulationsbasierten maschinellem Lernen für die agentenbasierte Steuerung von Materialflüssen auszuarbeiten und anzuwenden. Anhand des Prozesses der Verteilung von Fertigungsaufträgen an Bearbeitungsressourcen wird ein Agent entwickelt, der durch bestärkendes Lernen eine Taktik zur Reihenfolgebildung der Aufträge erlernt. Dazu werden die Prozessabläufe in einer ereignisorientierten Simulation konzipiert und die Kommunikation zur Anwendung des maschinellen Lernens erarbeitet. Im nächsten Schritt erfolgt die Entwicklung eines Algorithmus zum bestärkenden Lernen eines neuronalen Netzes im Open Source Framework Keras, das die Steuerungsstrategie zur Reihenfolgebildung repräsentiert. Nachdem das implementierte Konzept verifiziert und validiert ist, wird der Agent in einem Simulationsmodell erprobt.

Inhalt

Kurzfassung.....	II
Abbildungsverzeichnis	VII
Tabellenverzeichnis	IX
Abkürzungsverzeichnis	X
1 Einleitung	11
1.1 Problemstellung	11
1.2 Zielstellung.....	12
1.3 Aufbau und Vorgehensweise.....	12
2 Produktionsplanung und -Steuerung	14
2.1 PPS-Systeme	14
2.2 Logistische Zielgrößen.....	15
2.3 Wirkzusammenhänge der Fertigungssteuerung.....	16
2.4 Zentrale und dezentrale Fertigungssteuerung.....	17
2.5 Einfache Reihenfolgeregeln.....	18
2.6 Agentenbasierte Materialflussteuerung	19
3 Neuronale Netzwerke	21
3.1 Neuronen in der Biologie	21
3.2 Künstliches Neuron und das einfache Perzeptron.....	22
3.3 Künstliche neuronale Netzwerke (Mehrlagige Perzeptren)	23
3.4 Relevante Aktivierungsfunktionen	25
4 Maschinelles Lernen	28
4.1 Lernverfahren	28
4.1.1 Überwachtes Lernen.....	28
4.1.2 Unüberwachtes Lernen.....	30
4.2 Reinforcement Learning	31
4.2.1 Markov-Entscheidungsprozess.....	32

4.2.2	Komponenten eines Reinforcement-Learning-Agenten.....	33
4.2.3	Lernende Multiagentensysteme.....	35
4.3	Auswahl eines geeigneten Lernverfahrens.....	35
4.3.1	Q-Learning.....	35
4.3.2	Deep Q-Learning	37
4.3.3	Double Q-Learning	37
4.3.4	Erfahrungswiedergabe (<i>Experience Replay</i>)	39
4.3.5	Exploration vs. Exploitation.....	39
5	Konzept zur Implementierung eines Reinforcement-Agenten in einer diskreten Ablaufsimulation.....	41
5.1	Interaktion zwischen Agent und Umwelt.....	41
5.2	Ablaufprozess des Simulationsagenten.....	43
5.3	Der lernende Agent.....	44
5.3.1	Initialisierung des lernenden Agenten.....	44
5.3.2	Auswahl von Aktionen	45
5.3.3	Der Lernprozess	46
5.4	Technische Entwicklungsumgebung	47
5.4.1	Simulationssoftware.....	48
5.4.2	Machine-Learning-Umgebung	48
5.4.3	Framework für neuronale Netze	49
5.4.4	Datenschnittstelle	49
6	Umsetzung eines Prototypen zur Anwendung des maschinellen Lernens in der Materialflusssimulation.....	51
6.1	Agentenbasierte Reihenfolgeplanung.....	51
6.2	Umsetzung des Simulationsmodells	52
6.3	Aufbau und Funktionsweise des Simulationsagenten	53
6.3.1	Architektur des Simulationsagenten	53
6.3.2	Erfassen der Zustand-Aktionspaare	55
6.3.3	Die Schrittfunktion.....	57

6.3.4	Die Belohnungsfunktion	59
6.3.5	Datenverwaltung in der Umwelt	60
6.3.6	Verarbeitung der Daten zur Kommunikation mit dem RL-Agenten.....	60
6.4	Aufbau des Reinforcement-Learning-Agenten	62
6.4.1	Schnittstelle des RL-Agenten	62
6.4.2	Datenverarbeitung zur Kommunikation mit dem Simulationsagenten	62
6.4.3	Funktionen für den Ablauf des Lernalgorithmus	64
6.4.4	Aufbau und Design des neuronalen Netzes	66
6.4.5	Verarbeitung der Simulationsdaten in Python	68
7	Auswertung und Analyse.....	69
7.1	Verifikation und Validierung	69
7.2	Lernfähigkeit des Agenten	71
7.3	Vorhersagesicherheit des Netzes.....	71
7.4	Erprobung und kritische Betrachtung des lernenden Agenten	75
8	Zusammenfassung und Ausblick	77
Anhang A: Übersicht der technischen Entwicklungsumgebung		80
Anhang B: Plant-Simulation-Programmcode		81
Anhang B.1:	Die Schrittfunktion mStep	81
Anhang B.2:	Die Belohnungsfunktion des Simulationsagenten.....	82
Anhang B.3:	Ausführen der gewählten Aktion	82
Anhang B.4:	Datentransfer von Plant Simulation via Socket.....	83
Anhang B.5:	Empfangen der Daten aus Python	83
Anhang C: Python-Programmcode		85
Anhang C.1:	Python-Pakete für den RL-Agenten	85
Anhang C.2:	Initialisierung der Agentenklasse	85
Anhang C.3:	Die Funktion zur Auswahl einer Aktion	85
Anhang C.4:	Trainingsfunktion.....	86
Anhang C.5:	Funktion zur Speicherung der Daten im Erfahrungsspeicher	86

Anhang C.6: Die Trainingsfunktion zur Aktualisierung der Q-Werte	86
Anhang C.7: Die Hauptfunktion und der Aufbau der Socket-Verbindung	87
Anhang C.8: Die Aufgaben des RL-Agenten	87
Anhang C.9: ConvData	89
Literaturverzeichnis	90
Eidesstattliche Erklärung	93

Abbildungsverzeichnis

Abbildung 2-1: Aufgabensicht des Aachener PPS-Modells (Lödding 2016, S. 6).....	14
Abbildung 2-2: Der Fehlerkreis der Fertigungssteuerung (nach Mather) (Nyhuis und Wiendahl 2012, S. 5).....	16
Abbildung 2-3: Wirkzusammenhänge der Fertigungssteuerung (Wiendahl 2019, S. 334)	17
Abbildung 3-1: Konzept von Neuronen (Rashid 2017, S. 30)	21
Abbildung 3-2: Modell eines künstlichen Neurons (Rashid 2017, S. 35)	22
Abbildung 3-3: Mathematisches Modell eines Neurons (Frochte 2019, S. 162)	22
Abbildung 3-4: Mehrlagiges Perzepron (Frochte 2019, S. 172).....	23
Abbildung 3-5: Deep Neural Network (Tiefes neuronales Netz) (Frochte 2019, S. 174)	24
Abbildung 3-6: Berechnung der Kantengewichte in einem Netz	24
Abbildung 3-7: Aktivierungsfunktionen	26
Abbildung 3-8: Sigmoid und Tangens Hyperbolicus als Aktivierungsfunktion (Frochte 2019, S. 171).....	26
Abbildung 3-9: Aktivierungsfunktion ReLU (Frochte 2019, S. 211).....	27
Abbildung 4-1: Lineare Klassifikation von Daten (Eigene Darstellung)	29
Abbildung 4-2: Lineare Regression (Eigene Darstellung)	30
Abbildung 4-3: Cluster und Assoziation (Eigene Darstellung).....	31
Abbildung 4-4: Interaktion zwischen Agent und Environment in einem Markov-Entscheidungsprozess (Sutton und Barto 2018, S. 48)	32
Abbildung 5-1: Interaktion zwischen Simulationsagent und lernendem Agenten.....	42
Abbildung 5-2: Prozess der Interaktion zwischen Simulationsagent und lernendem Agent ..	43
Abbildung 5-3: Konzept der Simulationsumgebung und des Simulationsagenten	44
Abbildung 5-4: Initialisierungsprozess des Agenten.....	45
Abbildung 5-5: Auswahlprozess einer Aktion des Agenten	45
Abbildung 5-6: Ablauf des Lernprozess	46
Abbildung 6-1: Das angewandte Simulationsmodell	52
Abbildung 6-2: Aufbau des Agenten in Plant Simulation	54
Abbildung 6-3: Ablauf der mStep Funktion.....	57
Abbildung 6-4: Auswahl einer Aktion (Vorgänger-Nachfolger).....	58
Abbildung 6-5: Die Belohnungsfunktion des Agenten	59
Abbildung 6-6: Datentransferprotokoll (Memory)	61
Abbildung 6-7: Fully connected Deep Neural Network mit zwei Schichten.....	68
Abbildung 7-1: Durchschnittliche Belohnung des RL-Agenten.....	70
Abbildung 7-2: Lernerfolge des RL-Agenten	71
Abbildung 7-3: Fehlgeschlagene Episoden – Netzfehler vs. Zufallsfehler	72

Abbildung 7-4: Vorhersagequalität mit erhöhtem Trainingsumfang	73
Abbildung 7-5: Vorhersagequalität ohne Zufallsauswahl	74
Abbildung 7-6: Korrekt getroffene Netzvorhersagen	74

Tabellenverzeichnis

Tabelle 6-1: Eigenschaften des Pufferbausteins	52
Tabelle 6-2: Eigenschaften der Stationen	53
Tabelle 6-3: Prozesszeiten der Arbeitsstationen	53
Tabelle 6-4: Rüstzeitmatrix der Stationen	53
Tabelle 6-5: Zustandsgrößen (observation space).....	56
Tabelle 6-6: Aktionsraum (action space).....	56
Tabelle 6-7: Übersicht Belohnung und Bestrafung.....	60
Tabelle 6-8: Hyperparameter von Agent und neuronalem Netz.....	63
Tabelle 7-1: Statische Berechnung des Durchsatzes	69
Tabelle 7-2: Validierung anhand der Durchsätze	70

Abkürzungsverzeichnis

KNN	<i>künstliche neuronale Netze</i>
MDP	<i>Markov Decision Problem</i>
MLP	<i>multi layer perceptrons</i>
RL	<i>Reinforcement Learning</i>
TCP/IP	<i>Transmission Control Protocol/Internet Protocol</i>
UDP	<i>User Datagram Protocol</i>

1 Einleitung

1.1 Problemstellung

Der Trend zu individualisierten Produkten und die damit einhergehende hohe Variantenvielfalt führen zu stark verkürzten Produktlebenszyklen. Wandlungsfähige und flexibel vernetzte Strukturen und Abläufe der Produktionssysteme sind daher gefordert. Nur so können Logistiksysteme hinsichtlich Größe, Funktion und Struktur auf die sich rasant verändernden Randbedingungen angepasst werden (Wilke 2006, S. 7). Im Rahmen der Umsetzung von Industrie 4.0 wird sich die Logistik zur kognitiven Logistik weiterentwickeln, um schnell und flexibel auf das volatile Umfeld in der Produktion zu reagieren. Unternehmen stattet aktuell Anlagen und Maschinen mit immer stärker vernetzten Sensoren aus. Diese Sensoren stellen ihnen mehr Informationen zur Verfügung und ermöglichen, die Produktion anhand aufgezeichneter Daten zu optimieren und Rückschlüsse und Wissen zu extrahieren (IBM Manufacturing 2018, S. 3).

Durch den Trend der Dezentralisierung und Individualisierung der Produktion verändern sich auch die Ziele der Logistik hin zu möglichst anpassungsfähigen und ressourcensparenden Prozessen. Damit soll auch in Zukunft das richtige Produkt zum richtigen Zeitpunkt am richtigen Ort in der richtigen Menge und Qualität bereit gestellt werden (Bauernhansl et al. 2014, S. 297–298). Insbesondere die Logistik steht vor der Herausforderung, Unternehmen einen schnellen und flexiblen Zugang zu den immer stärker schwankenden Märkten zu sichern. Dies ist mit starren Materialflüssen und Dispositionstrategien jedoch kaum möglich. Logistik- und Intralogistikprozesse müssen sich demnach eigenständig vernetzen, Informationen austauschen und selbstständig den Materialfluss steuern, um die Wandelbarkeit von Logistiksystemen zu erhöhen (Bauernhansl et al. 2014, S. 298). Dies erfordert dezentrale Konzepte, die ein hohes Maß an Standardisierung und Modularisierung der Systeme ermöglichen.

Durch die geänderten Anforderungen steht die Auswahl neuer Software-Methoden und -Werkzeuge zur Vereinfachung bis hin zur Selbstorganisation der Systeme im Fokus der Forschung. Dadurch entstehen agilere Systeme mit einheitlichen Schnittstellen zur Interaktion zwischen den Steuerungsprogrammen. Erreicht werden kann dies durch eine Modularisierung der Funktionen und ihrer damit einhergehenden Wiederverwendbarkeit. Ein in den letzten Jahren zunehmend an Bedeutung gewinnender Ansatz ist der sogenannte Agentenansatz aus dem Gebiet der verteilten künstlichen Intelligenz. Dabei bietet das Konzept der Software-Agenten einen besseren Umgang mit Systemkomplexität. Die Agenten kommunizieren selbstständig durch einheitliche Schnittstellen, können unabhängig voneinander entwickelt werden und sind daher leichter austauschbar und wiederverwendbar. Diese Eigenschaften

machen die Agententechnologie zu einem vielversprechenden Ansatz für die Steuerung von Materialflüssen in Fertigungssystemen (Günthner und Hompel 2010, S. 25–29).

Zur Implementierung solcher Steuerungskonzepte wurden bereits von der Technischen Universität Dortmund und dem Fraunhofer-Institut für Materialfluss eine Referenzimplementierung von fahrerlosen Transportfahrzeugen auf Agentenbasis durchgeführt. Dort werden Daten der Akteure eines realen logistischen Systems in eine Simulationsumgebung übertragen, wo in einer virtualisierten Umgebung das Lernen stattfindet. Das digitale Abbild des Systems erlernt das beste Verhalten über Reinforcement Learning auf Basis realer Daten des physischen Systems. Dieses Wissen wird in neuronalen Netzen abgebildet und an die Akteure des Systems zurückgespielt (Murrenhoff et al. 2019, S. 8–11). Der Ansatz, das Lernen in einer virtualisierten Simulationsumgebung durchzuführen, wird in dieser Arbeit aufgegriffen und der Zielstellung angepasst.

1.2 Zielstellung

Ausgehend von der in der Problemstellung beschriebenen Notwendigkeit wandlungsfähige und flexible Logistik- und Produktionssysteme durch dezentrale Strukturen und intelligente Steuerungsprinzipien zu schaffen und diese mittels virtueller Prototypen abzusichern, wurde die Zielstellung der vorliegenden Arbeit formuliert. Die Zielsetzung ist ein Konzept zu entwickeln, das eine agentenbasierte Steuerung von Materialflüssen in der diskret ereignisorientierten Simulation mittels maschinellem Lernen ermöglicht. Ein weiteres zentrales Ziel ist die prototypische Umsetzung des erarbeiteten Konzeptes im Simulationswerkzeug Plant Simulation. Hierfür wird anhand eines Modells mit geringer Komplexität die Steuerungsaufgabe der Reihenfolgebildung abgebildet. Unter Anwendung des bestärkenden Lernens wird ein neuronales Netz im Open Source Framework Keras trainiert, das die Entscheidungstaktik für die Auftragsverteilung repräsentiert. Zur Bewertung der Entscheidungsqualität wird das neuronale Netz einer heuristischen Abfrage sowie einer zufallsverteilten Reihenfolgebildung gegenübergestellt.

1.3 Aufbau und Vorgehensweise

Die Arbeit ist in acht Kapitel gegliedert. Das **zweite Kapitel** führt an die grundlegenden Zusammenhänge in der Produktions- und Fertigungssteuerung heran. Im ersten Schritt werden die logistischen Zielgrößen und deren Wirkzusammenhänge mit den Stellgrößen der Fertigungssteuerung dargelegt. Weiter befasst sich dieses Kapitel mit grundlegenden Steuerungsmechanismen zur Bestimmung von Auftragsreihenfolgen durch einfache lokale Regeln. Im Anschluss wird auf Unterschiede und Vorteile des zentralen und dezentralen Steuerungsprinzips eingegangen. Den Abschluss bildet die Beschreibung des

agentenbasierten Ansatzes zur Steuerung von Materialflüssen als einen dezentralen Lösungsansatz der Fertigungssteuerung.

Das **dritte Kapitel** führt an den Einsatz neuronaler Netze als Entscheidungsträger für die Materialflussteuerung heran. Dazu wird anhand der grundlegenden Funktionsweise eines Neurons die mathematische Abbildung eines Neurons beschrieben. Anschließend folgt anhand eines Beispiels die Darstellung des Aufbaus und des Ablaufs innerhalb eines neuronalen Netzes.

Das **vierte Kapitel** stellt die benötigten maschinellen Lernverfahren für die Anpassung der Kontengewichte der neuronalen Netze vor und grenzt diese ab. Im Weiteren wird vertieft auf das bestärkende Lernen von Agenten eingegangen und die einzelnen Komponenten herausgearbeitet. Abschließend wird ein geeigneter Lernalgorithmus ausgewählt und dessen theoretischer Ablauf durch Ansätze aus aktuellen Forschungen erweitert.

Im **fünften Kapitel** findet die Entwicklung des Konzeptes zur simulationsbasierten Anwendung von maschinellem Lernen in der Materialflussteuerung statt. Der erste Schritt legt die grobe Interaktionsebene zwischen Simulationsumgebung und dem bestärkenden Agenten fest, auf deren Basis die Prozesse der beiden Elemente entwickelt werden. Im letzten Schritt der Konzeptionierung wird eine technische Entwicklungsumgebung ausgewählt, anhand der die prototypische Umsetzung durchgeführt wird.

Im **sechsten Kapitel** erfolgt die prototypische Umsetzung des Konzepts mit dem Simulationswerkzeug Plant Simulation und dem neuronalen Netz Framework Keras. Hierzu wird in einem ersten Schritt ein Modell zur Bildung rüstopimaler Reihenfolgen erstellt. Anschließend folgen im zweiten Schritt der Aufbau und Prozessablauf des entwickelten Simulationsagenten, sowie die Definition eines Datenverarbeitungsprotokolls zur Kommunikation mit dem bestärkenden Agenten. Zuletzt wird die Entwicklung und Umsetzung des bestärkenden Lernalgorithmus erläutert und dessen Aufgaben beschrieben, sowie dessen Kommunikation mit dem Simulationsagenten über ein WebSocket erarbeitet.

Im **sieben Kapitel** werden die beiden Elemente Simulationsagent und lernender Agent verifiziert und validiert. Anschließend findet die Analyse der Lernfähigkeit und der Vorhersagesicherheit des Netzes statt. Abschließend erfolgt eine Gegenüberstellung vom prototypischen Agenten, einer herkömmlichen Heuristik und einer Zufallsstrategie.

Die Schlussbetrachtung im **achten Kapitel** fasst die wesentlichen Ergebnisse dieser Arbeit zusammen und gibt einen kurzen Ausblick zur Weiterentwicklung des Konzeptes und der prototypischen Umsetzung.

2 Produktionsplanung und -Steuerung

Das Internet der Dinge erfordert eine tiefgreifende Veränderung der Steuerungsarchitektur von Materialflussanlagen und auch der Denkweise der Anlagenbauer und –programmierer. Denn durch die Abschaffung aller hierarchischen Strukturen müssen für die meisten Probleme neue Lösungen erarbeitet werden – an die Stelle zentral und fest einprogrammierter Abläufe und Strategievorgaben treten dezentrale Verhaltensregeln und Kooperationsmechanismen (Günthner und Hompel 2010, S. 46). Daher wird in diesem Kapitel zunächst auf die Einordnung der Steuerung von Fertigungssystemen im Gesamtprozess eingegangen und die Zielgrößen und die Wirkzusammenhänge der Steuerungsprozesse dargelegt. Anschließend werden zentrale Steuerungsprinzipien den dezentralen gegenübergestellt und der agentenbasierte Ansatz zur Materialflusssteuerung betrachtet.

2.1 PPS-Systeme

Die Produktionsplanung- und Steuerung (PPS) hat die Aufgabe, „das laufende Produktionsprogramm in regelmäßigen Abständen nach Art und Menge für mehrere Planungsperioden im Voraus zu planen“. Unter Berücksichtigung von Kapazitäten und unvorhergesehenen Störungen ermittelt sie die Material- und Ressourcenbedarfe für die Fertigung (Wiendahl 2019, S. 245). Die Produktionsplanung gestaltet den Inhalt und die Prozesse der Fertigung und Montage, wohingegen die Produktionssteuerung den Ablauf der Tätigkeiten bestimmt. Sie steuert daher die zeitliche Abfolge der Teilprozesse unter Berücksichtigung des Produktionsplans sowie der logistischen Zielgrößen (Schuh und Stich 2012, S. 29). Nach dem Aachener PPS-Modell können die Aufgaben der Produktionsplanung und Steuerung in Kernaufgaben und Querschnittsaufgaben gegliedert werden wie in Abbildung 2-1 dargestellt.

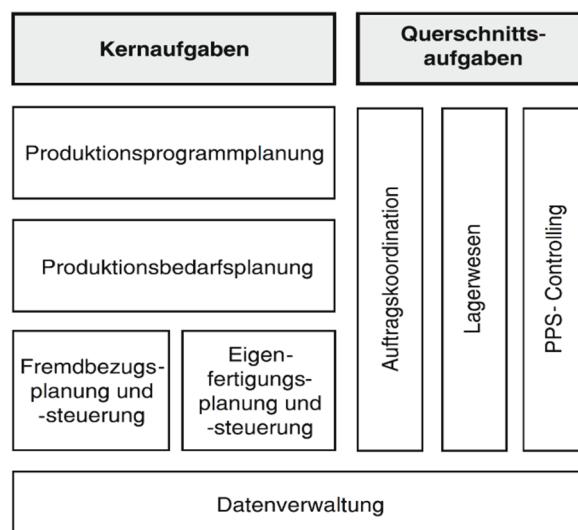


Abbildung 2-1: Aufgabensicht des Aachener PPS-Modells (Lödding 2016, S. 6)

Darin wird die Steuerung der Fertigung vor allem im Bereich der Eigenfertigungsplanung und -Steuerung angesiedelt, die zwar im gesamten PPS-Modell nur einen kleinen Teil einnimmt, jedoch für die Erreichung der logistischen Zielgrößen sehr wichtig ist. Daher ist es notwendig in den folgenden Kapiteln die logistischen Zielgrößen und deren Wirkzusammenhänge mit den Stellgrößen zu erörtern.

2.2 Logistische Zielgrößen

Die zentrale Aufgabe der Produktionslogistik ist es, die gegenseitig abhängigen logistischen und wirtschaftlichen Ziele bestmöglich zu erfüllen. Wiendahl (Wiendahl 2019, S. 246) unterscheidet dabei zwischen zwei Zielgrößen: Logistikeistung und Logistikkosten. Die *Logistikeistung* wird durch die Lieferzeit und die Liefertreue beeinflusst. Werden vom Kunden geringe Lieferzeiten verlangt, erfordert dies kurze Durchlaufzeiten in der Produktion. Wenn der Kunde hingegen eine hohe Liefertreue fordert, bedarf es einer hohen internen Termintreue. *Logistikkosten* auf der anderen Seite ergeben sich aus Kapitalbindungskosten durch hohe Bestände und den logistischen Prozesskosten wie beispielsweise Ein- und Auslagerungsvorgängen.

Das übergeordnete Ziel eines Unternehmens ist eine hohe Wirtschaftlichkeit durch möglichst geringe Herstellkosten. Eine möglichst hohe Auslastung der Maschinen erfordert auch hohe Bestände. Diese führen zu langen und schwankenden Durchlaufzeiten, die wiederum das Ziel einer hohen Termsicherheit gefährden. Diese Faktoren beeinflussen vor allem die Lagerhaltung des Produktionssystems. Bei erhöhter Durchlaufzeit und geringer Termintreue muss genügend Bestand im Lager sein, um den Servicegrad aufrecht zu halten. Je nachdem wie stark kundenbezogen das Produktionssystem ist, wird abgewogen, ob es sich lohnt mit einer geringeren Auslastung der Produktionsressourcen eine verkürzte Lieferzeit zu erreichen. Der damit einhergehende erhöhte Ressourcenaufwand führt wiederum zu höheren Stückkosten. Diese Problematik wird als Dilemma der Ablaufplanung bezeichnet (Nyhuus und Wiendahl 2012, S. 5–6). Ein Gesamtoptimum zur Ausrichtung des Produktionssystems ist so kaum zu erreichen und kann zum sogenannten Fehlerkreis der Fertigungssteuerung führen wie in Abbildung 2-2 dargelegt.

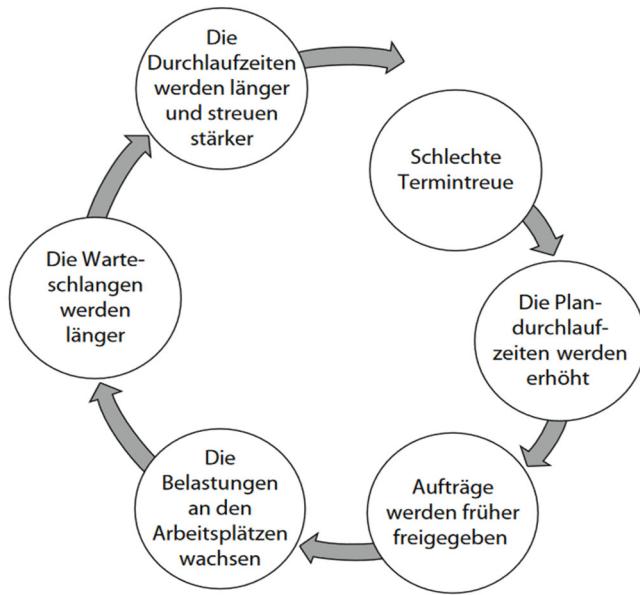


Abbildung 2-2: Der Fehlerkreis der Fertigungssteuerung (nach Mather) (Nyhuis und Wiendahl 2012, S. 5)

Eine schlechte Termintreue führt zu erhöhten Plandurchlaufzeiten, was wiederum frühere Freigaben der Aufträge bedingt. Dadurch wachsen die Belastungen an den einzelnen Arbeitsstationen, die zu längeren Warteschlangen in den vorgelagerten Puffern führen und somit die Durchlaufzeit und deren Streuung in die Höhe treiben. Um das Dilemma der Ablaufplanung näher zu erläutern, soll nun mit dem Blick auf die Fertigungssteuerung auf die Abhängigkeiten der logistischen Zielgrößen und auf Möglichkeiten, diese zu beeinflussen, eingegangen werden.

2.3 Wirkzusammenhänge der Fertigungssteuerung

Die Fertigungssteuerung hat die Aufgabe, das geplante Produktionsprogramm in der Fertigung umzusetzen. Die Zusammenhänge der einzelnen Größen kann anhand des Modells der Fertigungssteuerung von Lödding, durch Wiendahl (Wiendahl 2019, S. 334) um die Funktionen des Trichtermodells ergänzt, in Abbildung 2-3 erläutert werden.

Als Teil der Materialbedarfsplanung bestimmt die Auftragserzeugung die Stellgrößen Soll-Zugang, Soll-Abgang und Soll-Reihenfolge. Durch die Freigabe der Aufträge wird das Intervall des Materialzugangs zur Fertigung (Ist-Zugang) geregelt. Zusammen mit der Kapazitätssteuerung, die die verfügbare Kapazität (Ist-Abgang) eines Prozesses regelt, beeinflussen sie die Regelgröße Bestand. Diese wiederum bestimmt die Zielgrößen Bestand, Durchlaufzeit und Auslastung. Die Funktion der Reihenfolgebildung steuert die Abfolge der Aufträge im Trichter bzw. an den Arbeitsstationen. Muss die geplante Reihenfolge abgeändert werden, entsteht eine Reihenfolgeabweichung, die gemeinsam mit dem Rückstand aus Soll- und Ist-Abgang auf die Termintreue Einfluss nimmt (Wiendahl 2019, S. 333–334).

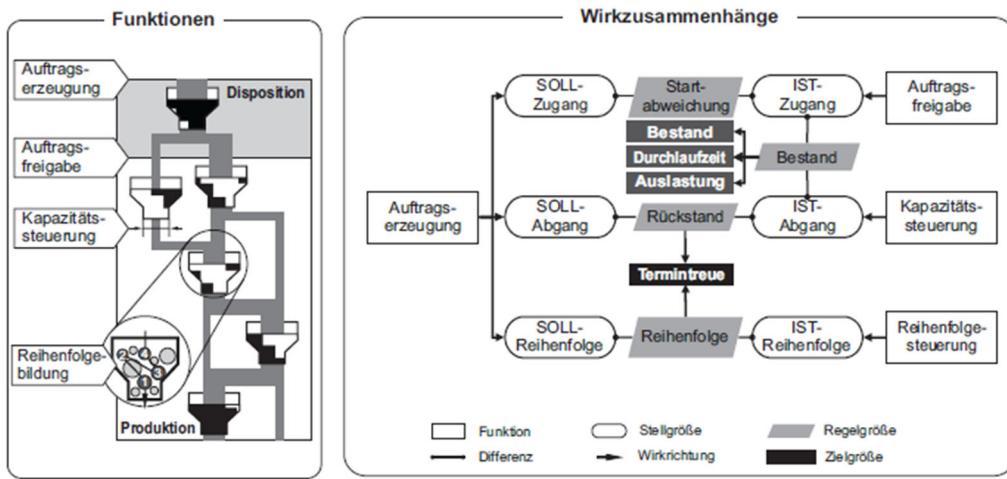


Abbildung 2-3: Wirkzusammenhänge der Fertigungssteuerung (Wiendahl 2019, S. 334)

2.4 Zentrale und dezentrale Fertigungssteuerung

Die Herausforderung der Fertigungssteuerung, die logistischen Zielgrößen ständig aufeinander abzustimmen, wird durch die Entwicklung des Internet der Dinge vom klassischen Ansatz der zentralen hin zur dezentralen Steuerung der Materialflüsse verschoben. Um ein Materialflusssystem flexibel und wandelbar zu gestalten, ist es notwendig, dieses zu modularisieren, um so mit den einzelnen Komponenten eine neue Architektur zu erzeugen (Günthner und Hompel 2010, S. 31).

Ein klassischer PPS-Ansatz zeichnet sich nach Hachtel und Holzbaur (2010, S. 114) durch eine zentrale Organisation aus. In diesem Ansatz wird die Planung und Steuerung der Fertigungsaufträge von einer zentralen Instanz aus übernommen. Vorteil dieses Ansatzes ist es, dass die benötigten Werkzeuge und das erforderliche Wissen zentral gesammelt und verwaltet werden (Jodlbauer 2008, S. 107). Das Zusammenführen von Systemen zielt auf die Reduzierung des Koordinationsaufwandes zwischen den Organisationseinheiten. Eine zentrale Koordination der Planungs- und Steuerungsprozesse ist nach Schuh und Stich (Schuh und Stich 2012, S. 302) auf einer groben Planungsebene sinnvoll. Dadurch können globale Ziele der gesamten Wertschöpfungskette erreicht werden.

Daraus ergeben sich allerdings auch eine Reihe von Nachteilen. Zum einen ist ein sehr hoher Aufwand notwendig, um alle Betriebsdaten zentral zu erfassen. Zum anderen entstehen dadurch oft Kommunikationsprobleme zwischen den Abteilungen. Erfahrungen und Wissen aus den ausführenden Abteilungen können in der zentralen Instanz bei Entscheidungen kaum mit einfließen, was zu einer geringen Akzeptanz auf operativer Ebene führt (Hachtel und Holzbaur 2010, S. 114).

Bei einer dezentralen Ausrichtung der Fertigungssteuerung ist das Ziel, die Unabhängigkeiten der Systeme zu erhöhen und möglichst schnell auf Kundenwünsche und Schwankungen des Marktes reagieren zu können. Daher erfordert die Dezentralisierung der Fertigung, die Fähigkeit zur Entscheidungsfindung innerhalb einzelner Komponenten und Module um wandlungsfähige und flexible Fertigungssysteme zu realisieren. (Günther und Hompel 2010, S. 31). Folglich schafft die Dezentralisierung vor allem in der Durchführung der Steuerung eine erhöhte Flexibilität aufgrund kurzer Entscheidungswege. Zusätzlich führt dies zu einer deutlichen Reduzierung des Planungs- und Steuerungsaufwands und zu einer besseren Kontrolle der Bestände. Allerdings erhöht dieser Ansatz für übergeordnete Systeme auch den Koordinationsaufwand. Zudem können die Ziele einer Organisationseinheit im Konflikt mit dem Gesamtziel des Unternehmens stehen (Schuh und Stich 2012, S. 301). Ein bekanntes und weit verbreitetes Beispiel für eine dezentrale Steuerung der Fertigung ist das Kanban-Prinzip. Die Grundidee besteht darin, die Fertigung in Gruppen von selbststeuernden Regelkreisen zu teilen (Schenk et al. 2014, S. 399).

Die im Rahmen dieser Arbeit näher betrachtete Steuerung der Reihenfolge mittels intelligenter Agenten, bestimmt die Warteschlangen vor den Prozessen. Relevant für die Zielgrößen Logistikleistung und Logistikkosten ist die Reihenfolgeplanung besonders in variantenreichen Fertigungssystemen. Um im weiteren Verlauf der Arbeit die verschiedenen Strategien bei der Steuerung der Reihenfolge vergleichen zu können, soll nun zunächst auf klassischen Reihenfolgeregeln näher eingegangen werden.

2.5 Einfache Reihenfolgeregeln

Zur Bildung von Reihenfolgen können unterschiedliche Regeln zum Abarbeiten einer Warteschlange eingesetzt werden. Jeder Auftrag wird nach bestimmten Kriterien zur Bearbeitung an einem freien Prozess in der Warteschlange priorisiert. Die Wirkung von Prioritätsregeln hängt stark von den Beständen vor den Prozessen ab. Je geringer die Bestände sind, desto weniger Einfluss hat eine Priorisierung von Aufträgen auf die Leistung und die Durchlaufzeit eines Systems. (Wiendahl 2019, S. 333)

In der Praxis häufig angewendete Prioritätsregeln sind:

- FIFO-Prinzip (First In – First Out)
- KOZ-Regel (kürzeste Operationszeit)
- LOZ-Regel (längste Operationszeit)
- FPE-Regel (frühester Plan-Endtermin)
- Schlupfzeit-Regel
- Rüstzeitminimale Reihenfolgen

Beim *FIFO-Prinzip* werden Aufträge in der Reihenfolge abgearbeitet, wie sie an der Arbeitsstation ankommen. Bei der *KOZ-Regel* haben die Aufträge mit dem geringsten Arbeitsinhalt Vorrang, wodurch eine Verringerung der mittleren Durchlaufzeit erreicht wird. Die Priorisierung von Aufträgen mit dem größten Arbeitsinhalt verlängert hingegen die mittlere Durchlaufzeit (entspricht der *LOZ-Regel*). Die *FPE-Regel* bevorzugt diejenigen Aufträge, deren Endtermin am nächsten in der Zukunft liegt. Konkret sind dies Aufträge, die am frühestens fertiggestellt werden müssen oder verspätet sind. Die *Schlupfzeit-Regel* priorisiert Aufträge mit den geringsten Zeiten bis zum planmäßigen Fertigstellungszeitpunkt. Dies wird auch als Rest-Schlupf bezeichnet. Hierfür können z. B. vor den Arbeitsstationen Reihenfolgen gebildet werden, die den Umstellungsaufwand bzw. Rüstaufwand des Arbeitssystems minimal halten (Wiendahl 2019, S. 334–335). Eine Veränderung der Reihenfolge der Arbeitsaufträge während des Produktionsprozesses hat jedoch auch zur Folge, dass geplante Endtermine anderer Aufträge nicht eingehalten werden können.

Die rüstzeitabhängige Priorisierung der Arbeitsaufträge nimmt unter den Reihenfolgeregeln einen besonderen Stellenwert ein, da sie sich positiv auf die Leistung eines Produktionssystems auswirkt, indem sie zusätzliche Kapazitäten freigibt. Rüstzeitregelungen erreichen dadurch zwar eine Bestandssenkung im System, sie wirken allerdings bei größeren Beständen besser. (Wiendahl 2019, S. 335)

2.6 Agentenbasierte Materialflussteuerung

Im Gegensatz zu den herkömmlichen, festen Regeln und Heuristiken zur Steuerung von Materialflüssen, stellt im Rahmen des Wandels zu Industrie 4.0 von Produktionssystemen das Internet der Dinge eine Vision eines hierarchiellen Steuerungsprinzips dar, bestehend aus untereinander kooperierenden autonomen Einheiten (Günthner und Hompel 2010, S. 97). Einen Ansatz, um dezentrale Materialflussteuerungen zu realisieren, sind Multiagentensysteme, deren Bedeutung im Produktionsbereich deutlich zunimmt. Gründe hierfür sind laut Westkämper (Westkämper und Zahn 2009, S. 167) spezielle Interaktionsmechanismen der Agentensysteme, die eine Kommunikation und Kooperation zwischen den einzelnen Agenten ermöglicht. Außerdem basiert ein agentenorientiertes Konzept auf bereits bestehenden objektorientierten Programmentwicklungen der PPS-Systeme. Dadurch wird sowohl die Flexibilität und Erweiterbarkeit der Systeme erhöht, als auch die Modularität bei der Ressourcenverteilung. Das bedeutendste Argument, die Agententechnologie umzusetzen, besteht darin, dass diese einfach in eine bestehende IT-Infrastruktur integrierbar ist (Westkämper und Zahn 2009, S. 167).

Für die Entwicklung und Umsetzung einer funktions- und leistungsfähigen, dezentralen, und multiagentenbasierten Steuerung von Materialflusssystemen ist aufgrund der hohen

Systemkomplexität der Einsatz von Simulation notwendig (Günthner und Hompel 2010, S. 151). Stand der Technik bilden hier kommerzielle Simulationswerkzeuge zur Modellierung von intralogistischen Abläufen. Um die Agenten auszuführen und zu verwalten, werden spezielle Umgebungen, sogenannte Softwareagentenplattformen, entwickelt. Hierfür existieren bereits Standards wie z. B. der FIPA-Standard von der Foundation for Intelligent Physical Agents (FIPA) oder die Mobile Agent Facility Specification (MAF). Dabei spezifiziert der FIPA-Standard Systemarchitekturen und die Kommunikation, während die MAF die mobilen Agenten verwaltet (Günthner und Hompel 2010, S. 36).

Um ein Produktions- und Logistiksystem steuern zu können, bedarf es eines Elements, das Entscheidungen in einem System trifft. Beim dezentralen Ansatz übernehmen dies einzelne autonome Einheiten, anstatt eines zentralen PPS-Systems. Beispielsweise können die Entscheidung über Reihenfolgen von Aufträgen, von festen Regeln wie der FIFO-Regel, oder aber von Heuristiken bestimmt werden, die eine Strategie verfolgen. Bei Heuristiken können jedoch manche Faktoren, wie beispielsweise die Auswirkungen aktuell eingeplanter Arbeitsgänge auf zukünftige, nicht berücksichtigen werden. In Betracht von Rüstvorgängen hat dies oftmals hohe Rüstaufwände und lange Durchlaufzeiten zur Folge.

In Verbindung mit der agentenbasierten Steuerung von Materialflüssen, befindet sich die Steuerungsentscheidung jedoch im Agent. Bis vor wenigen Jahren war die beste mathematische Herangehensweise aufgrund fehlender Rechnerleistung eine Schritt-für-Schritt-Planung mittels Heuristik. Mit den heutzutage deutlich leistungsfähigeren Rechnern können sehr viel größere Datenmengen verarbeitet und bessere Algorithmen entwickelt werden, die intelligentes Verhalten durch die Integration von KI und neuronaler Netze ermöglichen.

3 Neuronale Netzwerke

Anstelle fester Regeln soll nun ein neuronales Netzwerk die Steuerungsentscheidung treffen. Dieses wird mit Verfahren des maschinellen Lernens so trainiert, dass es eine Entscheidungsstrategie des Agenten abbilden kann. Daher wird in diesem Kapitel zunächst auf die grundlegende Funktionsweise von Neuronen eingegangen. Anschließend wird diese in eine mathematische Form gebracht und zu Netzwerken verbunden. Hierfür bedarf es noch der Abgrenzung relevanter Aktivierungsfunktionen der Neuronen, um die Charakteristik eines Netzes zu bestimmen.

3.1 Neuronen in der Biologie

Ein Neuron ist eine Zelle, die elektrische Aktivität aufnimmt, verarbeitet und diese weitergibt. Vom Zellkörper des Neurons gehen kurze Verästelungen ab, welche Dendriten genannt werden, sowie ein langer Fortsatz, das sogenannte Axon. Dies stellt eine feste Verbindung zur Kommunikation mit anderen Neuronen her. Am Ende des Axons sind kleine Verästelungen, die Endknöpfchen (siehe Abbildung 3-1). Sie sind sehr nahe an den Dendriten anderer Neuronen positioniert und berühren diese fast. Diese eng beisammen liegenden Dendriten und Endknöpfchen werden als Synapsen bezeichnet und leiten elektrische Signale weiter. Entsteht nun eine Änderung des elektrischen Potentials an den Synapsen, werden elektrischen Signale vom Neuron akkumuliert und beim Erreichen des Schwellenwertes über das Axon weitergeleitet. Dies führt bei den Endknöpfchen dazu, dass das elektrische Signal an die Dendriten des naheliegenden Neurons weitergegeben wird (Kruse et al. 2011, S. 9).

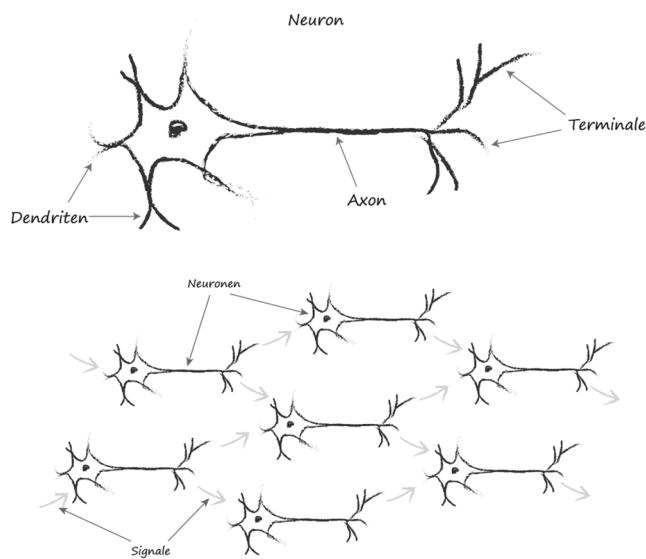


Abbildung 3-1: Konzept von Neuronen (Rashid 2017, S. 30)

3.2 Künstliches Neuron und das einfache Perzepron

Ähnlich dem Neuron eines Gehirns, empfängt auch ein künstliches Neuron mehrere Eingangssignale, verarbeitet diese und gibt ein anderes Signal aus, wie in Abbildung 3-2 zu erkennen. Jedoch reagiert das Neuron nicht sofort auf eine Eingabe, sondern unterdrückt diese, bis sie ausreichend groß ist, um ein Ausgabesignal auszulösen. Dies wird als Schwellenwert bezeichnet, der erreicht werden muss, bevor ein Ausgabesignal entsteht. Dies verhindert, dass kleinste Rauschsignale herausgefiltert werden (Rashid 2017, S. 32).

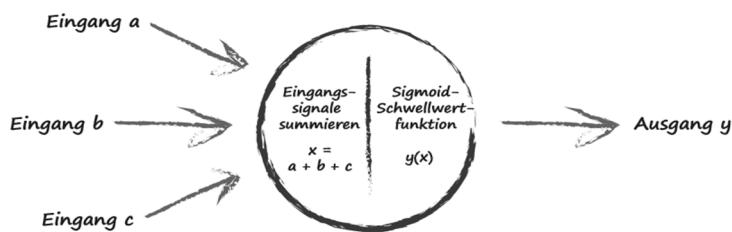


Abbildung 3-2: Modell eines künstlichen Neurons (Rashid 2017, S. 35)

Ein sogenanntes Perzepron überträgt die grundlegende Funktionsweise von Nervenzellen in ein mathematisches Modell. Das einlagige Perzepron ist ein vereinfachtes künstliches neuronales Netz, bestehend aus einem einzelnen künstlichen Neuron ergänzt mit anpassbaren Gewichtungen und einem Schwellenwert mittels Aktivierungsfunktion. Das mathematische Modell eines einfachen Perzeprons ist in Abbildung 3-3 dargestellt.

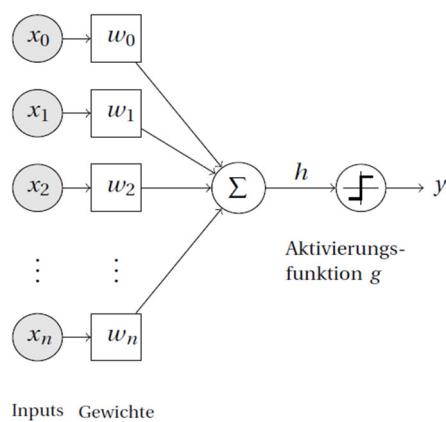


Abbildung 3-3: Mathematisches Modell eines Neurons (Frochte 2019, S. 162)

Das Perzepron erhält einzelne Eingangswerte, welche mit den Gewichten multipliziert und anschließend aufsummiert werden. Das Ergebnis ist wiederum das Eingangssignal für die Aktivierungsfunktion. Wird der Schwellenwert der Aktivierungsfunktion überschritten, feuert auch das künstliche Neuron und überträgt das Signal an die nachfolgenden Neuronen (Frochte 2019, S. 162).

3.3 Künstliche neuronale Netzwerke (Mehrlagige Perzeptren)

Künstliche neuronale Netze (KNN) sind „informationsverarbeitende Systeme, deren Struktur und Funktionsweise dem Nervensystem und speziell dem Gehirn von Tieren und Menschen nachempfunden sind. [...] Diese Neuronen senden sich Informationen in Form von Aktivierungssignalen über gerichtete Verbindungen zu“ (Kruse et al. 2011, S. 7).

Erweitert man das Netz aus Kapitel 3.2 um eine weitere Schicht zwischen dem Input- und Output-Layer, entsteht ein sogenannter Hidden-Layer. Dadurch entstehen mehrlagige Perzeptrone bzw. *multi layer perceptrons* (MLP). Dies ist eine Klasse künstlicher neuronaler Netze, die aus mindestens drei Schichten von Knoten bestehen (siehe Abbildung 3-4) (Frochte 2019, S. 172).

Der Ausgang eines Perzeptrons wird als Eingang für ein anderes Perzepron verwendet. Auf diese Weise können komplexe mehrschichtige Netzwerke aufgebaut werden. Die Komplexität eines Netzes wird durch die Anzahl der Neuronen innerhalb einer Schicht, also durch die Größe der Schichten, sowie der Anzahl der Schichten bestimmt. Meist ist die Architektur eines solchen Netzes vollvermascht bzw. *fully connected*. Das bedeutet, die Knoten einer Schicht sind mit allen Knoten der nachfolgenden Schichten verbunden. Neuronen der gleichen Schicht haben keine Verbindung untereinander. Das MLP gehört daher zur Klasse der *Feed-Forward-Netze* (Frochte 2019, S. 174). Es existieren verschiedene Architekturen von Netzen, wie z. B. Netze bei denen zwar alle Neuronen einer Schicht der nächsten Schicht verbunden sind, aber es gibt auch Verbindungen zu Neuronen der übernächsten Schicht, sogenannte Abkürzungen oder Shortcuts.

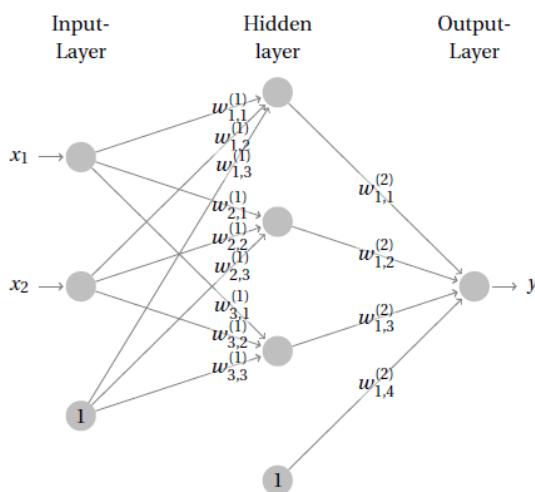


Abbildung 3-4: Mehrlagiges Perzepron (Frochte 2019, S. 172)

Werden mehr als zwei Hidden-Layer in einem Netzwerk verwendet, spricht man bereits von einem Deep Neural Network (siehe Abbildung 3-5).

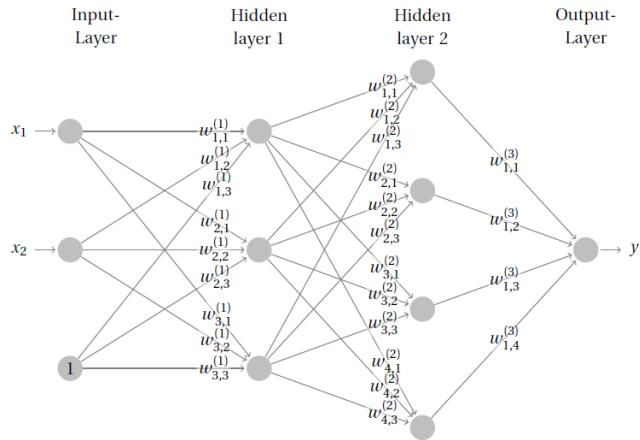


Abbildung 3-5: Deep Neural Network (Tiefes neuronales Netz) (Frochte 2019, S. 174)

Berechnungsbeispiel

Nachfolgend werden anhand eines einfachen Beispielnetzes nach (Rashid 2017, S. 35) Berechnungen in einem neuronalen Netz dargestellt.

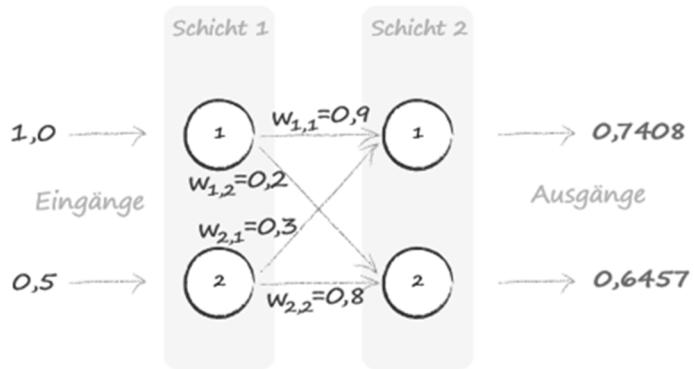


Abbildung 3-6: Berechnung der Kantengewichte in einem Netz

Die erste Schicht eines Netzes, die Eingabeschicht, stellt die Eingangssignale dar und leitet sie direkt an die nachfolgenden Neuronen weiter, ohne eine Aktivierungsfunktion darauf anzuwenden. In der zweiten Schicht werden nun für jeden Knoten die zusammengefassten Eingangssignale ermittelt. Die anfangs zufällig gewählten Verknüpfungsgewichte der verbindenden Kanten müssen hierfür berücksichtigt werden. In dem Beispiel aus Abbildung 3-6 gibt es zwei Eingangsknoten in der ersten Schicht mit den Rohwerten 1,0 und 0,5. Von beiden Knoten besteht eine Verbindung zum ersten Knoten der zweiten Schicht mit den Kantengewichten von 0,9 und 0,3. Somit berechnet sich der kombinierte Eingang wie folgt:

$$x_1 = (1,0 \times 0,9) + (0,5 \times 0,3) = 0,9 + 0,15 = 1,05$$

Mit Hilfe der Aktivierungsfunktion

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

wird schließlich die Ausgabe dieses Knotens berechnet:

$$y_1 = \frac{1}{(1 + 0,3499)} = 0,7408$$

Die Berechnung der Ausgabe des zweiten Knotens der zweiten Schicht ergibt sich aus:

$$x_2 = (1,0 \times 0,2) + (0,5 \times 0,8) = 0,6$$

$$y_2 = \frac{1}{(1 + 0,5488)} = 0,6457$$

3.4 Relevante Aktivierungsfunktionen

Nun stellt sich die Frage, welche Funktionen zur Berücksichtigung des Schwellenwertes existieren und welchen Einsatzzweck diese bei der Verarbeitung von Signalen in künstlichen neuronalen Netzen haben. Im nächsten Abschnitt werden zunächst allgemein für das maschinelle Lernen wichtige Funktionen beschrieben, um dann auf für die vorliegende Arbeit relevante Funktion einzugehen.

Das Prinzip des Schwellenwertes wird bei künstlichen Neuronen mittels einer mathematischen Funktion berücksichtigt, der sogenannten Aktivierungsfunktion. Sie wird bei jedem Neuron des Netzes angewendet und bestimmt, ob dieses aktiviert werden soll oder nicht. Sie übernimmt ein Eingangssignal und erzeugt unter Berücksichtigung des Schwellenwertes ein Ausgangssignal. Die Aktivierungsfunktion entscheidet, ob der Eingang relevant für die Vorhersage des Modells ist oder nicht und ist dadurch ausschlaggebend für das Ausgangsverhalten jedes Neurons.

Die einfachste Aktivierungsfunktion ist die Stufen-/Sprungfunktion bzw. Heaviside-Funktion (siehe Abbildung 3-7). Ist der Eingangswert über oder unter eines bestimmten Schwellenwertes wird das Neuron aktiviert und sendet exakt das gleiche Signal zur nächsten Schicht. Diese Funktion erlaubt jedoch keine Mehrfachwerte. Sie findet meist in einfachen neuronalen Netzen mit nur einer Schicht Anwendung. Diese Art von Netzwerk kann linear trennbare Probleme klassifizieren, wie z. B. ein AND- oder ein OR-Gatter.

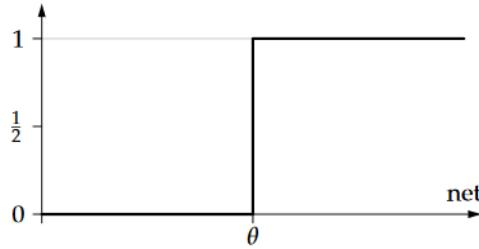


Abbildung 3-7: Aktivierungsfunktionen

Nichtlineare Aktivierungsfunktionen hingegen ermöglichen dem Modell eine komplexe Abbildung der Eingangsdaten auf die Ausgänge. Eine häufig für überwachtes Lernen verwendete, nichtlineare Funktion ist die Sigmoidale bzw. logistische Funktion, sowie die Tangens Hyperbolicus (siehe Abbildung 3-8). Der Wertebereich der Sigmoidfunktion liegt zwischen 0 und 1. Die Tangens Hyperbolicus folgt einer ähnlichen Form jedoch im Wertebereich zwischen -1 und 1. Durch beide Funktionen können Wahrscheinlichkeiten ausgedrückt werden. Sie werden zur Lösung von nichtlinearen Problemen mit einer Mehrfachklassifizierung genutzt.

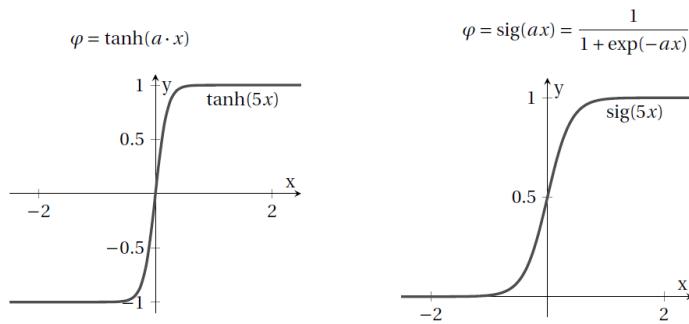


Abbildung 3-8: Sigmoid und Tangens Hyperbolicus als Aktivierungsfunktion (Frochte 2019, S. 171)

ReLU (Rectified Linear Unit)

Die ReLU-Funktion ist eine nichtlineare Aktivierungsfunktion und wird wie folgt dargestellt:

$$f(x) = \max(0, x) \rightarrow x \text{ als Eingabewert}$$

Sie beschreibt den maximalen Wert zwischen Null und einem Eingabewert x (siehe Abbildung 3-9). Ist der Eingabewert negativ, gibt die Funktion den Wert Null aus. Bei positiven Eingabewerten entspricht der Ausgabewert dem Eingabewert. Anders ausgedrückt:

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$$

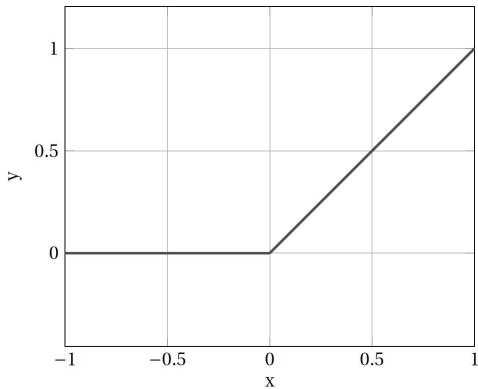


Abbildung 3-9: Aktivierungsfunktion ReLU (Frochte 2019, S. 211)

Einsatz findet die ReLU-Aktivierungsfunktion in Deep Neural Networks, da hiefür keine aufwändigen Berechnungen von Fehlertermen während der Trainingsphase notwendig sind. Zudem löst sie nicht das sogenannte „vanishing gradient problem“¹ aus, wenn mehrere Schichten im Netzwerk verwendet werden. Dies ermöglicht der ersten versteckten Schicht, die von der letzten Schicht kommenden Fehler zu empfangen und alle Gewichte zwischen den Schichten anzupassen, da sie keine asymptotische Ober- und Untergrenze hat. Andere Aktivierungsfunktionen, wie z. B. die Sigmoid, haben einen Wertebereich zwischen 0 und 1. Dadurch werden bei der Fehlerrückrechnung die Fehler der ersten Schicht verschwindend klein und verändern die Gewichte des Netzes kaum, was zu einem schlecht ausgebildeten Netzwerk führt. Auf dieses Problem wird an dieser Stelle jedoch nicht genauer eingegangen (deepai.org 2019).

¹ Je mehr Schichten zu neuronalen Netzen hinzugefügt werden, desto stärker nähern sich die Gradienten der Verlustfunktion Null an, was das Netz schwer trainierbar macht.

4 Maschinelles Lernen

Um ein neuronales Netz für die Abbildung einer Steuerungsstrategie trainieren zu können, sollen in diesem Kapitel zunächst die allgemeinen Verfahren des maschinellen Lernens vorgestellt und anhand der Anwendungsgebiete differenziert werden. Anschließend wird das für diese Arbeit relevante Verfahren des bestärkenden Lernens vorgestellt. Dabei soll speziell auf das Q-Lernen in Verbindung mit neuronalen Netzen eingegangen werden und ausgewählte und angewandte Elemente dieses Lernalgorithmus dargelegt werden.

4.1 Lernverfahren

Beim maschinellen Lernen unterscheidet man allgemein drei Lernmethoden, die jeweils für andere Zwecke geeignet sind und eine andere Informationsbasis zur Verfügung haben. Üblicherweise werden sie in folgende Felder unterteilt:

- Überwachtes Lernen (*Supervised Learning*)
- Unüberwachtes Lernen (*Unsupervised Learning*)
- Bestärkendes Lernen (*Reinforcement Learning*)

Diese Methoden dienen jeweils zur Lösung unterschiedlicher Probleme, haben jedoch alle das Ziel, die Funktion $f = x \rightarrow y$ zu erlernen. Mit Hilfe dieser Funktion soll aus gegebenen Inputdaten x ein passender Output y abgebildet werden (Inga Döbel et Al 2018, S. 10).

4.1.1 Überwachtes Lernen

Mit Hilfe von überwachtem Lernen, auch *Supervised Learning*, wird die Frage beantwortet, wie man automatisch eine Funktion aus einer Menge von Beispieldaten aufbaut, die einen Input auf einen Output abbildet. Überwacht bedeutet in diesem Zusammenhang, dass zu jedem Trainingsbeispiel bereits die richtige Antwort vorliegt (Lapan 2020, S. 1). Hierfür muss eine genügend große Datenmenge von Ein- und Ausgaben vorhanden sein, die bereits den richtigen Funktionswert enthalten. Solche Datensätze werden als *gelabelte* Datensätze bezeichnet. Ein typisches Beispiel hierfür ist ein Bilddatensatz von Hunden und Katzen, wofür zu jedem Bild die Information hinterlegt ist, ob darauf ein Hund oder eine Katze abgebildet ist. Anhand eines solchen Datensatzes wird ein Computer darauf trainiert, Hunde und Katzen zu unterscheiden. Sobald neue Datensätze ohne ein bekanntes Label hinzukommen, soll der Computer diese selbstständig richtig einordnen. Überwachtes Training dient meistens der Klassifikation oder der Regression. (Frochte 2019, S. 20–21)

Klassifikation

Bei der Klassifizierung soll die Abbildung der Eingangsgröße x auf die Ausgangsgröße $y \in \{1, \dots, C\}$ erlernt werden, wobei C die Anzahl der Klassen angibt (Murphy 2012, S. 3). Dabei

steht bereits im Vorfeld fest, in welche Gruppen ein Objekt eingeordnet werden kann. Es geht darum, Merkmale aus den Daten herauszufinden, die für die Zuordnung der Objekte am signifikantesten sind (Petzka et al. 2018, S. 27). Eine Möglichkeit, dieses Problem zu formalisieren, ist eine Näherungsfunktion $y = f(x)$. Das Ziel ist, die Funktion f anhand eines „gelabelten“ Trainingsdatensatzes zu schätzen, um damit Vorhersagen bei unbekannten, neuen Eingangsgrößen durch $\hat{y} = \hat{f}(x)$ treffen zu können. Dies wird auch als Generalisierung bezeichnet. (Murphy 2012, S. 3)

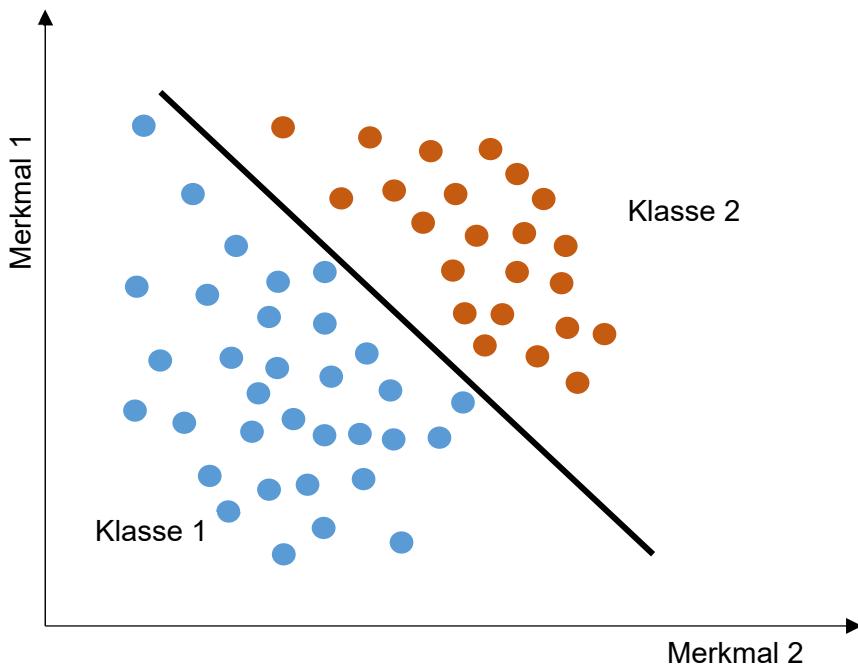


Abbildung 4-1: Lineare Klassifikation von Daten (Eigene Darstellung)

Regression

Regression unterscheidet sich von der Klassifizierung durch eine kontinuierliche Reaktionsvariable. Anstatt einer diskreten Klasse, z. B. $C = \{0,1\}$, erhält man bei der Regression einen Zahlenwert im kontinuierlichen Bereich als Vorhersagewert.

Typische Beispiele von Regressionsproblemen sind:

- Vorhersagen des Aktienmarktkurses
- Vorhersage des Alters von Zuschauern von Online-Videos
- Vorhersage von Temperaturen innerhalb eines Gebäudes anhand von Wetterdaten und Sensoren

Das einfachste Beispiel der Regression ist die lineare Regression. Es wird eine Zielgröße y auf Basis einer Eingangsvariablen x vorhergesagt, wenn eine lineare Abhängigkeit zwischen Zielgröße und Vorhersagewert besteht. Durch maschinelles Lernen kann eine Funktion

$$y = a + bx$$

approximiert werden, die eine Gerade durch die Datenpunkte abgebildet, wie in Abbildung 4-2 zu erkennen ist.

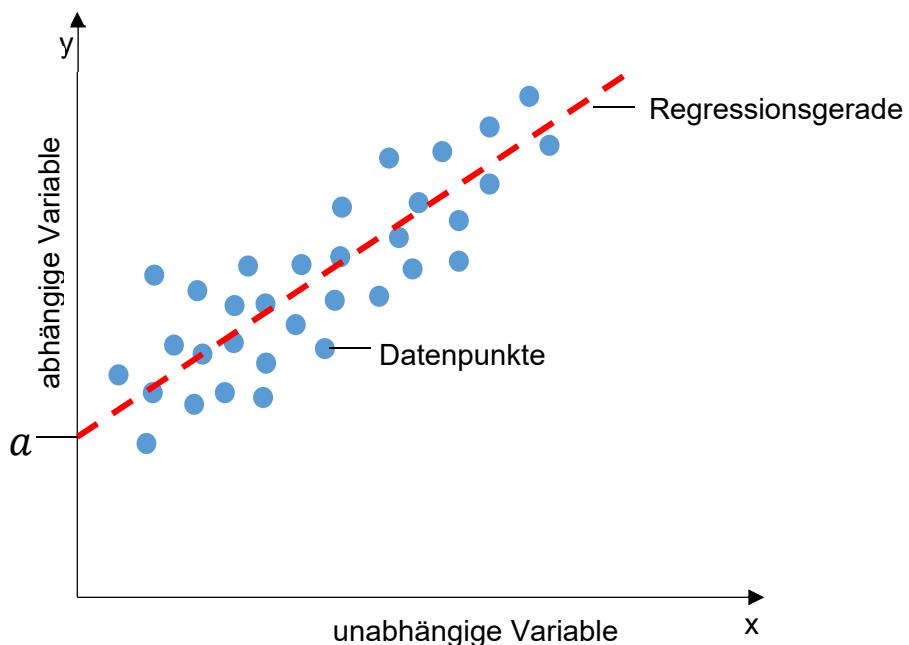


Abbildung 4-2: Lineare Regression (Eigene Darstellung)

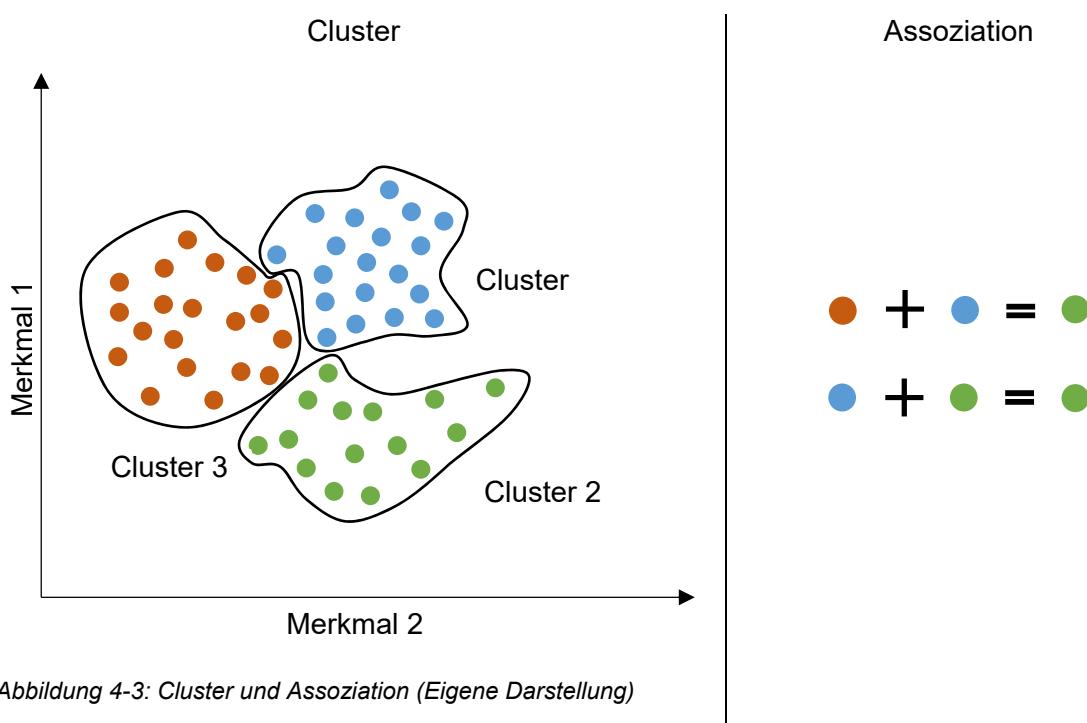
4.1.2 Unüberwachtes Lernen

Ziel des unüberwachten Lernens (*Unsupervised Learning*) ist es, Muster in der Datenmenge X_i zu erkennen, die nicht allein durch ein gleichmäßig auf den Raum verteiltes, zufälliges Rauschen entstanden sind. Damit werden insbesondere Strukturen von unklassifizierten Daten beschrieben und analysiert, um Wissen daraus zu gewinnen. Eine klassische Anwendung ist Data Mining zur Identifizierung von Abhängigkeiten in großen Datensätzen (Görz et al. 2013, S. 405–406).

Die Algorithmen zum Auffinden solcher Strukturen stellen eine Modellannahme an die Verteilung von X oder deren Eigenschaften. Dann wird statistisch geprüft, wie gut der Algorithmus die Werte schätzt, welche auch in der Modellannahme vorkommen. Die häufigsten Techniken beim unüberwachten Lernen sind Clustering und Assoziation.

Clustering und Assoziation

Beim Clustering werden Gruppen von ähnlichen Daten gefunden. Dabei steht noch gar nicht fest, welche Merkmale genau diese Ähnlichkeiten und Unterschiede ausmachen. In einer Menge von E-Mails können sich zum Beispiel zwei Cluster herausbilden, die ein Experte anschließend als „Spam“ und „Wichtig“ erkennt. (Petzka et al. 2018, S. 27). Bei der Assoziation werden Regeln und Muster aus den Datensätzen extrahiert, die Zusammenhänge zwischen den Variablen aufzeigen. Beispielsweise werden bei großen Online-Händlern die Inhalte der Warenkörbe analysiert und den Kunden passende Produkte vorgeschlagen (Schwaiger und Steinwendner 2019, S. 320).



4.2 Reinforcement Learning

Eine Möglichkeit, einen Prozess zu implementieren, der die Leistung eines Systems mit der Zeit durch maschinelles Lernen verbessert, ist das Reinforcement Learning (RL). Reinforcement Learning unterscheidet sich von den zuvor beschriebenen Verfahren des maschinellen Lernens dahingehend, dass keine Trainingsdatensätze betrachtet werden, sondern Daten durch Interaktion mit der Umgebung erzeugt werden. Um daraus lernen zu können und entsprechend zu handeln, werden viele Informationen und Parameter aus der Umgebung benötigt. Dabei handelt es sich um Szenarien aus der realen Welt, die durch 2D- oder 3D-Simulations- bzw. Spieleszenarien abgebildet werden (Nandy und Biswas 2018, S. 11).

Das lernende Element beim Reinforcement Learning, der Agent, interagiert mit seiner Umwelt (*environment*), in der er sich befindet. Anhand unterschiedlicher Sensoren kann er hinsichtlich seines Zustandes innerhalb der Umwelt eine Entscheidung treffen und eine bestimmte Aktion ausführen, die wiederum den Zustand der Umgebung verändert. Für jede Aktion erhält der Agent eine Rückmeldung der Umwelt und wird für diese belohnt oder bestraft (Alpaydin 2019, S. 539).

4.2.1 Markov-Entscheidungsprozess

Das folgende Diagramm zeigt die zwei elementaren RL-Elemente – Agent und Umgebung – und ihre Kommunikationskanäle – *actions*, *rewards* und *observations*.

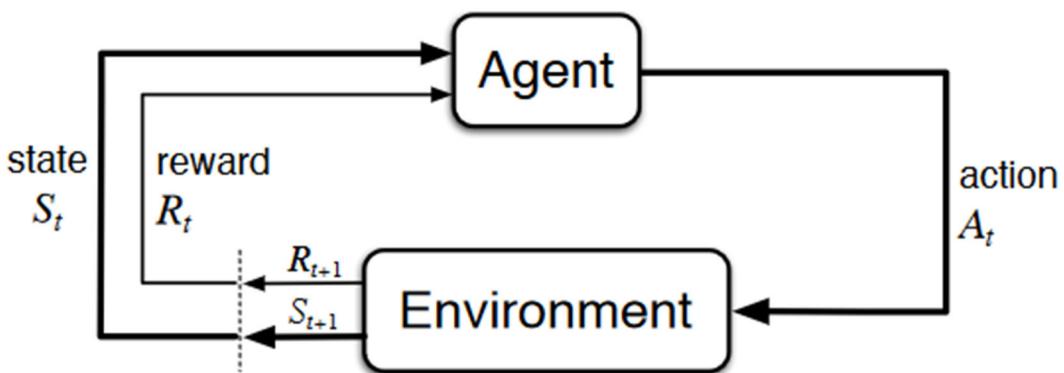


Abbildung 4-4: Interaktion zwischen Agent und Environment in einem Markov-Entscheidungsprozess (Sutton und Barto 2018, S. 48)

Ein Agent im Kontext des Reinforcement Learning ist ein Computerprogramm, welches ein Problem auf mehr oder weniger effiziente Weise lösen soll. Es interagiert mit der Umwelt, macht Beobachtungen und trifft eigenständig bestimmte Aktionen, wofür es eventuell Belohnungen erhält – ohne Eingriff von außen (Lapan 2020, S. 1). Der Agent und die Umgebung (*environment*) interagieren jeweils in diskreten Zeitschritten ($t = 0, 1, 2, \dots$). Bei jedem Zeitschritt erhält der Agent eine Abbildung des Zustands (*state*) $S_t \in S$ der Umgebung und wählt auf dieser Grundlage eine Aktion $A_t \in A(s_t)$ aus. Nachdem der Agent eine Aktion ausgeführt hat, erhält er einen numerischen Wert, die Belohnung (*reward*) $R_{t+1} \in R$, und geht in den nächsten Zustand S_{t+1} über (Sutton und Barto 2018, S. 48). Daraus entsteht eine Sequenz, die wie folgt aussieht:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots$$

4.2.2 Komponenten eines Reinforcement-Learning-Agenten

Im vorherigen Kapitel wurde bereits auf die grundlegen Funktionsweise eines Reinforcement-Learning-Agenten eingegangen. Nun werden die hierfür notwendigen grundlegenden Komponenten näher beschrieben.

Der Reinforcement-Learning-Agent besteht im Allgemeinen aus den Elementen, Taktik (*policy*), Belohnungsfunktion (*reward-function*), Wertefunktion (*value-function*) und einer Umwelt (*environment*). Die Taktik ist die Entscheidungsfunktion des Agenten. Sie bildet den Kern des Agenten und bestimmt welche Maßnahmen er in einer Situation ergreift. Die anderen Komponenten dienen nur dazu, die Taktik zu ändern und zu verbessern. Die Taktik selbst ist der letztendliche Bestimmungsfaktor für Verhalten und die Leistung des Agenten (Sutton und Barto 2018, S. 6–7).

Umwelt

Die **Umwelt** (*Environment*) ist alles außerhalb eines Agenten. Sie bildet ein Modell der Umwelt ab, welches das Verhalten der Umwelt nachahmt bzw. Rückschlüsse darauf zulässt, wie z. B. ein Schachbrett oder eine Spieleumgebung eines Computerspiels bis hin zu Simulationsmodellen, die zur Planung von Produktionssystemen eingesetzt werden. Die Kommunikation des Agenten mit der Umgebung beschränkt sich auf die Belohnungen (die er von der Umwelt erhält), Aktionen (die vom Agenten ausgeführt und von der Umwelt vorgegeben werden) und Beobachtungen (parametrische Beschreibungen des Zustands der Umwelt) (Lapan 2020, S. 1).

Belohnung

Die **Belohnung** (*reward*) definiert das Ziel eines Reinforcement-Learning-Problems. Dabei handelt es sich lediglich um einen skalaren Wert, den der Agent periodisch nach jedem Zeitschritt aus der Umwelt erhält (Lapan 2020, S. 1). Dieser kann einen positiven oder negativen Wert annehmen, je nach gewählter Aktion in diesem Zustand. Der Agent hat das Ziel, die gesamte Belohnung, die er auf lange Sicht bekommt, zu maximieren. Durch die Belohnung kann der Agent eine gute Aktion von einer schlechten Aktion unterscheiden. Das bildet die Grundlage dafür, eine Taktik abzuändern. Wenn beispielsweise die Taktik eine Aktion wählt, der eine niedrige Belohnung folgt, kann diese angepasst werden, um in Zukunft in dieser Situation eine andere Aktion auszuwählen (Sutton und Barto 2018, S. 6). Die Vergabe der Belohnung kann jede Sekunde oder nur einmal im Leben des Agenten erfolgen. Im Falle von einmaligen Belohnungssystemen sind alle Belohnungen bis auf die letzte gleich null (Lapan 2020, S. 1). Im Allgemeinen können Belohnungssignale stochastische Funktionen des Zustands der Umwelt und der ergriffenen Maßnahmen sein (Sutton und Barto 2018, S. 6).

Wertefunktion (Value function)

Belohnungen (*rewards*) wirken unmittelbar und werden direkt von der Umwelt vergeben, während **Werte** (*values*) den möglichen Gewinn über eine Sequenz von Beobachtungen des Agenten von einem bestimmten Zustand aus ausdrücken. Die Auswahl der richtigen Aktionen wird anhand der höchsten Werte getroffen und nicht durch die höchste Belohnung, da diese Aktionen die größte Belohnung über einen langen Zeitraum versprechen. Daher spielt eine geeignete Methode zur Schätzung der Werte (*values*) beim Reinforcement Learning die zentrale Rolle. (Sutton und Barto 2018, S. 6)

Taktik (Policy)

Eine **Taktik** (*policy*) definiert die Art und Weise, wie sich der Lernende zu einem bestimmten Zeitpunkt verhält. Eine Taktik ist eine Abbildung des betrachteten Zustands (*state*) der Umwelt auf die Aktionen (*actions*), die gewählt werden, wenn man sich in diesen Zuständen befindet. Dabei kann es sich um eine Funktion $f(x)$ oder um einfache Nachschlagetabelle handeln. Die Taktik bildet den Kern eines Reinforcement-Learning-Agenten, da diese das Verhalten des Agenten bestimmt. Eine Taktik kann stochastisch sein und Wahrscheinlichkeiten für jede Aktion definieren (Sutton und Barto 2018, S. 6).

Die optimale Taktik

Hinsichtlich der Zeitdauer gibt es nach Alpaydin (Alpaydin 2019, S. 538–544) zwei Aufgabentypen für ein Agentensystem. Beim Modell mit einem finiten Horizont versucht der Agent für die erwartete Belohnung E , den Wert V der Taktik für alle Schritte im Zeitraum T , zu maximieren.

$$V^\pi(s_t) = E[r_{t+1} + r_{t+2} + \dots + r_{t+T}] = E \left[\sum_{i=1}^T r_{t+i} \right] \quad (4-1)$$

V^π ist der Wert im Zustand s_t wenn man der policy π folgt. E ist die erwartete Gesamtbelohnung, die über alle Schritte erreicht werden kann.

Hingegen gibt es beim Modell mit infinitem Horizont kein festgelegtes Limit für die Sequenz an Aktionen. Hierfür werden zukünftige Belohnungen diskontiert, wobei Gamma mit $0 \leq \gamma < 1$ die Skontorate ist. Dadurch wird im infiniten Horizont eine endliche Belohnung ermöglicht.

$$V^\pi(s_t) = E[r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \dots] = E \left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \right] \quad (4-2)$$

Für $\gamma = 0$, ist lediglich die sofortige Belohnung relevant. Je mehr sich Gamma 1 annähert, desto stärker werden Belohnungen in der Zukunft berücksichtigt. Der Agent wird dadurch weitsichtiger.

4.2.3 Lernende Multiagentensysteme

Im Kontext des Reinforcement Learning können Einzelagenten oder Multiagentensysteme Anwendung finden. Bei komplexen Problemen wird Reinforcement Learning mit Multiagenten angewendet. Der Unterschied zwischen Einzelagenten- und Multiagentenlösungen sind wie folgt: Ein Einzelagent ist eine intelligente Software, die nur in einer Umgebung und nicht mit mehreren parallelen Umgebungen interagieren kann. In Szenarien hingegen bei denen mehrere Umgebungen in simultanem Austausch stehen, werden Multiagentensystemen eingesetzt (Nandy und Biswas 2018, S. 16).

Die Agenten können in unterschiedlichen Umgebungen interagieren und verschiedenste Aufgabe und Ziele verfolgen. Multiagentensysteme sind nicht-deterministische Systeme, da es bei der Interaktion der Agenten eine sehr hohe Anzahl an möglichen Aktionen oder Zustandsübergänge gibt, auf deren Basis eine Entscheidung getroffen werden muss.

Das Lernen in einer Umgebung mit mehreren Agenten gestaltet sich wesentlich komplexer als mit einzelnen Agenten, da diese gleichzeitig mit der Umwelt und eventuell untereinander interagieren. Dies verursacht häufig Komplikationen bei der Bestimmung von Zustandsübergängen. Der Ansatz des dezentralen Lernens nutzt Prinzipien und Algorithmen der Einzelagenten in einer Multi-Agenten Umgebung. Da die Umgebung jedoch nicht mehr stationär ist, werden vor allem die Markovschen Eigenschaften ungültig, wonach zukünftiges Verhalten von Übergängen und Belohnungen nur vom aktuellen Zustand der Umwelt abhängen (Hernandez-Leal et al. 2018, S. 10).

4.3 Auswahl eines geeigneten Lernverfahrens

4.3.1 Q-Learning

Für Anwendungen bei Steuerungssystemen, wie in der vorliegenden Arbeit, ist es weniger relevant den Wert $V(s_t)$ zu kennen, sich in einem bestimmten Zustand zu befinden (siehe Kapitel 4.2.2). Weitaus interessanter für ein Steuerungssystem ist es, den Wert eines Zustands-Aktions-Paares $Q(s_t, a_t)$ zu kennen. Dieses gibt an, wie vorteilhaft es ist, Aktion a_t im Zustand s_t auszuführen. Dadurch kann die Taktik π direkt implizit mitgelernt werden. Mit den Q-Werten werden mehrere Werte pro State gespeichert, jeweils einer je ausführbarer Aktion (Alpaydin 2019, S. 540).

Nach Alpaydin (Alpaydin 2019, S. 538–544) kann $Q^*(s_t, a_t)$ als die erwartete kumulative Belohnung für die Aktion a_t definiert werden, wenn die optimale Taktik π^* im Zustand s_t befolgt

wird. Dies kann als Formel mit der sogenannten Bellmanschen Gleichung ausgedrückt werden.

Der optimale Wert eines Zustandes V^* ist gleich dem Wert der bestmöglichen Aktion.

$$V^*(s_t) = \max_{a_t} Q^*(s_t, a_t) \quad (4-3)$$

Durch Umformen und Einsetzen der Formel 4-2 erhält man die Bellmansche Gleichung:

$$V^*(s_t) = \max_{a_t} (E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1})) \quad (4-4)$$

Vom Zustand s_t gelangt man in den Folgezustand s_{t+1} mit der Wahrscheinlichkeit $P(s_{t+1}|s_t, a_t)$. Geht man von dort aus weiter unter Befolgung der optimalen Taktik π^* , erhält man die erwartete kumulative Belohnung $V^*(s_{t+1})$ des Folgezustands. Es werden anschließend alle möglichen Folgezustände addiert und mit Gamma diskontiert. Addiert man nun noch die direkte Belohnung, erhält man die gesamte erwartete Belohnung für Aktion a_t .

Ähnlich kann die Bellman-Gleichung als Wertefunktion (Q-Funktion bzw. Aktions-Werte-Funktion) ausgedrückt werden, wobei Q^* die optimale Wertefunktion ist:

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \quad (4-5)$$

Sind die optimalen $Q^*(s_t, a_t)$ -Werte bekannt, kann durch Ausführen der Taktik π die Aktion a_t^* mit dem höchsten Wert aus $Q^*(s_t, a_t)$ gewählt werden. Dadurch ergibt sich mit Anwendung einer Greedy-Suche bei jedem Schritt eine optimale Sequenz, die die Belohnung maximiert.

Lernen mit temporaler Differenz

Bei der Anwendung von Reinforcement Learning nimmt die Methode des *Temporal-Difference-Learning* (temporale Differenz) eine zentrale Rolle ein. Diese Methode kann direkt von Erfahrungen lernen, ohne ein Modell der Umgebung zu haben. Das Lernen mit temporaler Differenz aktualisiert Werte, zum Teil basierend auf anderen gelernten Werten ohne auf das Endergebnis warten zu müssen (Patrick Dammann, S. 6). Relevant für das Lernen mit temporaler Differenz ist der Unterschied zwischen der momentanen Schätzung des Wertes eines Zustands bzw. Zustand-Aktions-Paares und dem diskontierten Wert des Folgezustands und der erhaltenen Belohnung (Alpaydin 2019, S. 543).

Alpaydin (Alpaydin 2019, S. 544–545) unterscheidet dabei deterministische und nicht-deterministische Belohnungen und Aktionen. Bei dem deterministischen Fall gibt es zu jedem Zustands-Aktions-Paar eine Belohnung sowie einen möglichen Folgezustand. In diesem Fall

vereinfacht sich Gleichung 4-5 und man erhält die Bellmann-Gleichung für die optimale Q-Funktion Q^* zur Aktualisierung der Werte:

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (4-6)$$

$$Q^*(s, a) = E \left[r_{t+1} + \gamma \max_{a' \in A} Q^*(s', a') \right] \quad (4-7)$$

Handelt es sich nun bei den Belohnungen und dem Ergebnis der Aktionen um nicht-deterministische, besteht eine Wahrscheinlichkeitsverteilung für die Belohnungen und für den Folgezustand. Dadurch kann die Ungewissheit im System modelliert werden, die von der Umwelt ausgeht. Man bekommt für denselben Zustand und dieselbe Aktion unter Umständen verschiedene Belohnungen bzw. geht in unterschiedliche Folgezustände über. Hier wird dann Gleichung 4-5 angewendet.

Das Ziel der vorliegenden Arbeit ist die Implementierung von Reinforcement Learning in eine diskrete, ereignisorientierte Simulation. Hier muss ein Lernverfahren angewendet werden, welches ohne Modell der Zustandsübergänge lernen kann und dem deterministischen Ansatz zur Aktualisierung der Q-Werte folgt.

4.3.2 Deep Q-Learning

Für die Umsetzung von Reinforcement Learning in der Materialflusssimulation von fiktiven oder realitätsnahen Produktionssystemen, deren Zustandsmenge weitaus größer ist als bei einfachen Modellen, wird eine einfach zu trainierende Funktionsapproximation notwendig, um die Q-Werte effizient und sicher zu lernen. Bei solch großen Datensätzen von Umgebungszuständen kann nicht mehr mit Nachschlagtabellen gearbeitet werden. Deshalb bedient man sich bei komplexen Umgebungen neuronaler Netze als Wissensrepräsentation. Mit neuronalen Netzen können beliebige Funktionen approximiert und gelernt werden und die Netze erleichtern die Verarbeitung der Daten. So kann dem Agenten der Zustand der Umgebung in beliebigen Formen übergeben werden. Das neuronale Netz bekommt beim Deep-Q-Learning die Informationen der Umgebung als Eingangsgröße und gibt als Ausgangsgröße einen Vektor mit Q-Werten aller möglicher Aktionen an, die der Agent ausführen kann. Dies wird auch als Deep Q-Network (DQN) bezeichnet.

4.3.3 Double Q-Learning

Eine wichtige Ergänzung des DQN ist die Nutzung eines zweiten Netzwerks während des Trainings. Diese Technik wurde zuerst in einem Konferenzartikel der „International Conference on Learning Representations“ (ICLR) 2016 veröffentlicht. Dort wird ein zweites Netzwerk

verwendet, um die Ziel-Q-Werte zu generieren, die zur Berechnung des Verlusts für jede Aktion während des Trainings verwendet werden. Das Problem ist, dass sich bei jedem Schritt des Trainings die Werte des DQ-Netzwerks verschieben. Wenn ein sich ständig verschiebender Satz von Werten zur Anpassung der Netzwerkwerte verwendet wird, dann können die Wertschätzungen leicht außer Kontrolle geraten. Das Netzwerk kann destabilisiert werden, da es in Rückkopplungsschleifen zwischen den Ziel- und den geschätzten Q-Werten gerät. Um dieses Risiko zu mindern, werden die Gewichte des Zielnetzwerks fixiert und nur periodisch oder langsam auf die primären Q-Netzwerkswerte aktualisiert. Auf diese Weise kann das Training auf eine stabilere Weise ablaufen (Lillicrap et al. 2015, S. 1–4), (Arthur Juliani 2016).

Der Max-Operator im Standard Q-Learning (Kapitel 4.3.1) verwendet die gleichen Werte sowohl zur Auswahl als auch zur Bewertung einer Aktion. Dadurch wird es laut van Hasselt wahrscheinlicher, dass der Max-Operator zu hoch geschätzte Werte auswählt, was zu überoptimistischen Wertschätzungen führt. Mit der Idee des Double-Q-Learning wird versucht die Auswahl einer Aktion von der Bewertung einer Aktion zu entkoppeln (van Hasselt et al. 2015, S. 2).

Der ursprüngliche Doppel-Q-Lernalgorithmus lernt zwei Wertfunktionen, indem jede Erfahrung einer der beiden Wertfunktionen zufällig zugeordnet wird um diese zu aktualisieren, so dass es zwei Sätze von Gewichtungen gibt. Van Hasselt erweitert den Ansatz und nutzt zur Bestimmung der *greedy*-Taktik das Online-Netzwerk und zur Vorhersage der Werte das Zielnetzwerk.

Weiterhin werden die Gewichte des Zielnetzes durch die Gewichte des Online-Netzes ersetzt um die *greedy*-Taktik zu bewerten. Die Aktualisierung des Zielnetzwerks bleibt dem gegenüber während des Lernens unverändert und bleibt so eine periodische Kopie des Online-Netzwerks.

Als Ziel für das neuronale Netz wird ein sogenanntes Target-Netz eingeführt. Dies zeigt dem trainierenden Agenten das gewünschte Ergebnis und definiert sich wie folgt:

$$r + \gamma \max_{a' \in A} Q(s', a') \quad (4-8)$$

Dadurch sammelt der Agent mit einer fixierten Strategie Erfahrungen während einer Episode und trainiert nach Abschluss der Episode das Online-Netz.

Dieser Ansatz findet auch in der vorliegenden Arbeit bei der Implementierung des Lernprozesses Anwendung. Das Zielnetz wird über einen gesamten Simulationslauf konstant

gehalten und jeweils an dessen Ende durch das primäre Netzwerk aktualisiert. Dies bedeutet die Ziel-Q-Werte, also die Q-Werte für die Vorhersage der Folgezustände, bleiben fixiert.

4.3.4 Erfahrungswiedergabe (*Experience Replay*)

Agenten, die online mit Reinforcement Learning (RL) lernen, aktualisieren inkrementell ihre Parameter, wie z. B. die Q-Wertefunktion, während sie einen Erfahrungsschritt beobachten. In ihrer einfachsten Form verwerfen sie eingehende Daten sofort nach einer einzigen Aktualisierung. Daraus ergeben sich zwei Probleme: Stark korrelierende Aktualisierungen der Werte treten auf und seltene Erfahrungen werden schnell vergessen, die später evtl. nützlich sein könnten (Tom Schaul et al. 2015, S. 1).

Daher ist auch für den Lernalgorithmus der vorliegenden Arbeit eine weitere wichtige Ergänzung notwendig, die das Funktionieren von DQNs ermöglicht. Die sogenannte Erfahrungswiedergabe oder auch *Experience Replay*. Der Konferenzartikel „*Prioritized Experience Replay*“ (Tom Schaul et Al.) beschreibt dies. Das Netzwerk wird durch das Speichern der Erfahrungen eines Agenten und dem anschließenden Ziehen von zufälligen Losen daraus trainiert. Durch die Erfahrungswiederholung können zeitliche Korrelationen ausgeglichen werden, indem aktuelle Erfahrungen mit älteren für die Aktualisierung gemischt werden. Seltene Erfahrungen fließen so zudem in mehr als nur eine einzelne Aktualisierung mit ein (Tom Schaul et al. 2015, S. 1). So wird verhindert, dass das Netzwerk nur das lernt, was es unmittelbar in der Umwelt erfährt. Jede dieser Erfahrungen wird als ein Tupel aus Zustand, Aktion, Belohnung, Folgezustand gespeichert. Dieser Puffer an Erfahrungen hält eine feste Anzahl der neuesten Erinnerungen fest. Kommen neue hinzu, werden alte entfernt. Sobald das Training beginnt, wird ein zufälliger Auszug von Tupel aus dem Puffer gezogen und das Netzwerk damit trainiert (Arthur Juliani 2016).

4.3.5 Exploration vs. Exploitation

Beim Reinforcement Learning ist eine weitere Herausforderung, den Kompromiss zwischen der Anwendung der gelernten Erfahrung und der Erkundung neuer Aktionen zu finden. Sutton (Sutton und Barto 2018, S. 3) beschreibt dies als Dilemma zwischen Exploration und Exploitation. Der Agent wählt diejenige Aktion aus, die sich in vergangen Spielzügen als gewinnbringendste herausgestellt hat. Um solche Aktionen zu entdecken, muss er auch diejenigen ausprobieren, die er zuvor noch nie versucht hat. Der Agent muss zum einen das bereits Gelernte nutzen, um die Belohnung zu maximieren, jedoch zum anderen auch neue Aktionen erforschen, um in Zukunft eine bessere Auswahl der Aktionen treffen zu können.

Als Strategie, dieses Dilemma zu lösen, wird in der vorliegenden Arbeit die ε – *greedy* – *Suche* angewendet. Dabei wirft der Agent vor jeder Aktion eine unfaire Münze, so dass er mit einer Chance von $1 - \varepsilon$ nach seiner Strategie (policy) handelt, was als Exploitieren bezeichnet wird.

Mit einer Wahrscheinlichkeit von ε wird eine völlig zufällige Aktion ausgewählt. Dies wird als Exploration bezeichnet (Alpaydin 2019, S. 543). Während ε zur Test-Zeit für gewöhnlich auf 0 gesetzt wird, kann es zur Trainingszeit je nach Bedarf konstant sein, mit der Zeit abnehmen oder anderweitig an die Gegebenheiten angepasst werden (Patrick Dammann, S. 8).

5 Konzept zur Implementierung eines Reinforcement-Agenten in einer diskreten Ablaufsimulation

Für die Entwicklung des Ansatzes, in der Simulation mit maschinellem Lernen Materialflüsse zu steuern, wird zunächst ein Konzept als Leitfaden für eine praktische Umsetzung eines Prototyps entwickelt. Hierfür wird im ersten Schritt das grundlegende Zusammenspiel der notwendigen Komponenten erarbeitet und festgelegt. Anschließend werden die Prozesse und die Abläufe innerhalb dieser Komponenten skizziert und deren Datenaustausch zur Kommunikation untereinander dargelegt. Zum Schluss wird eine softwaretechnische Entwicklungsumgebung festgelegt, um eine prototypische Umsetzung des Konzeptes zu realisieren.

5.1 Interaktion zwischen Agent und Umwelt

Die Hauptelemente des Agentenkonzeptes dieser Arbeit bestehen, wie in Kapitel 4.2.1 beschrieben, aus dem lernenden Agenten selbst, sowie der Umwelt mit welcher der Agent interagiert. In Anlehnung an dieses Modell wurde auch das Konzept einer agentenbasierten Steuerung von Materialflüssen aufgebaut. Dabei bildet ein Modell eines Produktionssystems in einer Simulationssoftware die Umgebung bzw. Umwelt des Agenten. Dieses beinhaltet einen oder mehrere Simulationsagenten, die die Steuerung des Materialflusses innerhalb des Modells übernehmen. Sie sollen beispielsweise die Bestimmung von Reihenfolgen durch herkömmliche Methoden, wie z. B. einfache lokale Regeln oder Heuristiken, ersetzen. Neben dem Simulationsagenten wird ein Element des Agenten benötigt, das die Steuerungsintelligenz abbildet. Dieser Teil wird als lernender Agent bezeichnet. Darin bildet ein neuronales Netz das zentrale Wissenselement zur Steuerung des Materialflusses. Dessen Gewichte werden durch Interaktion der beiden Agententeile und über Methoden des maschinellen Lernens angepasst.

Im Rahmen dieser Interaktion übergibt der lernende Agent dem Simulationsagenten die Aktion A_t . Dieser führt diese Aktion innerhalb des Simulationsmodells aus und geht in einen neuen Zustand S_{t+1} über. Je nachdem wie gut diese Aktion war, wird hierfür ein Belohnungs- bzw. Bestrafungswert vom Simulationsagenten berechnet. Der lernende Agent erhält den neuen Zustand S_{t+1} und die Belohnung R_{t+1} . Auf Basis dieser Informationen wird die nächste Aktion vorhergesagt und wieder an den Simulationsagenten zurückgegeben. Abbildung 5-1: Interaktion zwischen Simulationsagent und lernendem Agenten verdeutlicht dies grafisch.

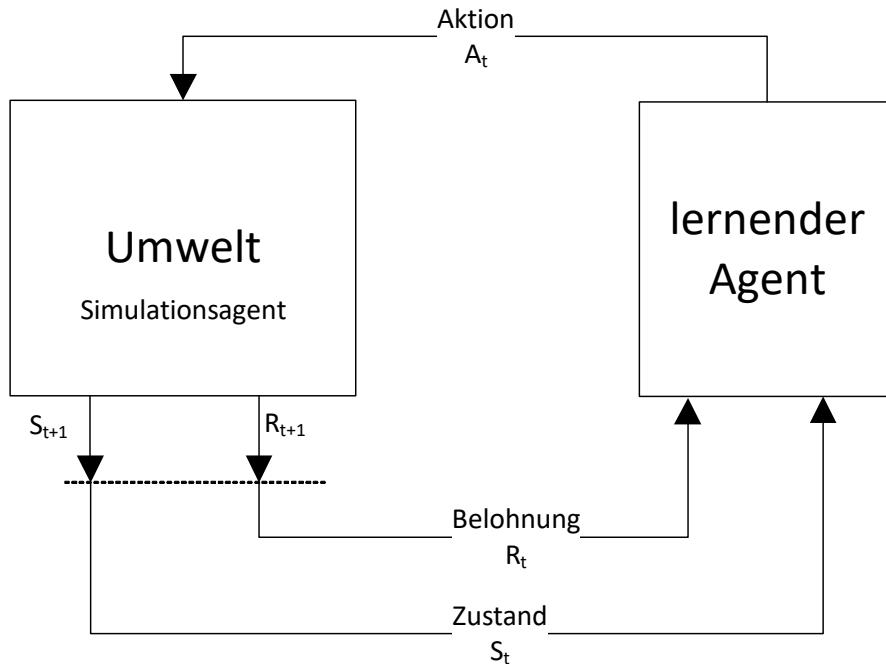


Abbildung 5-1: Interaktion zwischen Simulationsagent und lernendem Agenten

Das Zusammenspiel dieser beiden Softwaresysteme wird im Folgenden nun konkretisiert und anhand des Prozessdiagramms in Abbildung 5-2 beschrieben. Aus der Simulationsumgebung heraus wird der lernende Agent gestartet und initialisiert. Dieser wartet auf eine Antwort der Simulationssoftware mit der Übermittlung der zu erledigenden Aufgabe. Nun kann der Prozess zum Anlernen des Agenten mit verschiedenen Experimenten vom Simulant gestartet werden. Nachdem der Anfangszustand des Simulationsmodells initialisiert wurde, beginnt der Agent durch Ausführen einer Schrittfunktion *mStep* mit der Umwelt zu interagieren. Sie fordert beim lernenden Agenten eine Aktion an und führt diese entsprechend aus. Dafür wird ein Punktewert anhand der Belohnungsfunktion berechnet und zusammen mit dem aktuellen Zustand der Umwelt und deren Folgezustand an den lernenden Agenten übertragen. Dieser startet daraufhin einen Trainingslauf und speichert die Daten in einem Datenspeicher. Ist die Aktion nicht durchführbar oder sind keine Aktionen des Agenten mehr notwendig, wird die Schrittfunktion abgebrochen und es startet ein neuer Experimentlauf. Sobald alle Experimente durchgeführt wurden, wird das Training beendet.

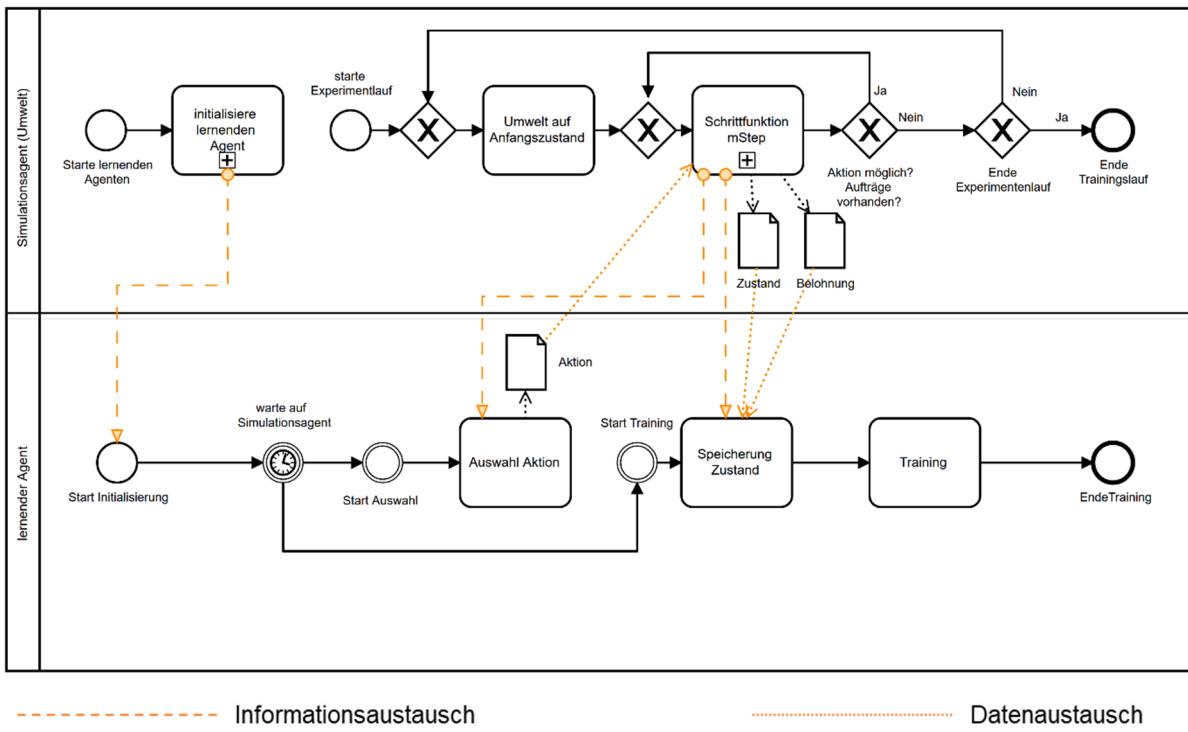


Abbildung 5-2: Prozess der Interaktion zwischen Simulationsagent und lernendem Agent

5.2 Ablaufprozess des Simulationsagenten

Anhand der Allgemeinen Interaktion eines lernenden Agenten und seiner Umwelt, wie in Kapitel 4.2.1 beschrieben, wird folgend das Konzept zum Aufbau und Ablauf des Simulationsagenten innerhalb einer Simulationsumgebung entwickelt. Der Prozess greift während des Trainings direkt in den Simulationslauf ein und bestimmt daher auch die Simulationsgeschwindigkeit. Die Simulationssoftware bildet die Zentrale zur Steuerung des Trainings. Eine Benutzereingabe initialisiert den lernenden Agenten mit den benötigten Hyperparametern. Ist der lernende Agent aktiviert, kann ein Experimentlauf gestartet werden. Zu Beginn jedes Laufes setzt der Experimentverwalter den Zustand der Umwelt und des Simulationsagenten auf einen Ausgangszustand zurück. Dieser wird durch die zuvor definierten Experimente definiert. Nach dem Start wird die zentrale Schrittfunktion *mStep* des Simulationsagenten ausgelöst. Diese Funktion regelt die Interaktion zwischen Umwelt und dem lernenden Agenten. Sie speichert die Zustände der Umwelt, führt die vom lernenden Agenten vorhergesagten Aktionen durch, berechnet die hierfür erhaltenen Belohnungen und übermittelt diese an den lernenden Agenten.

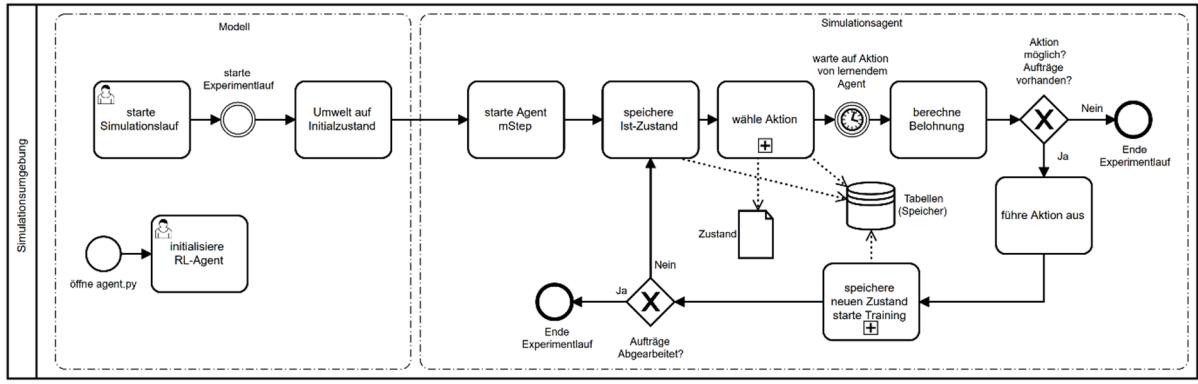


Abbildung 5-3: Konzept der Simulationsumgebung und des Simulationsagenten

Zunächst muss der aktuelle Zustand der Umgebung in einen Datenspeicher, z. B. in Form von Tabellen, innerhalb der Simulationssoftware gespeichert werden. Nun kann eine Anfrage zur Auswahl einer Aktion an den lernenden Agenten gesendet werden. Findet eine Vorhersage durch das neuronale Netz statt, benötigt der lernende Agent zudem den aktuellen Zustand der Umwelt. Anschließend muss der Simulationslauf angehalten werden, bis der lernende Agent eine Aktion rückmeldet. Die gewählte Aktion wird ebenfalls in den Speicher aufgenommen. Eine weitere Methode berechnet nun für die gewählte Aktion die Belohnung bzw. Bestrafung. Kann die Aktion nicht ausgeführt werden, führt dies zu einem Abbruch des Simulationslaufs. Ist sie hingegen möglich, führt der Simulationsagent die Aktion entsprechend durch. Nun kann ein Trainingslauf innerhalb des lernenden Agenten gestartet werden. Hierfür wird der neue Zustand der Umwelt im Datenspeicher der Simulationssoftware abgelegt und anschließend alle notwendigen Daten zum lernenden Agenten übermittelt. Nach jedem Durchlauf der Schritt-funktion *mStep* wird geprüft, ob weitere Aktionen auszuführen sind. Sind noch Aktionen durchzuführen, findet eine Wiederholung des Ablaufs statt. Nachdem alle Aktionen erledigt sind, wird der Experimentlauf beendet und der nächste startet.

5.3 Der lernende Agent

5.3.1 Initialisierung des lernenden Agenten

Der lernende Agent soll von dessen Umwelt (Simulationsmodell) aus aufgerufen und gestartet werden. Dies kann über den Aufruf einer Batch-Datei realisiert werden. Diese Batch-Datei führt den lernenden Agenten aus und wartet auf den Empfang von Daten aus der Simulationsumgebung. Durch die Initialisierung werden die für den Lernprozess benötigten Hyperparameter übergeben. Anschließend kann auch das tiefe neuronale Netzwerk mit einem entsprechenden Framework erzeugt werden. Die auszuführenden Aktionen sind im Prozessgraph in Abbildung 5-4 dargestellt.

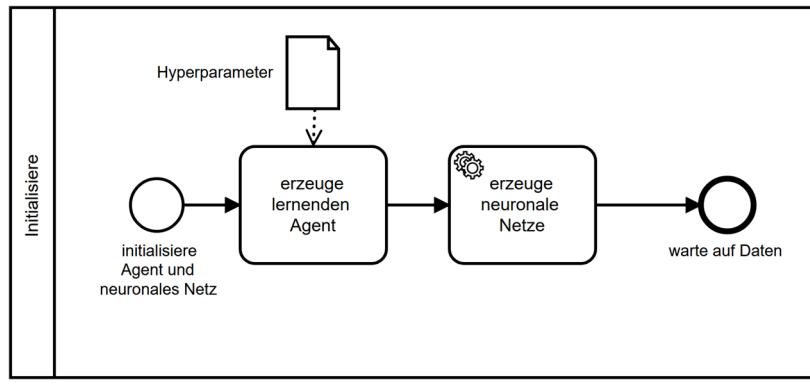


Abbildung 5-4: Initialisierungsprozess des Agenten

5.3.2 Auswahl von Aktionen

Die Auswahl bzw. Vorhersage einer Aktion muss der lernende Agent ausführen. Die Simulationssoftware stößt dies beim lernenden Agenten an und löst den Prozess in Abbildung 5-5 aus. Der Prozess der Aktionswahl kann auf unterschiedliche Art und Weise durchgeführt werden. Prinzipiell muss entschieden werden, ob der lernende Agent eine Aktion durch eine Netzafrage oder per Zufall auswählt. Beispielhaft wird dieser Vorgang anhand des Epsilon-Greedy-Verfahrens nun erläutert. Hierfür wird eine gleichverteilte Zufallszahl zwischen eins und null gewählt. Ist diese Zahl kleiner oder gleich dem Hyperparameter ε (Epsilon), wird eine zufällige Aktion generiert. Dies funktioniert auf gleiche Weise wie zuvor: Eine gleichverteilte Zufallszahl im Zahlenraum der Anzahl der Aktionen wird generiert und als Aktion an den Simulationsagenten übergeben. Ist die Zufallszahl größer ε , macht das neuronale Netz eine Vorhersage anhand des aktuellen Zustandes und übergibt diese Aktion. Epsilon wird dabei im Laufe des Trainings dynamisch angepasst. Dadurch kann ein Gleichgewicht zwischen Exploration und Nutzen (Exploitation) des Erlernten geschaffen werden. Es ermöglicht dem Agenten, neue Aktionen auszuprobieren, trotz der bereits gelernten Verhaltensweise. So kann verhindert werden, dass der Agent einseitige Aktionen lernt. Eine nähere Erläuterung dieses Verfahrens wurde in Kapitel 4.2.2 behandelt.

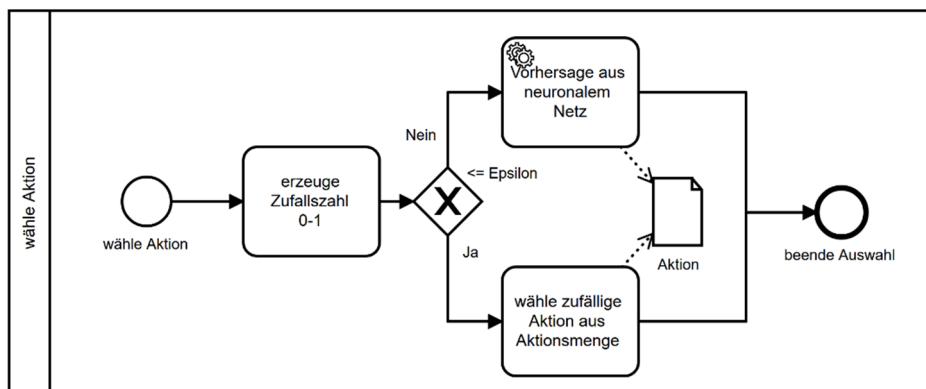


Abbildung 5-5: Auswahlprozess einer Aktion des Agenten

5.3.3 Der Lernprozess

Der gesamte Lernprozess wurde in drei Funktionsgruppen aufgeteilt. Die Funktion „*train*“ übernimmt die globale Aktionsausführung des Trainingsablaufs. Von dieser ausgehend wird ein weiterer Prozess „*remember*“ ausgeführt, der für die Speicherung der Daten und für die Anpassung des Parameters Epsilon zuständig ist. Ebenfalls vom „*train*“-Prozess aus wird der Prozess „*replay*“ gestartet, der das Training des Netzes und die Aktualisierung der Q-Werte durchführt. Der Gesamtablauf dieser Einzelprozesse wird nun anhand Abbildung 5-6 erläutert.

Nachdem ein Simulationsschritt abgeschlossen wurde, soll der Trainingsprozess des lernenden Agenten stattfinden. Die Simulationsumgebung initiiert diesen am Ende jeden Zyklus' des Simulationsagenten (am Ende der Schrittfunktion *mStep*) und übergibt die benötigten Daten wie Zustand, Aktion, Belohnung und Folgezustand. Die Daten müssen für die Trainingsschritte allerdings zuerst formatiert und umgewandelt werden.

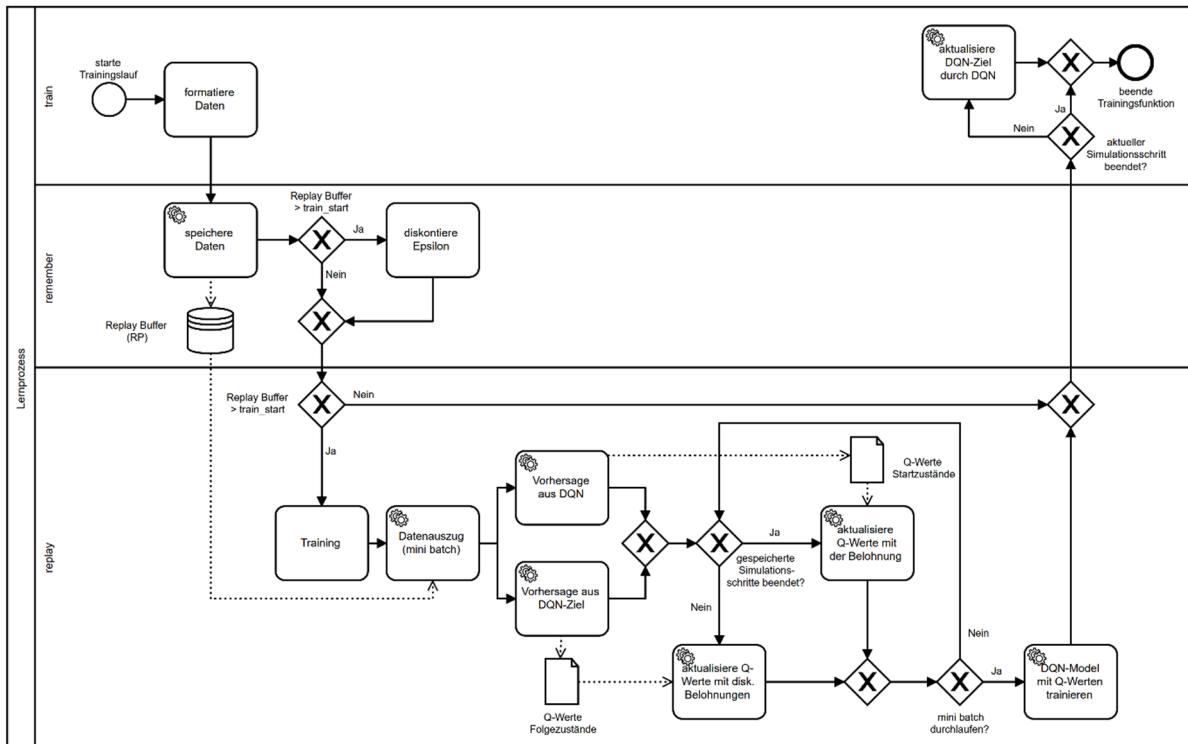


Abbildung 5-6: Ablauf des Lernprozess

Danach werden die Daten in einen Datenspeicher („*Replay Buffer*“) abgespeichert. Dieser Datenspeicher ist ein Array mit fixierter Größe und funktioniert wie ein Überlauspspeicher. Ist der Speicher voll, werden alte, überschüssige Daten gelöscht, sobald neue hinzukommen. Befinden sich im Speicher genügend Daten wird der Parameter ε um einen Faktor diskontiert und das Training des neuronalen Netzes kann beginnen.

Für das Training wird aus dem Datenspeicher ein kleiner Auszug der Daten entnommen, ein sogenannter „*mini batch*“. Dieser Auszug enthält eine Reihe von Zustands-Aktionspaaren, sowie die dazugehörigen Belohnungen und die Information, ob der Simulationsschritt durch die Aktion des Agenten beendet wurde oder eine weitere durchgeführt werden konnte. Aus diesem „*mini batch*“ wird anhand der Zustände eine Vorhersage aus beiden neuronalen Netzen getroffen. Als Q-Werte der Ausgangszustände wird die Vorhersage aus dem Hauptnetz gespeichert. Für die Folgezustände wird eine Vorhersage aus dem Zielnetz gemacht und als Q-Werte der Folgezustände gespeichert. Das Zielnetz wird als Referenznetz für einen fixierten Vorhersagewert der Folgezustände benötigt. Es bleibt während aller Schritte eines Simulationslaufs des Simulationsagenten gleich. Dies ist jedoch nur für das hier beispielhaft beschriebene spezielle Verfahren des „Double Q-Learning“ notwendig.

Anschließend wird der gesamte Datenauszug („*mini batch*“) durchlaufen, um die Q-Werte aller Zustands-Aktionspaare zu aktualisieren. Führte ein Zustands-Aktionspaar zum Ende eines Simulationslaufs, werden die Q-Werte des Netzes mit den direkten Belohnungen der Aktionen aktualisiert. Führte eine Aktion nicht zum Ende des Simulationslaufs, wird die Summe der Belohnungen und der mit ϵ diskontierten Q-Werten aller Folgezustände gebildet, da der Agent mehrere Schritte hintereinander durchführen konnte. Diese Q-Werte werden nun in die Trainingsfunktion eines neuronalen Netz-Frameworks gegeben, um die Netzwerkgewichte anzupassen.

Zum Abschluss wird nun das Zielmodell mit dem tatsächlichen Modell aktualisiert, falls der letzte vollzogene Simulationslauf des Simulationsagenten nicht beendet wurde. Wurde sie beendet, wird der Trainingslauf direkt abgebrochen.

5.4 Technische Entwicklungsumgebung

Das eben beschriebene Konzept kann mit verschiedensten Softwarewerkzeugen umgesetzt werden. Im Wesentlichen müssen zwei Bestandteile zusammengebracht werden: Zum einen die ereignisdiskrete Simulation und zum anderen ein lernendes Element, welches mit Methoden des maschinellen Lernens und der Nutzung von neuronalen Netzen die Steuerungsintelligenz eines Agenten übernehmen soll. Daher wird in der vorliegenden Arbeit begrifflich zwischen dem Simulationsagenten und dem *Reinforcement-Learning-Agenten* (RL-Agenten) unterschieden. Dabei bildet der Simulationsagent den Teil der Anwendung im Simulationsmodell ab und der RL-Agent den intelligenten Teil in einem *Machine-Learning-Framework*. Diese Definition und Aufteilung wurde sowohl im konzeptionellen Aufbau als auch in der späteren Umsetzung der agentenbasierten Materiaflussteuerung herangezogen.

5.4.1 Simulationssoftware

Bei der Entwicklung eines Ansatzes zum simulationsbasierten maschinellen Lernen von Strategien zur Steuerung von Materialflüssen, entstehen Anforderungen an das zu verwendende Simulationswerkzeug. Zum einen bedarf es einer flexiblen und vielseitig anwendbaren Simulationsumgebung, die eine abstrakte Abbildung von Produktionsprozessen zulässt. So kann ein Agent entwickelt werden, dessen Handlungen hervorragend mit neuronalen Netzen generalisiert werden. Zum anderen ist die Eigenschaft der diskreten Simulation relevant, da der Agent während eines Simulationslaufs in den Ablauf des Simulationsmodells eingreifen muss und in einer Drittanwendung die Informationen des Simulationsmodells direkt in einem neuronalen Netz verarbeitet. Daher kommt in der vorliegenden Arbeit ein in der Industrie weit verbreitetes Simulationswerkzeug, Plant Simulation, zum Einsatz. Dieses Werkzeug ist Teil des Product-Lifecycle-Management-Portfolios Tecnomatix von Siemens. Es wird zur Simulation von Logistik- und Produktionssystemen eingesetzt. Damit werden digitale Modelle dieser Systeme erstellt und deren komplexe Abläufe und Steuerungen untersucht und optimiert. Anhand eines Modells können Experimente und „Was wäre wenn“-Szenarien durchgeführt werden, ohne die physikalischen Produktionssysteme zu beeinflussen. Die ereignisdiskrete Materialflusssimulation ist zudem ein wesentlicher Baustein in Konzepten rund um die Digitale Fabrik und Industrie 4.0 (Eley 2012, S. 12–13), was die Relevanz von agentenbasierten Konzepten nochmals unterstreicht. Eine Übersicht der verwendeten Softwarewerkzeuge ist in Anhang A: „Übersicht der technischen Entwicklungsumgebung“ zu finden.

5.4.2 Machine-Learning-Umgebung

Der lernende Teil dieses Ansatzes, der RL-Agent, wurde in der Arbeit mit dem Softwarewerkzeug Python umgesetzt. Python ist eine universelle, höhere Programmiersprache, die einen knappen, gut lesbaren Programmierstil ermöglicht. Darin finden mehrere Programmierparadigmen, wie z. B. die objektorientierte und die funktionale Programmierung, Anwendung. Zudem bietet sie eine dynamische Typisierung von Daten, weshalb sie oft als Skriptsprache genutzt wird. Diese Programmiersprache ist besonders für die Entwicklung von lernenden Agenten geeignet, da hierfür bereits viele Programmpakete für das maschinelle Lernen entwickelt wurden. Sie ermöglichen mit geringem Aufwand bei der Einarbeitung und ohne langjährige Erfahrung im Bereich des maschinellen Lernens, Methoden und Programme zu entwickeln, die selbstständig lernen.

Für den Aufbau des RL-Agenten wurde die Anaconda-Distribution genutzt. Dabei handelt es sich um eine Data-Science-Plattform, die bereits eine große Anzahl wichtiger Module für die Anwendung von maschinellem Lernen bietet wie z. B. NumPy, TensorFlow und Keras.

Anaconda bietet unter anderem ein sehr anwenderfreundliches Verwalten, Administrieren und Starten von virtuellen Entwicklungsumgebungen. Dadurch können für spezielle Aufgaben bzw. Entwicklungen benötigte Bausteine einfach installiert und getestet werden ohne eine andere Umgebung zu beeinflussen. Des Weiteren können die virtuellen Umgebungen unkompliziert auf ein anderes Computersystem transferiert werden.

Für die Entwicklung des Reinforcement-Agenten verwendeten Softwarekomponenten kann Anhang A: „Übersicht der technischen Entwicklungsumgebung“ entnommen werden.

5.4.3 Framework für neuronale Netze

Keras ist eine Open-Source-Bibliothek zur schnellen Implementierung von neuronalen Netzwerken für Anwendungen des Deep Learnings. Keras ist seit der Version 1.4 zwar Teil der TensorFlow Core API, wird jedoch als eigenständige Bibliothek weitergeführt. Sie wird in der Programmiersprache Python geschrieben und bietet eine einheitliche Schnittstelle für Frameworks wie TensorFlow, Theano und Microsoft Cognitive Toolkit. Dabei ist Keras kein Framework für das Deep-Learning, wie beispielsweise TensorFlow, sondern ein Interface für dieses. Keras unterstützt mehrere Backends, wodurch einfach und schnell auf ein anderes Backend gewechselt werden kann. Ein weiterer Vorteil von Keras ist der flexible Einsatz auf verschiedenen Computerplattformen, wie z. B. iOS, Android oder Google Cloud. Merkmale wie Erweiterbarkeit, Modularität und Einfachheit ermöglichen für die vorliegende Arbeit eine unkomplizierte und schnelle Implementierung des Deep-Learning-Agenten, einen universellen Einsatz sowie eine flexible Erweiterung. Als Backend von Keras wird in der vorliegenden Arbeit TensorFlow verwendet. Auf die genaue Funktionsweise dieses Backends soll jedoch nicht weiter eingegangen werden (Wikipedia 2020a)

5.4.4 Datenschnittstelle

Zwischen der Simulationsumgebung und dem lernenden Agenten besteht eine Verbindung zum Austausch von Zustandsdaten, Aktionen und Belohnungen des RL-Agenten. Der Simulationsagent muss vom lernenden Agenten Vorhersagen über eine geeignete Aktion erhalten. Hierfür wurde die Schnittstelle über sogenannte Sockets herangezogen, die bereits in der Simulationssoftware vorhanden ist und in Python einfach und schnell implementierbar ist. Als Socket wird eine bidirektionale Software-Schnittstelle bezeichnet, welche Anwendungen mittels eines Netzwerkprotokolls wie TCP/IP oder UDP miteinander verbindet (d. h. Server und Client). Sie bietet eine höhere Abstraktionsebene für die darunterliegende Kommunikationsinfrastruktur und ermöglicht die Unterstützung einer netzwerkbasierten Anwendungsentwicklung (Ciubotaru und Muntean 2013, S. 73). Die Socket-Kommunikation bildet die Grundlage der heute am weitesten verbreiteten Kommunikationssoftware. Sockets bauen eine Punkt-zu-Punkt-Verbindung in der Initialisierungsphase auf und erlauben danach

einen Online-Informationsaustausch. Dabei fungiert ein Prozess als Server, bei dem sich dann weitere Prozesse als Clients anmelden. Sowohl Plant Simulation als auch eine Python-Anwendung kann dabei Client oder Server sein. Für die Auswahl des Datenprotokolls wurde TCP festgelegt, da es sich hierbei um ein verbindungs- und paketorientiertes Übermittlungsprotokoll handelt. Es wird eine sichere Verbindung zwischen zwei Endpunkten (Sockets) hergestellt, die abgeschlossene Datenpakete versendet. Dadurch wird bei der Verbindung, im Gegensatz zu UDP, sichergestellt, dass alle Datenpakete beim anderen Ende der Verbindung ankommen. Beim UDP-Protokoll kann nicht garantiert werden, dass die gesendeten Daten auch in der gleichen Reihenfolge bzw. vollständig ankommen, wie sie versendet wurden. Zudem setzt die Socket-Verbindung direkt auf TCP/IP. Somit ist eine schnelle Datenübertragung ohne großen zusätzlichen Overhead gewährleistet (Wikipedia 2020b).

6 Umsetzung eines Prototypen zur Anwendung des maschinellen Lernens in der Materialflusssimulation

Anhand eines Beispiels soll nun das Konzept des agentenbasierten Ansatzes in die Simulationsumgebung Plant Simulation implementiert werden. Die Arbeit zeigt, dass ein Softwarebaustein durch die Interaktion mit seiner Umwelt und mittels Verfahren des maschinellen Lernens die richtigen Aktionen in seiner Umgebung zur Erreichung eines bestimmten Ziels erlernen kann und somit zu einem selbststeuernden Agenten wird.

Das Modell bildet auf einer hohen Abstraktionsebene eine klassische Aufgabe der Logistik ab. Hierfür wurde die Aufgabe der Auftragsreihenfolgebildung zur Belegung von Maschinen eines Steuerungssystems gewählt.

6.1 Agentenbasierte Reihenfolgeplanung

Das Modell bildet einen Prozess zur Auftragsverteilung bzw. Reihenfolgebildung ab. Dieser findet vor einem Bearbeitungsprozess mit drei parallel angeordnete Arbeitsstationen statt. Die Reihenfolge wird aus fünf sortenreinen Eingangspuffern gebildet (siehe Abbildung 6-1). Die Initialisierung des Modells findet durch die Methode *init* statt. Darin werden die Puffer (BufferA, BufferB, BufferC, BufferD, BufferE) mit einer vom Experimentverwalter zufällig gewählten Menge (0-10) sortenrein befüllt. Der Agent soll nun die Aufgabe übernehmen, den Stationen die Aufträge in einer geeigneten Reihenfolge bereitzustellen. Es besteht keine direkte Verbindung zwischen Puffer und Agentenbaustein bzw. zwischen Agentenbaustein und den Stationen, um die automatische Umlagerung der Standardbausteine zu unterdrücken. Die Stationen werden anschließend umgerüstet und beginnen mit der Bearbeitung der Aufträge. Nach Abschluss des Bearbeitungsvorgangs verlassen die Aufträge das System und werden in der Senke („drain“) vernichtet.

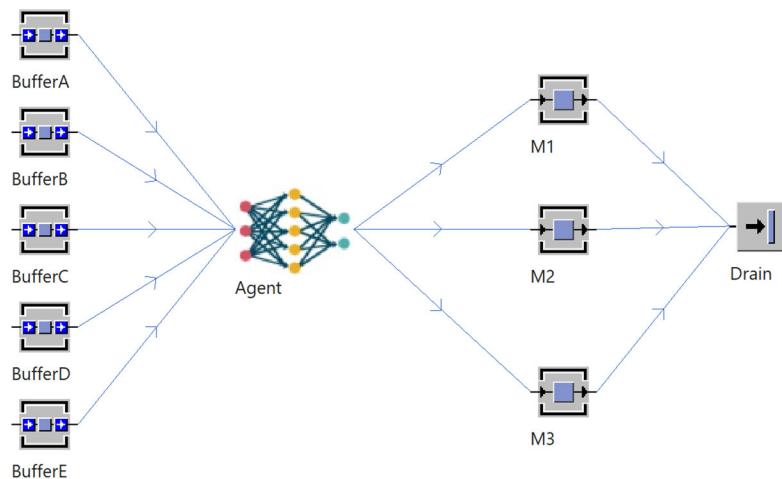


Abbildung 6-1: Das angewandte Simulationsmodell

Auf die Funktionsweise des Agenten-Bausteins und eine detaillierte Beschreibung wird im nachfolgenden Kapitel eingegangen.

6.2 Umsetzung des Simulationsmodells

Das Simulationsmodell für das Training des RL-Agenten besteht aus insgesamt vier Bausteinen. Bei der Auswahl der Bausteine wurde darauf geachtet, Standardbausteine der Simulationssoftware Plant Simulation zu verwenden, um die Komplexität des Simulationsmodells zu reduzieren. Im Folgenden wird nun zum besseren Verständnis auf die einzelnen Bestandteile des Modells eingegangen.

Puffer

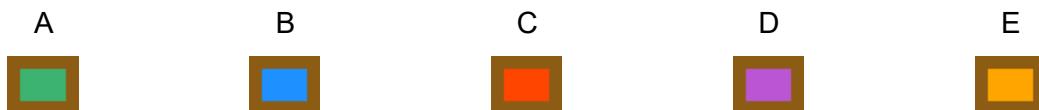
Insgesamt wurden fünf Puffer für eine sortenreine Lagerung vor den Arbeitsstationen eingesetzt. Die Puffer arbeiten nach dem FIFO-Prinzip und besitzen eine Kapazität von zehn Plätzen.

Baustein	Eigenschaften
 Buffer	<ul style="list-style-type: none">• Kapazität: 10 Plätze• Puffertyp: Warteschlange (First in – First out)

Tabelle 6-1: Eigenschaften des Pufferbausteins

Fördergüter (BEs)

Als bewegliches Element (BE) des Materialflusses wurden fünf Fördergüter (A bis E) festgelegt, die verschiedene Aufträge bzw. Produkte einer Fertigung abbilden. Jedes der Fördergüter wird farblich unterschieden:



Stationen

Drei parallel angeordnete Einzelarbeitsstationen führen den Bearbeitungsprozess der Aufträge durch. Jede Arbeitsstation kann alle fünf Auftragstypen (A-E) bearbeiten, jedoch mit unterschiedlicher Bearbeitungsdauer und Rüstzeit zwischen den Aufträgen.

Baustein	Typ	Eigenschaften
	Bearbeitungszeiten (stochastisch)	Erlang-Verteilung
Station		
	Produktspektrum	A-E
	Rüstzeiten	Rüstzeitmatrix
	Störungen	Verfügbarkeit 100%

Tabelle 6-2: Eigenschaften der Stationen

Die Arbeitszeiten folgen einer Erlang-Verteilung und werden mittels interner Methode `self.OnEntrance` im Eingang jeder Station vor Bearbeitungsbeginn berechnet. Die Zeitdaten werden hierfür aus der Tabelle `tbl/ProcTimes` ausgelesen, um die Zeiten zentral und einfach abändern zu können. Sie besteht aus weiteren Untertabellen `times` pro Station, worin die unterschiedlichen Bearbeitungszeiten für die fünf Auftragsarten hinterlegt sind. Die Tabelle der Prozesszeiten ist schematisch in Tabelle 6-3 dargestellt.

Stationen	Prozesszeiten [type:table]	MTTR [type:time]
M1	times	1:00 min
M2	times	1:00 min
M3	times	1:00 min

Tabelle 6-3: Prozesszeiten der Arbeitsstationen

	A	B	C	D	E
-	-	-	-	-	-
A	-	50s	30s	10s	30s
B	30s	-	10s	50s	30s
C	50s	10s	-	40s	30s
D	10s	50s	40s	-	10s
E	30s	30s	30s	10s	-

Tabelle 6-4: Rüstzeitmatrix der Stationen

Die Rüstzeiten der Stationen werden anhand einer Rüstzeitmatrix in jeder Station hinterlegt. Eine Umrüstung erfolgt automatisch in Abhängigkeit des BE-Namens. Die Rüstzeitmatrix gilt für alle drei Stationen und kann Tabelle 6-4 entnommen werden.

6.3 Aufbau und Funktionsweise des Simulationsagenten

6.3.1 Architektur des Simulationsagenten

Der Simulationsagent wurde als eigenständiges Netzwerk modelliert und besteht im Wesentlichen aus Methoden, Variablen und Tabellen, die im Zusammenspiel die Charakteristik des Simulationsagenten bestimmen. Dabei ist das Ziel, den Baustein als wiederverwendbaren Bibliotheksbaustein zu entwickeln, der sich in ein beliebiges Modell einbinden lässt. Um weiterführenden Arbeiten eine verständliche und modulare Architektur zu

bieten, gliedert sich die informationstechnische Struktur des Agenten in sechs Funktionsgruppen (siehe Abbildung 6-2).

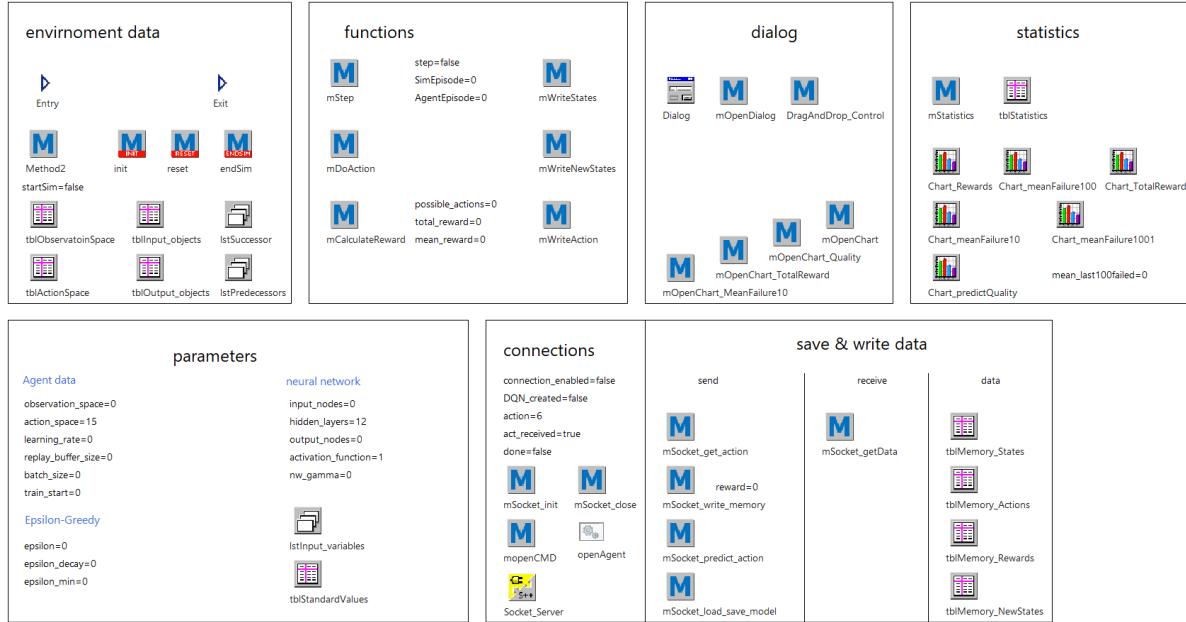


Abbildung 6-2: Aufbau des Agenten in Plant Simulation

„Environment data“

In der Gruppe „environment data“ befinden sich zwei Übergänge als Eingang und Ausgang des Agenten. Sie dienen als Verbindungselement zwischen den Elementen der Simulationsumgebung und dem Agentenbaustein selbst. Dadurch kann die *init*-Methode die Vorgänger- und Nachfolgerobjekte des Bausteins abfragen. Alle über eine Kante verbundenen Objekte werden in den Tabellen *tblInput_objects* und *tblOutput_objects* erfasst, um die Zustände der Umgebungsobjekte, die in direktem Kontakt mit dem Agenten stehen, automatisiert zu erfassen.

„Functions“

Als „functions“ werden alle Methoden zusammengefasst, die für den Agentenalgorithmus innerhalb von Plant Simulation zuständig sind. Das zentrale Element des Agenten ist die *mStep*-Methode. Sie gibt den Rhythmus des Simulationsagenten vor und bestimmt die Arbeitsabfolge des Simulationsagenten und des RL-Agenten. Von dieser Methode aus werden weitere Methoden zur Interaktion des Simulationsagenten mit der Umgebung gestartet. Dazu zählen Berechnungen sowie die Erfassung von Daten der Zustände und Aktionen, die erreicht und getätigt werden. Zusätzlich beinhaltet das Element „functions“ wichtige Parameter, die während der Aktivität des Simulationsagenten relevant sind, wie z. B. die Anzahl der möglichen

und der durchschrittenen Episoden oder die Summe der Belohnungen. Eine detaillierte Auflistung der Elemente sowie eine Beschreibung der Funktionsweise wird später in diesem Kapitel dargelegt.

„Parameters“

In diesem Abschnitt befinden sich alle wichtigen Parameter für den RL-Agenten, der für das Training zuständig ist, sowie sämtliche Parameter, die das neuronale Netz benötigt. Diese können mittels Dialogfenster manipuliert und angepasst werden. Einige Parameter des RL-Agenten werden jedoch nur von der Umgebung bestimmt und können nicht individuell angepasst werden. Zusätzlich gibt es eine Liste mit den Eingangsparametern für den RL-Agenten in Python, welche die Standardwerte der Parameter gespeichert hat.

„Connections“

Hierunter werden die Bausteine für die Kommunikation zwischen Plant Simulation und dem RL-Agenten in Python abgelegt. Die Kommunikation läuft über eine Socket-Schnittstelle die per TCP/IP-Protokoll Informationen weiterleitet. Dabei stellt die Simulationsumgebung über eine Socket-Verbindung den lokalen Host des Rechners als Server dar. Die Datei `agent.py` erzeugt ein Client-Socket, welches über den lokalen Host mit Plant Simulation kommuniziert.

„Save & write data“

Darunter sind alle Methoden und Tabellen erfasst, die als Teil des „connections“-Moduls die Kommunikation zum RL-Agenten in Python übernehmen. Plant Simulation speichert alle Daten bezüglich Umgebungszustände, erhaltener Belohnungen und ausgeführter Aktionen in Tabellen. Die Methoden bereiten die Daten, wie Zustände, Belohnungen und Aktionen von Plant Simulation, zum Versenden über die Socket-Schnittstelle vor und senden diese als Byte-Array per Socket an die Client-Anwendung. Alle ankommenden Daten werden erfasst und an die entsprechenden Variablen und Tabellen weitergegeben.

6.3.2 Erfassen der Zustand-Aktionspaare

Damit der RL-Agent mit seiner Umwelt interagieren kann, benötigt er Sensoren in der Umgebung, die deren Zustand beschreiben. Hierfür müssen genügend Sensoren zur Verfügung gestellt werden, die einen Einfluss auf das Verhalten des Agenten ausüben. Die Prämisse hierfür ist es, so wenige Informationen wie nötig zu übergeben. Dadurch kann die Modellkomplexität des RL-Agenten deutlich reduziert werden. Zudem ist somit gewährleistet, dass der RL-Agent eine generalisierte Antwort auf die Eingangsgrößen liefern kann. Zwar führen mehr Informationen zu einem schnelleren Lernergebnis, jedoch wird der RL-Agent dadurch stärker auf eine bestimmte Situation trainiert und kann in für ihn unbekannten

Situationen und Umgebungen schlechter eine optimale Lösung approximieren. Werden hingegen zu wenige Information der Umgebung bereitgestellt, kann das neuronale Netz keine geeignete Funktion abbilden und somit auch keine gute Vorhersage treffen.

„Observation space“

Die Anzahl der Sensoren und Parameter, die einen Zustand der Umgebung gegenüber dem RL-Agenten darstellen, nennt sich „*observation space*“ und wird in Tabelle 6-5 für das Konzeptbeispiel dargestellt. Insgesamt besteht eine Beobachtung im Konzeptmodell aus 15 Zustandsgrößen. Die Anzahl der enthaltenen Aufträge jedes Eingangspuffers ist eine der Zustandsgrößen. Diese kann Zahlenwerte zwischen null und zehn annehmen, je nach Anzahl der Aufträge. Eine weitere Beobachtung bildet die Belegung aller drei parallel angeordneten Maschinen ab. Diese nimmt nur Werte von 0 (false) oder 1 (true) an. Um dem RL-Agenten für die Reihenfolgebildung geeignete Informationen über den Zustand der Umgebung zur Verfügung zu stellen, wird der Rüststatus der drei Stationen gespeichert. Die Werte zwischen eins und fünf repräsentieren den Rüststatus der fünf Auftragstypen A-E (1-5).

Objekte	A	B	C	D	E	M1	M2	M3	M1	M2	M3
Zustände	0-10	0-10	0-10	0-10	0-10	true/ false	true/ false	true/ false	1-5	1-5	1-5

Tabelle 6-5: Zustandsgrößen (*observation space*)

„Action space“

Der Aktionsraum des Agenten bestimmt die Handlungsmöglichkeiten des Simulationsagenten. Diese wiederum legen auch die Anzahl der Knoten der Ausgangsschicht der neuronalen Netze fest. Der DQN-Algorithmus wählt das am stärksten feuernende Neuron der Ausgangsschicht aus, welches der jeweils festgelegten Aktion entspricht. In Tabelle 6-6 sind alle möglichen Aktionspaare des Agenten aufgeführt. Jede Aktion entspricht einer Kombination von Eingangspuffer zu Bearbeitungsstation. Beispielsweise kann vom Eingangspuffer A (1) ein Auftrag zur Ausgangsstation M2 (2) umgelagert werden, was der Aktionsnummer 6 entspricht.

Aktion	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Eingangsobjekt	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Ausgangsobjekt	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3

Tabelle 6-6: Aktionsraum (*action space*)

6.3.3 Die Schrittfunktion

Die Methode „*mStep*“ stellt das zentrale Element des Simulationsagenten in Plant Simulation dar. Sie regelt den Ablauf des Agenten und die Kommunikation mit dem Reinforcement-Learning-Agenten. Da diese Methode auch für das Verständnis und die Systematik im Ablauf des entwickelten Agentenansatzes elementar ist, wird diese nun im Detail anhand Abbildung 6-3: Ablauf der *mStep* Funktion erläutert.

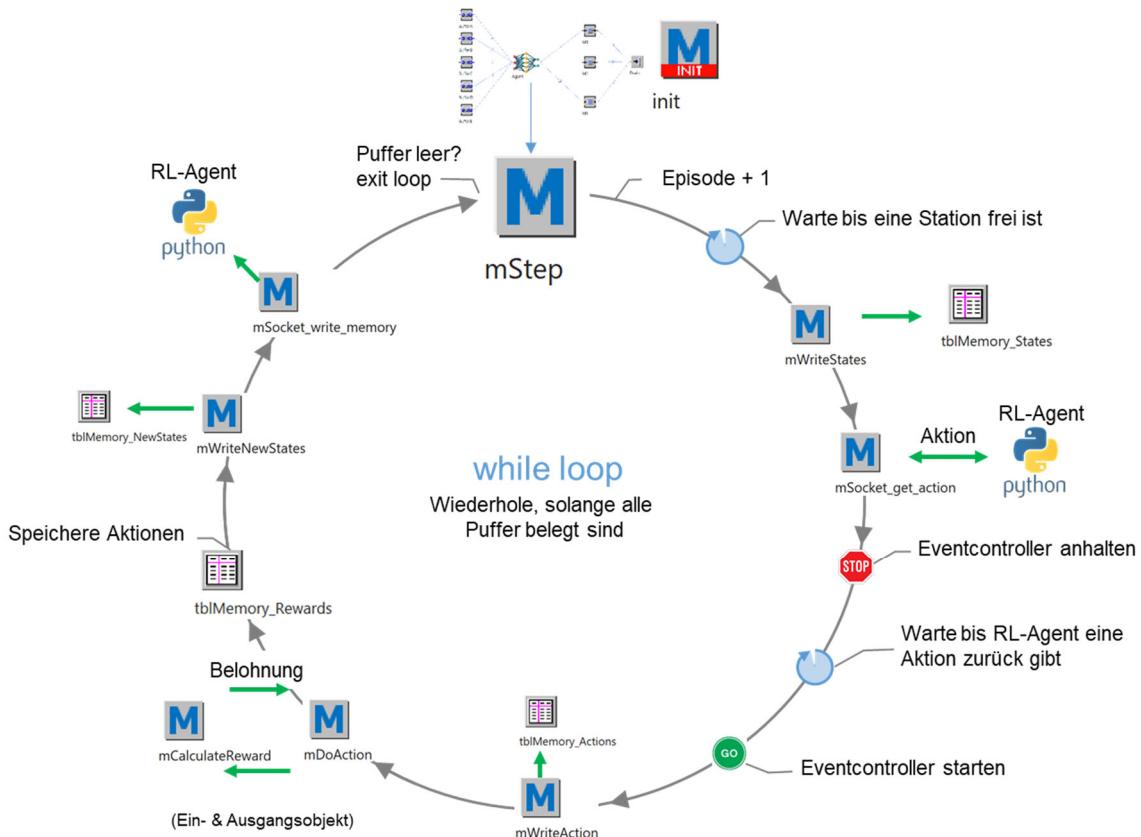


Abbildung 6-3: Ablauf der *mStep* Funktion

Die *init*-Methode löst beim Starten des Simulationslaufes die Methode *mStep* aus. Diese besteht im Wesentlichen aus einer While-Loop-Anweisung, die solange ausgeführt wird bis alle vorgelagerten Puffer des Agenten keine Aufträge mehr enthalten. Der Agent durchläuft diese Schleife pro Aktion einmal und zählt dabei die Anzahl der Durchläufe. Ist die vom RL-Agenten ausgewählte Aktion in der Simulationsumgebung nicht zulässig, wird der Abbruch der Schleife eingeleitet. Sobald alle Aufträge in den Puffern abgearbeitet sind, wird die Schleife ebenfalls verlassen.

Sobald eine der nachfolgenden Stationen frei wird, wird der aktuelle Zustand der Umgebung s in die Tabelle *tblMemory_States* abgespeichert. Nun schickt der Simulationsagent eine Anfrage zur Auswahl einer Aktion a an den RL-Agenten. Die Auswahl der Aktion sowie die

Berechnung des neuronalen Netzes innerhalb des RL-Agenten benötigen etwas Zeit, daher wird die Simulation angehalten, bis eine Antwort des RL-Agenten in Form einer Aktion a zurückgegeben wurde. Dieser wählt zufällig eine Aktion aus dem Aktionsraum (*action space*) aus oder schickt eine Anfrage an das neuronale Netz für eine Vorhersage der bestmöglichen Aktion. Sobald diese vorhanden ist, wird die Aktion in eine Datentabelle *tblMemory_Actions* abgespeichert und von Plant Simulation weiterverarbeitet. Hierfür wird aus der Tabelle *tblActionSpace* anhand der Aktionsnummer ein Vorgänger-Nachfolger-Paar ausgelesen. Anhand dieses Zahlenpaares kann der entsprechende Vorgänger (*in*) bzw. Nachfolger (*out*) des Simulationsagenten für eine Umlagerung der Aufträge angesprochen werden (siehe Abbildung 6-4).

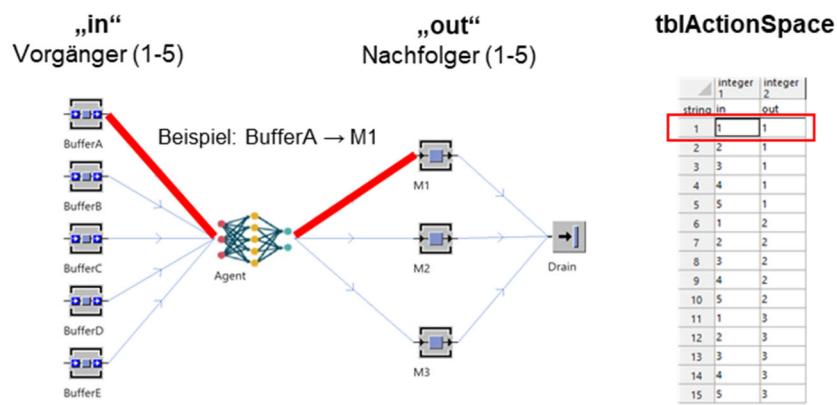


Abbildung 6-4: Auswahl einer Aktion (Vorgänger-Nachfolger)

Nun prüft die Methode *mDoAction*, ob die vom RL-Agenten ausgewählte Aktion zulässig ist. Sie bekommt von der Schrittfunction *mStep* die Nummer des Vorgängers und des Nachfolgers übergeben. Ist die Aktion zum Umlagern möglich, wird ein Auftrag aus dem Eingangsobjekt (Puffer) entnommen und auf das Ausgangsobjekt (Station) umgelagert. Ist die Aktion unzulässig, wird die Aktion nicht ausgeführt, eine Bestrafung vergeben und die Trainingsepisode abgebrochen. In beiden Fällen muss die Belohnung für die ausgewählte Aktion mit der Belohnungsfunktion *mCalculateReward* berechnet werden. Da diese Funktion für das Lernen eines RL-Agenten besonders wichtig ist, wird sie in Kapitel 6.3.4 im Detail beschrieben.

Anschließend werden die Belohnungen in die Tabelle *tblMemory_Rewards* gespeichert und der neue Zustand s' der Umgebung in *tblMemory_NewStates* notiert. Zum Schluss werden alle Daten an den RL-Agenten per Socket-Schnittstelle übertragen. Der RL-Agent speichert alle Zustands-Aktionspaare sowie deren Belohnung in den Datenspeicher („Replay Buffer“) und führt einen Trainingsschritt aus. Sowohl Schnittstelle als auch der RL-Agent werden in den nachfolgenden Kapiteln ausführlich dargestellt. Nun beginnt die Schleife der *mStep*-Funktion

von vorne. Sie wird solange wiederholt bis alle Eingangspuffer geleert sind oder die Episode vom Agenten abgebrochen wird.

6.3.4 Die Belohnungsfunktion

Über die Belohnungsfunktion erfährt der Agent eine Rückmeldung seiner Umwelt, wie gut oder schlecht seine Aktion im aktuellen Zustand war. Daher ist die Berechnung der Belohnung („reward“) das wichtigste Element für die Methode des Reinforcement Learning. Belohnung bzw. Bestrafung wirken sich direkt auf das Lernen aus, daher ist eine genaue Abstimmung der beiden Größen für den Agenten essentiell. Die Belohnungsfunktion im vorliegenden Konzeptmodell basiert auf der Vergabe von ganzen Zahlen als Punkte für eine Aktion in einem bestimmten Zustand der Umgebung. Der Aufbau und die Arbeitsweise der Belohnungsfunktion erläutert Abbildung 6-5.

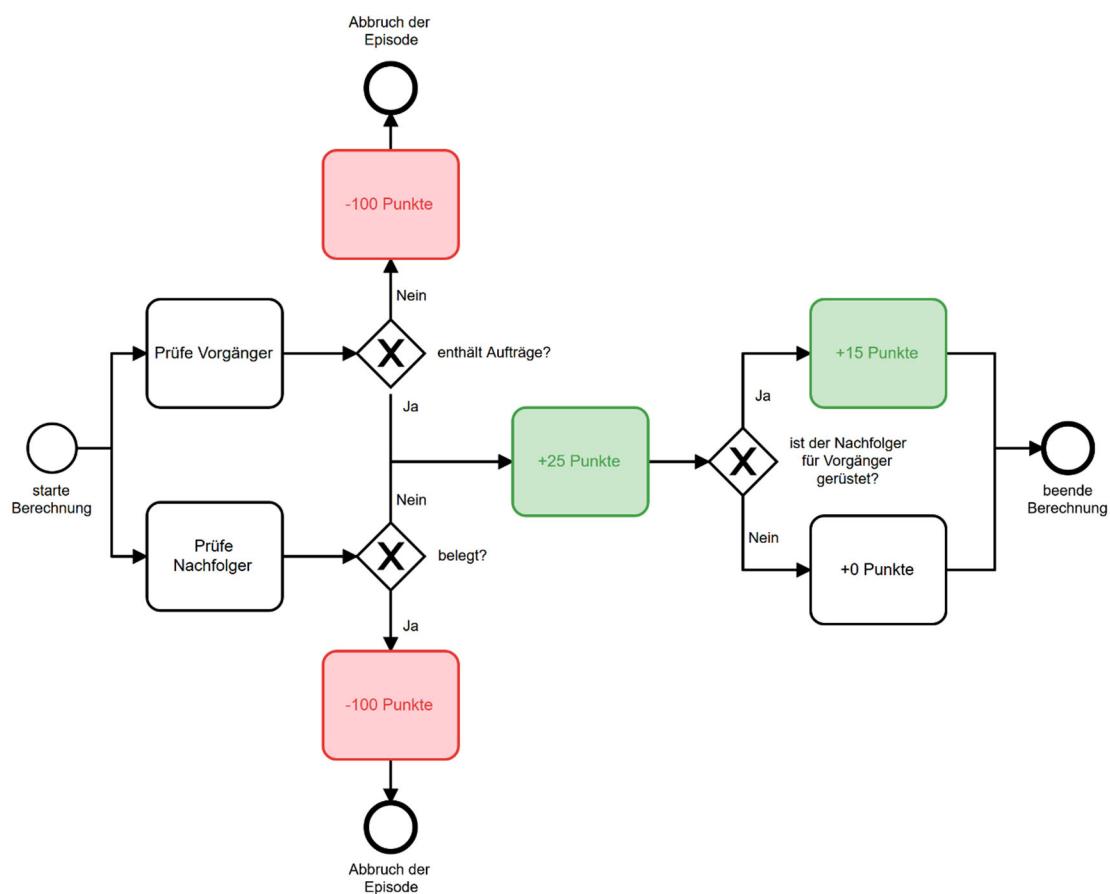


Abbildung 6-5: Die Belohnungsfunktion des Agenten

Der RL-Agent wählt eine Aktion aus, bestehend aus einem Vorgänger und Nachfolger des Simulationsagenten. Diese bekommt die Belohnungsfunktion übergeben und prüft, ob der Vorgänger mit Aufträgen befüllt ist. Gleichzeitig frägt die Funktion die Belegung der Nachfolgerstation ab. Ist der Puffer leer, erhält der Agent eine Bestrafung (-100 Punkte) und die Simulationsepisode bricht ab. Eine bereits belegte Nachfolgerstation führt ebenfalls zu

einem Abbruch und resultiert in einer Bestrafung von 100 Punkten. Ist jedoch der Puffer mit Aufträgen befüllt und die Nachfolgerstation frei, so erhält der Agent eine Belohnung nach Abbildung 6-5. Eine weitere Abfrage prüft, ob der Nachfolger (Station) für den Auftrag des Vorgängers (Puffer) gerüstet ist oder nicht. In diesem Fall werden weitere 15 Punkte Belohnung verteilt. Ist die Maschine nicht für den Auftrag gerüstet, werden keine weiteren Punkte vergeben. So ergibt sich ein Belohnungsspektrum zwischen -100 und 40 Punkten pro erfolgreichem Simulationsschritt bzw. pro Simulationsepisode.

Anforderungen an die Belohnungsfunktion

Bei der Vergabe von Belohnungen ist besonders auf ein ausgewogenes Verhältnis zwischen Bestrafung von schlechten Aktionen und der Belohnung von guten Aktionen zu achten. Wird der Agent für eine schlechte Aktion zu stark bestraft, können die Kantengewichte der Neuronen im Netz vom Algorithmus kaum mehr in die richtige Richtung angepasst werden. Daher muss für eine gute Aktion auch eine entsprechend hohe Belohnung vergeben werden. Für das Beispielmodell der vorliegenden Arbeit wurde ein Verhältnis von optimaler Belohnung zu Bestrafung von 2:5 (+40/-100) experimentell erprobt. Durch die zweieinhalbfache Bestrafung wird die nicht zulässige Aktion im Netz entsprechend stark gewichtet. So lernt der Agent, welche Aktionen erlaubt sind und welche nicht. Allerdings kann er während des Trainings auch nicht zulässige Aktionen durchführen. Eine Beschränkung auf zulässige Aktionen wurde nicht implementiert, um das Lernen von Aktionen in den Vordergrund zu stellen und dem Agenten zu ermöglichen, sein Verhalten in verschiedenen Umgebungen flexibel auf abstrakter Ebene zu generalisieren. In Tabelle 6-7 sind alle möglichen Aktionen und die hierfür erhaltene Belohnung aufgeführt.

Nr.	Vorgänger (Puffer)	Nachfolger (Station)	Nachfolger (Station)	Belohnung
1	belegt	belegt	gerüstet	-100
2	belegt	belegt	nicht gerüstet	-100
3	belegt	frei	gerüstet	+40
4	belegt	frei	nicht gerüstet	+25
5	leer	belegt	gerüstet	-100
6	leer	belegt	nicht gerüstet	-100
7	leer	frei	gerüstet	-100
8	leer	frei	nicht gerüstet	-100

Tabelle 6-7: Übersicht Belohnung und Bestrafung

6.3.5 Datenverwaltung in der Umwelt

6.3.6 Verarbeitung der Daten zur Kommunikation mit dem RL-Agenten

Der RL-Agent benötigt für das Training zum einen den Zustand (*state*) der Simulationsumgebung, zum anderen die ausgeführte Aktion (*action*) sowie die hierfür

erhaltene Belohnung (*reward*) und den Folgezustand (*next state*) der Umgebung. Wie im vorherigen Kapitel 6.3.5 beschrieben, werden diese in Tabellen der Simulationssoftware abgespeichert und am Ende eines Schrittes als Array per TCP/IP-Socket an den RL-Agent gesendet.

Senden von Zuständen und Belohnungen

Die Socket-Schnittstelle enthält eine Reihe von Befehlen zum Senden von Daten, um dem RL-Agenten die Daten in einem einheitlichen Datengerüst bereitzustellen. Anhand des Datengerüsts werden diese extrahiert und entsprechend in die benötigten Dateitypen umgewandelt. Hierfür wurde ein geeignetes Datenprotokoll, s. Abbildung 6-6: Datentransferprotokoll (*Memory*), entwickelt.

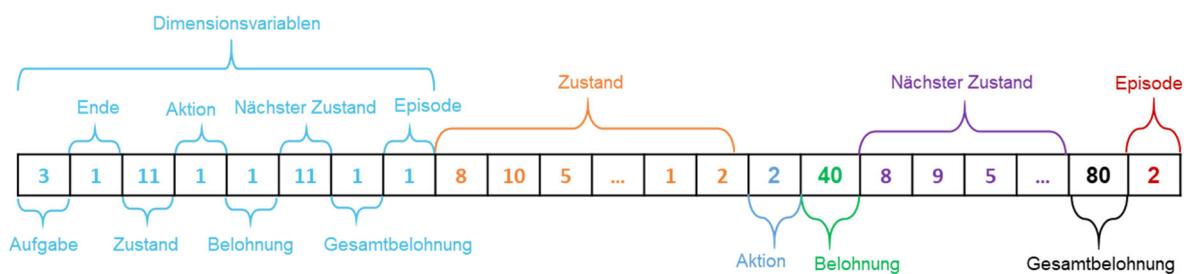


Abbildung 6-6: Datentransferprotokoll (*Memory*)

Zu Beginn des Arrays werden die Dimensionsvariablen notiert. Diese geben an, in welcher Form die Daten vorhanden sind. Sie sind für die Umwandlung und Weiterverarbeitung der im RL-Agenten notwendig. Auf die Verarbeitung und die Bedeutung der Daten wird im Kapitel 6.4 näher eingegangen. Der Zustand der Umgebung sowie die durchgeführte Aktion, die erhaltene Belohnung und der nächste erreichte Zustand werden an das Array angehängt. Am Ende werden dann die Gesamtbewertung, berechnet aus mehreren Durchläufen des Agenten, und die aktuelle Simulationsepisode mitgesendet.

Die Daten werden anhand eines Bytearrays vom Datentyp „*real32*“ an den RL-Agenten übertragen. In der Simulationsumgebung werden mit dem Befehl „*appendValueOfType*“ alle Zustände, Aktionen und Belohnungen als Bytearray gespeichert. Alle Gleitkommazahlen sowie die *Integer*-Zahlen werden durch vier einzelne Bytes repräsentiert und an das Array angehängt. Das Array enthält insgesamt 34 Zahlenwerte. Als einzelne Bytes ergibt sich so ein Array mit einer Länge von 136 Einträgen. Mit dem Befehl „*writearray*“ des Socket-Bausteins wird nun das Bytearray an den RL-Agenten übermittelt und von diesem wieder konvertiert, um mit den Daten weiterzuarbeiten.

Das nachfolgende Kapitel geht genauer auf die Verarbeitung des RL-Agenten ein und erläutert die Funktionsweise sowie seinen Aufbau.

6.4 Aufbau des Reinforcement-Learning-Agenten

6.4.1 Schnittstelle des RL-Agenten

Die Schnittstelle des lernenden Agenten (RL-Agenten) wurde anhand eines WebSockets umgesetzt. Hierfür wird in der Datei „agent.py“ ein TCP/IP Client-Socket erzeugt, welches eine Verbindung zum Server-Socket der Simulationsumgebung aufbaut. Diese kommunizieren über den lokalen Host Server des Rechners. Die Client-Anwendung hört auf der IP-Adresse des lokalen Hosts mit und wartet auf Daten des Server-Sockets. Die Größe der empfangenden Datenpakete beschränkt sich auch einen festen Wert von 1024 Bytes, da in den vorliegenden Prototypen keine größeren Datenpakete gesendet werden. Die Daten werden als Bytarray nach der Umformung der *String*-Werte durch das UTF8 Format gesendet und empfangen. Für eine Weiterentwicklung der Prototypen sollte diese Schnittstelle überarbeitet werden. Problematisch ist bei dem aktuellen Übertragungsweg ist die Handhabung von Datenpaketen. Ist ein Datenpaket größer als die fixierte Bytegröße, kann die vollständige Nachricht nicht empfangen werden und geht verloren.

6.4.2 Datenverarbeitung zur Kommunikation mit dem Simulationsagenten

Die Datei „agent.py“ enthält zwei Hauptelemente des RL-Agenten: die Hauptfunktionen – das Empfangen von Daten und die Ausführung der korrekten Aufgaben – sowie die Klasse „Agent“, welche alle wichtigen Funktionen für das Training des RL-Agenten enthält.

Ähnlich wie im Simulationsagenten wiederholt der RL-Agent eine Endlosschleife solange, bis von der Simulationsumgebung das Ende der Verbindung übermittelt wird oder ein Programmfehler auftritt.

Zu Beginn wird ein Client-Socket über den lokalen Host aufgebaut. In einer weiteren Schleife bleibt das Client-Socket auf Empfang und hört auf der angegebenen IP-Adresse so lange mit bis Daten empfangen werden. Aus dem erhaltenen Datenpaket wird der erste Eintrag des Arrays ausgelesen, die die Aufgabe des Agenten auswählt. Mittels Wenn-Dann-Abfrage werden sieben Aufgabenfälle unterschieden. Folgende Aufgaben sind zu erledigen:

1. Initialisierung des Agenten mit den Basisdaten
2. Training und Sicherung der Daten (Zustände, Aktionen, Belohnungen)
3. Wähle eine Aktion aus
4. Aktion vom neuronalen Netz vorhersagen
5. Sichere das Netz als Datei
6. Lade ein Netz aus einer Datei
7. Beende die RL-Anwendung

Aufgabe 1: Initialisierung des Agenten mit den Basisdaten

Das erhaltene Datenpaket wird anhand der hierfür eigens programmierten Klasse „convData“ übergeben und verarbeitet. Aus dem Bytearray wird ein *numpy-Array* geformt, das für die Anwendung in Python und Keras notwendig ist. Anschließend müssen alle Hyperparameter zur Initialisierung des Agenten und der neuronalen Netze dem Objekt der Klasse „Agent“ übergeben werden.

Hyperparameter sind Parameter, die vor Beginn des Lernprozesses eingestellt werden. Sie sind abstimmbar und können direkt beeinflussen, wie gut ein Modell trainiert wird. Nachfolgende Hyperparameter werden benötigt.

Agent	Neuronales Netz
Zustandsraum (observation space)	Zustandsform (= Zustandsraum)
Aktionsraum (action space)	Aktionsraum (action space)
Replay buffer	Lernrate (learning rate)
Batch size	Anzahl Knoten in versteckter Schicht (hidden layer)
Train start	Aktivierungsfunktion
Epsilon	
Gamma	

Tabelle 6-8: Hyperparameter von Agent und neuronalem Netz

Der Parameter „Replay Buffer“ bestimmt die Größe für den Datenspeicher des RL-Agenten, dessen Form einem Array mit fixierter Länge entspricht. Ist der Datenspeicher voll und es kommen neue Daten hinzu, werden alte Daten am Anfang des Speichers entfernt. Dadurch bekommt der RL-Agent ein Gedächtnis, das mit der Zeit immer wertvollere Informationen zu den Zustands-Aktionspaaren erhält. Die Auszugsgröße bzw. „batch size“ wird für die Aktualisierung der Q-Werte benötigt. Diese bestimmt die Größe des Datenauszugs, anhand dessen der Agent die Gewichte der Netze anpasst und dadurch lernt. Der Parameter „train_start“ sorgt dafür, dass die Datenmenge im „Replay Buffer“ ausreichend groß für einen Trainingslauf ist. Um das Gleichgewicht zwischen Exploration und der Nutzung des bereits gelernten Wissens zu schaffen, findet der Parameter „epsilon“ im *Epsilon-Greedy*-Verfahren Anwendung. Ein weiterer wichtiger Parameter, der die Weitsichtigkeit des Agenten über mehrere Schritte beeinflusst, ist „gamma“. Bei Werten, die nahe „1“ sind, berücksichtigt der Algorithmus über viele Schritte hinweg die Belohnungen. Geht der Wert Richtung null, wird der Agent kurzsichtig, denn es finden nur noch die unmittelbaren Belohnungen Berücksichtigung bei der Berechnung der Q-Werte.

Alle Parameter werden in lokalen Variablen der Klasse „Agent“ gespeichert um diese innerhalb der Hauptfunktion abrufen zu können. In der Initialisierung werden auch die neuronalen Netze als Objekt „model“ und „target_model“ mit den dazugehörigen Parametern erzeugt (siehe Tabelle 6-8). Weitere Hyperparameter, die für das neuronale Netz relevant sind, beschreibt Kapitel 6.4.4.

Aufgabe 2: Training und Sicherung der Daten

Bei dieser Aufgabe startet die Trainingsfunktion des Agenten. Hierdurch werden die Daten (Zustände, Aktionen, Belohnungen, neue Zustände) in den „Replay Buffer“ abgespeichert und das Training des Netzes angestoßen. Dies geschieht nach jedem Schritt des Agenten. Auf das Training wird im nachfolgenden Kapitel näher eingegangen.

Aufgabe 3: Wähle eine Aktion aus

Für diese Aufgabe werden die Inputdaten wieder mittels der Datenverarbeitung „conndata“ konvertiert und aufbereitet. Diese werden als „numpy-Array“ umgeformt und an den Agenten übergeben. Der RL-Agent wählt diese Aktion anhand der internen Funktion „get_action“ aus und gibt sie als *Integer*-Zahl zurück. Ist die Auswahl der Aktion mittels Zufall oder per Vorhersage durch das Netz getroffen, wird sie in der Variablen „rp“ gespeichert. Nachdem die Aktion feststeht, wird diese als Bytarray an Plant Simulation zurückgesendet.

Aufgabe 4: Aktion vom neuronalen Netz vorhersagen

Diese Aufgabe entspricht Aufgabe 3, jedoch wird hier nur die Vorhersage des Netzes ohne eine zufällige Auswahl der Aktion genutzt.

Aufgaben 5-7

Die Aufgaben 5 und 6 sind für das Speichern des Netzes sowie für das Laden eines vorhandenen Netzes notwendig. Aufgabe 7 beendet die Schleife und führt zu einem Abbruch der Socket-Verbindung des RL-Agenten.

6.4.3 Funktionen für den Ablauf des Lernalgorithmus

Die Agentenklasse besteht aus vier Funktionen die nachfolgend erläutert werden. In der Klasse werden für das Lernen und das Training wichtige Hyperparameter bereitgestellt.

get_action

Diese Funktion erhält den aktuellen Zustand der Umgebung und wählt eine der 15 möglichen Aktionen aus. Sie wählt zwischen einer zufälligen Aktion oder einer durch das Netz vorhergesagten Aktion aus. Hierfür wird eine Zufallszahl zwischen 0 und 1 erzeugt. Ist diese kleiner oder gleich dem Hyperparameter Epsilon, wird eine zufällige Zahl aus dem

Aktionsraum zwischen 1-15 erzeugt. Diese Zahl entspricht dann der jeweiligen Aktionsnummer. Ist die zu Beginn gewählte Zufallszahl größer als Epsilon, trifft das neuronale Netz mittels der „*predict*“-Funktion aus Keras eine Vorhersage der Aktion mit dem größten Wert. Als Rückgabewert der Funktion wird neben der ausgewählten Aktion notiert, ob die Auswahl zufalls- oder netzbasiert getroffen wurde. Dies geschieht anhand einer „0“ (Zufallsauswahl) oder einer „1“ (Netzantwort). Dieser Parameter wird in der Simulationsumgebung verwendet um die Entscheidungen des Agenten bewerten zu können.

train

Die „*train*“-Funktion bildet den Trainingsalgorithmus des RL-Agenten ab. Sie wird nach jedem Schritt des Simulationsagenten aufgerufen und führt einen Trainingsschritt aus. Zu Beginn werden die Eingangsdaten aus der Simulationsumgebung übertragen und konvertiert. Hierfür wandelt die Konvertierungsfunktion „*get_memory*“ aus der Klasse „*convData*“ die Daten in die korrekte Form und den korrekten Dateitypen um und speichert diese in die Variablen „*state*“, „*action*“, „*reward*“, „*next_state*“, „*total_reward*“, „*sim_episode*“. Der aktuelle Zustand und der Folgezustand müssen nun noch als „*numpy-Array*“ formatiert werden bevor alle Variablen von der Funktion „*remember*“ in den „*Replay-Buffer*“ abgespeichert werden.

Nachdem die Daten im internen Speicher vorhanden sind, wird die Funktion „*replay*“ gestartet. Sie ist für die Aktualisierung der Q-Werte zuständig und löst die Aktualisierung der Gewichte in den beiden Netzen „*model*“ und „*target_model*“ aus. Die Funktion wird nur gestartet, wenn der „*Replay Buffer*“ die zuvor über den Parameter „*train_start*“ festgelegte Mindestgröße erreicht hat. Dann kann ein zufälliger Auszug des Datenspeichers (*mini batch*) entnommen und alle Zustände, Aktionen, Belohnungen, Folgezustände und die Endvariable der Episoden in Form von Listen gespeichert werden. Diese Listen der Zustände („*states*“) und Folgezustände („*states_next*“) müssen nun noch als *numpy-Array* für Keras formatiert werden. Anschließend beginnt der Aktualisierungsprozess der Q-Werte.

Zuerst müssen die Q-Werte aller aktuellen Zustände aus dem Basisnetz „*model*“ und die Q-Werte der Folgezustände aus dem Zielnetz „*target_model*“ mit der *predict*-Funktion vorhergesagt werden. Diese werden jeweils in die Listen „*q_values*“ und „*q_values_next*“ gespeichert. Als nächstes wird der komplette Datenauszug (*mini batch*) durchlaufen und die Q-Werte aktualisiert. Wurde bei einem Zustands-Aktionspaar die Simulationsepisode beendet (*done=true*) werden die Q-Werte der aktuellen Zustände der unmittelbaren Belohnung für die Aktion in diesem Zustand gleichgesetzt. Wurde die Simulationsepisode hingegen nicht beendet (*done=false*), ist eine weitere Episode möglich. Daher ergibt sich dann der Q-Wert aus der Summe der Belohnung des aktuellen Zustands und der diskontierten Belohnung für den Folgezustand. Die Diskontierung erfolgt über den Parameter „*gamma*“. Dieser Wert kann

zwischen null und eins sein. Für Gamma-Werte gegen null ist der Agent kurzsichtig und betrachtet nur die unmittelbare Belohnung für eine Aktion in einem bestimmten Zustand. Geht der Gamma-Wert gegen eins, wird der RL-Agent weitsichtiger und kann weiter „in die Zukunft“ blicken und erkennt, welche zusätzliche Belohnung er für eine Aktion in späteren Schritten erhält. Abschließend werden die Gewichte des „*model*“-Netzes mit den aktuellen Zuständen und deren Q-Werte mit der Keras-Funktion „*fit*“ angepasst. In Anhang C.9: kann der Ablauf nochmals anhand des kommentierten Programmcodes nachvollzogen werden.

6.4.4 Aufbau und Design des neuronalen Netzes

Um von der Klasse DQN eine Instanz eines neuronalen Netzwerks abzuleiten, sind Programmpakete für neuronale Netze zur Erstellung der Schichten und zur internen Berechnung der Gewichte nötig. In der vorliegenden Arbeit wurde das Paket Keras verwendet. Für die Modellierung des Netzwerks wurden folgende Modell- und Optimierungspakete von Keras importiert:

- *Models*
- *Layers*
- *Optimizers*

Die Klasse DQN

Für den RL-Agenten wurde die Klasse „DQN“ in Python programmiert. Davon wird im RL-Agenten eine Instanz gebildet und ein neuronales Netz erzeugt. Dieses Objekt beinhaltet eine Reihe von internen Funktionen für das Lernen der Gewichte aller Knotenverbindungen. Nachfolgend wird der Aufbau des Klasse erläutert.

Das neuronale Netz erhält bei der Initialisierung fünf Parameter. Dazu zählen zum einen der Hyperparameter Lernrate, die Aktivierungsfunktion der Neuronen sowie die Anzahl der Knoten der versteckten Schichten. Zum anderen benötigt das Netz noch die Umgebungsvariablen Zustandsraum (*state space*) und Aktionsraum (*action space*).

All diese Parameter werden als interne Variablen des DQN-Objektes gespeichert. Zur Erstellung des Netzwerks wird die Anzahl der Eingangsknoten gleich des Zustandsraums festgelegt. Zur Erzeugung der versteckten Schichten müssen der gewünschte Typ der Knotenverbindungen, die Anzahl der Knoten pro Schicht, sowie die Anzahl der vorherigen Knoten bekannt sein.

Design des Netzes

Für die vorliegende Arbeit wurden sogenannte „*Fully-connected Neural Networks*“ verwendet. In Keras werden diese als „*Dense-Layer*“ bezeichnet. Dabei handelt es sich um Netze, bei

denen eine Verbindung zwischen jedem Knoten der vorangegangenen Netzsicht mit allen Knoten der Nachfolgeschicht besteht. Die Dimensionierung des Netzes wird zum Teil vom Anwendungsfall vorgegeben. So besteht die Eingangsschicht aus 11 Knoten. Jeder dieser Knoten steht für eine Eingangsgröße des Umgebungszustandes. Bei der Auswahl der Anzahl der Neuronen für die versteckten Schichten ist nach Frochte (Frochte 2019, S. 189) Folgendes zu beachten:

Prinzipiell sollte die Anzahl der Neuronen eher gering gehalten sein, da zum einen mit mehr Neuronen die Komplexität steigt und sich dadurch der Trainingsaufwand erhöht. Zum anderen neigen neuronale Netze zum Auswendiglernen und es besteht die Gefahr des *Overfittings* (Überanpassung) der Netzgewichte. Daher muss beachtet werden, wie viele Daten für die Funktionsapproximation vorliegen.

Um eine geeignete Anzahl an Knoten zu identifizieren, kann vorliegende Datenmenge mit der Anzahl der Freiheitsgrade des neuronalen Netzes abgestimmt werden. Mit der Formel (6-1) kann die Anzahl der Freiheitsgrade berechnet werden.

$$Freiheitsgrade = (m + 1) \cdot h_1 + h_1 \cdot h_2 + h_2 \cdot n \quad (6-1)$$

$$Freiheitsgrade des DQN = (11 + 1) \cdot 64 + 64 \cdot 64 + 64 \cdot 15 = 5.824$$

Für das Netz „*model*“ und das Netz „*target_model*“ wurden zwei verdeckte Schichten mit je 64 Knoten festgelegt. Dies bedarf einer Datenmenge im „*Replay Buffer*“ von mindestens 5.824 Zustands-Aktions-Paaren. Jeder der Knoten erhält zusätzlich als Aktivierungsfunktion die „*relu*“-Funktion. Diese Aktivierungsfunktion wird bei tiefen neuronalen Netzen am häufigsten verwendet. Der Grund hierfür ist, dass deren Gradient stückweise konstant und daher robust ist. Auf die genaue Funktionsweise dieser Funktion wird hier jedoch nicht näher eingegangen.

Die Ausgangsschicht entspricht der Anzahl der Aktionen, die der RL-Agent auswählen kann. Diese Anzahl wird durch das Simulationsmodell vorgegeben. Dort gibt es fünf Eingänge und drei Ausgänge, was zu insgesamt 15 verschiedenen Möglichkeiten zur Umlagerung führt. Der Aufbau der Netze wird in Abbildung 6-7 grafisch veranschaulicht.

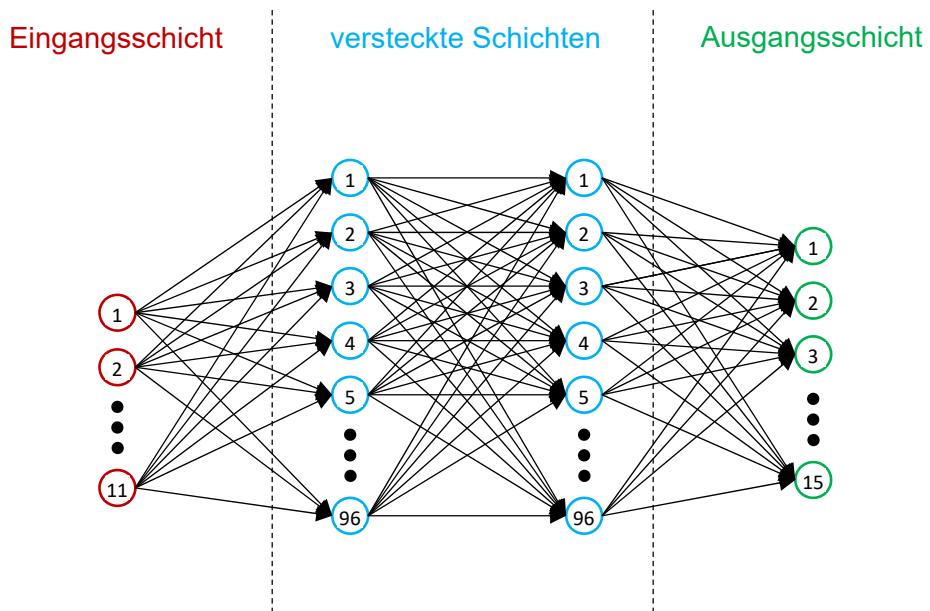


Abbildung 6-7: Fully connected Deep Neural Network mit zwei Schichten

Abschließend werden die Netzwerke in Keras erzeugt und die Methoden zur Berechnung der Fehler sowie die des Optimierers innerhalb der Netzwerke festgelegt.

6.4.5 Verarbeitung der Simulationsdaten in Python

Die Simulationsumgebung sendet per Socket-Schnittstelle ein großes *Bytearray* an den RL-Agenten, das zu Beginn jede Aufgabe des RL-Agenten verarbeitet und in die korrekten Datentypen umwandelt. Da es sich hierbei um eine wiederholende Funktionalität des RL-Agenten handelt und der objektorientierte Ansatz in Python berücksichtigt werden soll, wurde hierfür die Klasse „ConvData“ entwickelt.

Für jede Aufgabe des RL-Agenten wurde eine eigne Funktion zur Verarbeitung der Eingangsdaten implementiert. Da sich diese Methoden in ihrem Aufbau sehr ähnlich sind, wird die Funktionsweise allgemein beschrieben und kann anhand einer expliziten Funktion im Anhang C.9: „ConvData“ nachvollzogen werden.

Ausgehend von den Eingangsdaten wird der erste Eintrag des Arrays nicht beachtet, da dieser im eigens entwickelten Datenprotokoll (vgl. Kapitel 6.3.6) die Aufgabe des RL-Agenten bestimmt. Alle Zahlen sind in Form von 4 Bytes innerhalb des Arrays gespeichert. Diese werden nun mittels Python-Paket „*struct*“ umgewandelt und aus den 4 Bytes wird eine reelle Zahl erzeugt.

Das gesamte Array wird mit einer *For*-Schleife in kleine Pakete von 4 Bytes geteilt und die daraus umgewandelte Zahl in einer Liste gespeichert. Diese Liste erhält der RL-Agent dann von der Funktion zurück.

7 Auswertung und Analyse

Das folgende Kapitel verifiziert und validiert den Simulationsagenten und RL-Agenten anhand von statischen Berechnungen und Simulationsläufen. Anschließend untersucht es die Lernfähigkeit des Agenten und den Einfluss der Hyperparameter auf diese. Die Analyse betrachtet neben den Netzparametern die Fehlerhäufigkeit des neuronalen Netzes und die Einflussgrößen auf den Lernprozess.

7.1 Verifikation und Validierung

Für die Verifizierung des in Kapitel 6 beschriebenen Beispielmodells werden stichprobenartige Tests und grafische Analysen durchgeführt. Diese Tests verifizieren und veranschaulichen die korrekte Arbeitsweise der wichtigsten Methoden.

Zur Verifizierung der Funktionalität des Agenten muss die Arbeitsweise der korrekten Umlagerung überprüft werden. Dies wird im ersten Schritt mittels Debugging der einzelnen Methoden innerhalb des Simulationsagenten und der Funktionen des RL-Agenten realisiert. Dadurch kann im Einzelschrittverfahren jede Zeile der programmierten Methoden auf deren Funktionsweise getestet werden. Sobald die Methoden fehlerfrei bis zum Ende der definierten Experimente durchlaufen, wird der Agent als ablauffähig betrachtet. Auf die genaue Durchführung dieser Methode wird hier jedoch nicht weiter eingegangen.

Im zweiten Schritt wird die Plausibilität der Simulationsergebnisse geprüft. Hierzu kann anhand einer statischen Berechnung des Durchsatzes der Agent validiert werden. Die drei parallel angeordneten Maschinen besitzen eine mittlere Bearbeitungszeit von ca. 57 Sekunden und eine durchschnittliche Rüstzeit von 29 Sekunden. Damit ergibt sich mit der parallelen Anordnung der Durchsatz durch die Addition der Einzeldurchsätze (siehe Tabelle 7-1).

Station	Verfügbarkeit	Bearbeitungszeit	Rüstzeit	Durchsatz pro Tag
1	100%	57s	29s	1011,89
2	100%	58s	29s	993,79
3	100%	57s	29s	1014,84
Gesamtdurchsatz:				3020,52

Tabelle 7-1: Statische Berechnung des Durchsatzes

Ausgehend von der maximalen Kapazität einer Maschine von ca. 1.515 Stück pro Tag, reduziert sich die Kapazität nach Berücksichtigung der Rüstzeiten auf ca. 1.000 Stück pro Tag. Der sich dadurch ergebende durchschnittliche maximale Gesamtdurchsatz von 3.020,52 Aufträgen pro Tag wird nun mit einem Simulationslauf des Agenten verglichen. Dieser erreicht einen Durchsatz von 3.201,20 bei gleicher Ankunftsrate, was einer Abweichung von 6 %

entspricht (vgl. Tabelle 7-2). Die geringen Abweichungen des Simulationsagenten im Vergleich zur statischen Berechnung sprechen deutlich für die Funktionalität des Simulationsagenten.

Statische Berechnung	Simulation Agent
3.020,52 Stück	3.201,20 Stück

Tabelle 7-2: Validierung anhand der Durchsätze

Um die Funktionsfähigkeit des RL-Agent zu prüfen, wurde der durchschnittlich erhaltene Belohnungswert für eine Aktion des RL-Agenten untersucht. Wie in Kapitel „6.3.4 Die Belohnungsfunktion“ erläutert, wird ein maximaler Belohnungswert pro Simulationsschritt von 40 Punkten für eine rüstoptimierte Umlagerung vergeben. Abbildung 7-1 stellt den gleitenden Durchschnitt der erhaltenen Belohnungen der letzten 100 Simulationsepisoden dar. Insgesamt wurden 15.000 Experimentläufe durchgeführt. Nachdem zu Beginn der Belohnungswert negativ ist, steigt dieser exponentiell auf ca. 35 Punkte an. Dies zeigt, dass der RL-Agent diejenigen Aktionen auswählt, die zu einer rüstoptimierten Reihenfolge führen. Jedoch sinkt der Belohnungswert nach einigen Episoden wieder. Dies ist auf einen Effekt des neuronalen Netzes zurückzuführen, dem sogenannten *overfitting*. Da die detaillierte Analyse dieses Problems nicht Teil der Arbeit ist, wird hierauf nicht näher eingegangen. Abschließend ist festzuhalten, dass somit neben dem Simulationsagenten auch der RL-Agent als verifiziert und validiert gilt.

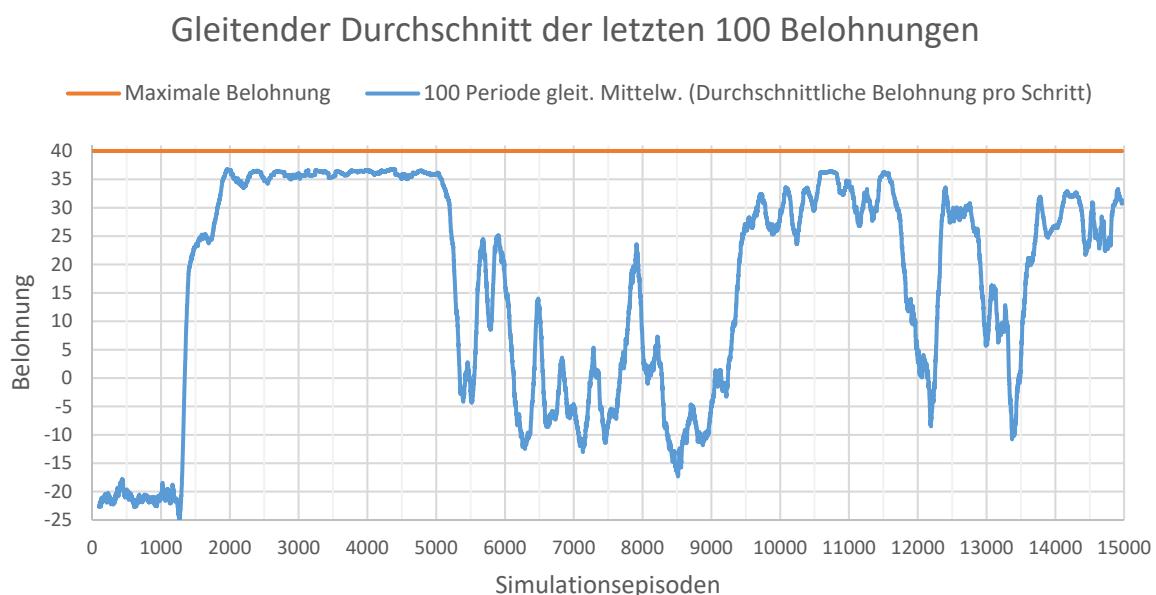


Abbildung 7-1: Durchschnittliche Belohnung des RL-Agenten

7.2 Lernfähigkeit des Agenten

Nachdem die Funktionalität von Simulationsagent und RL-Agent dargelegt wurde, wird als nächstes analysiert, wie gut diese ihre Aufgabe erfüllen. Relevant hierfür ist, wie viele Episoden der Simulationsagent erfolgreich durchlaufen, ohne einen Abbruch zu verursachen. Daher gelten für alle Trainingsläufe die Bestandsgrößen der Eingangspuffer als maximale Anzahl an möglichen Aktionen. Diese werden gegen die tatsächlich durchgeführten Aktionen in Relation gesetzt. Daraus erhält man den Prozentwert der erfolgreichen Schritte pro Experimentlauf. Wie in Abbildung 7-2 zu sehen, sind in den ersten 1.300 Experimenten lediglich 10 % der Schritte erfolgreich. Danach steigt der Anteil erfolgreicher Schritte pro Experiment deutlich an und nähert sich an 100 % an. Jedoch gibt es auch immer wieder Abbrüche. Dies ist auf die Trainingsweise des RL-Agenten zurückzuführen, da durch den Kompromiss zwischen Exploration und Exploitation weiterhin zufällige Aktionen in ca. 1 % der Fälle ausgewählt werden und so Netzfehler verursacht werden können.

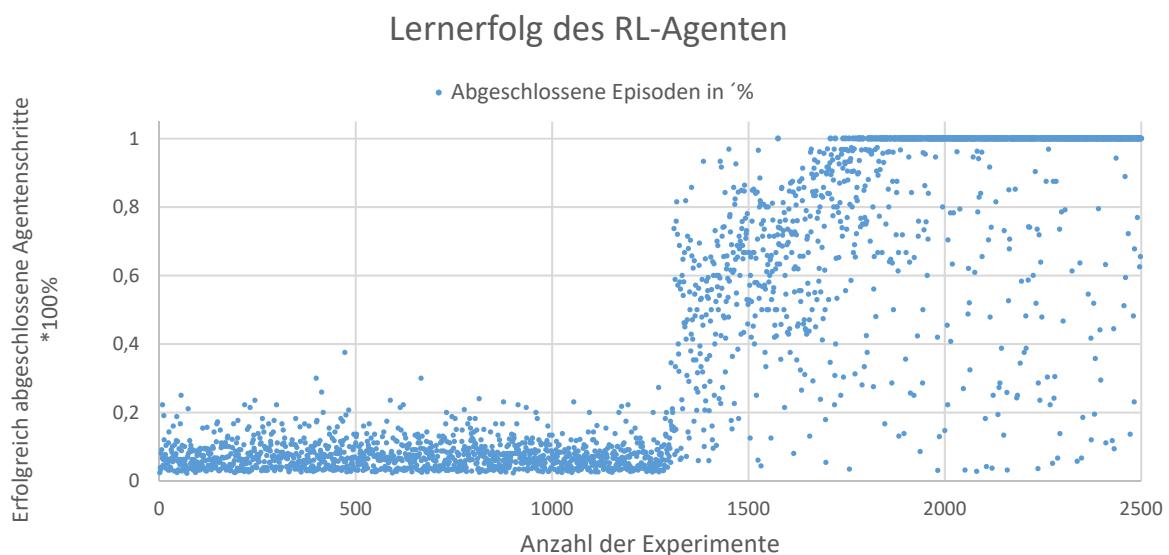


Abbildung 7-2: Lernerfolge des RL-Agenten

Da es sich bei Abbrüchen durch unzulässigen Aktionen nicht unterscheiden lässt, ob diese durch zufällige Entscheidungen des RL-Agenten oder durch eine Fehlentscheidung des neuronalen Netzes verursacht wurden, wird dies genauer analysiert.

7.3 Vorhersagesicherheit des Netzes

Um festzustellen, wie gut das neuronale Netz die Funktion zur Vorhersage einer richtigen Aktion zum Umlagern der Aufträge generalisiert hat, wurden die Daten um den zusätzlichen Parameter „Aktionstyp“ ergänzt. Dieser gibt an, ob die Aktion zufällig getätigter wurde oder vom neuronalen Netz ausgewählt. Das Diagramm in Abbildung 7-3: Fehlgeschlagene Episoden – Netzfehler vs. Zufallsfehler verdeutlicht den prozentualen Anteil erfolgreich abgeschlossener

Simulationsschritte bzw. Umlagerungsvorgänge des Agenten. An diesem Parameter ist erkennbar, wie zu Beginn des Trainings von Episode 0 bis ca. 1.200 der RL-Agent nur zufällige Aktionen auswählt (s. blaue Punkte in Abbildung 7.3). Dies ist notwendig, um den Erfahrungsspeicher zu füllen, bevor das Training beginnen kann. Ab diesem Zeitpunkt beginnt der RL-Agent die Auswahl einer Aktion aus dem Netz hervorzusagen, erkennbar anhand der roten Punkte im Diagramm. Ebenfalls zu erkennen ist, dass die Häufigkeit der Netzvorhersagen zunimmt. Dies wird durch das *Epsilon-Greedy*-Verfahren bedingt. Zufällige Entscheidungen nehmen ab und Vorhersagen des Netzes nehmen zu. Komplett beendete Experimentläufe werden durch grüne Punkte dargestellt. Sobald die Gewichte des Netzes so gut angepasst sind, dass nur noch wenige Episoden fehlschlagen, reduzieren sich auch die Fehlvorhersagen des Netzes (rote Punkte). Es ist jedoch deutlich zu erkennen, dass im späteren Verlauf ab Episode 2.000 dennoch Fehler des Netzes stattfinden. Eine zufällige Auswahl einer Aktion wird konstant bei ca. 1 % getroffen. Daher führt eine Zufallsauswahl auch regelmäßig zu Abbrüchen der Episoden. Dies stärkt die Annahme, dass der RL-Agent eine längere Trainingsphase benötigt, um mehr Erfahrungen sammeln zu können.

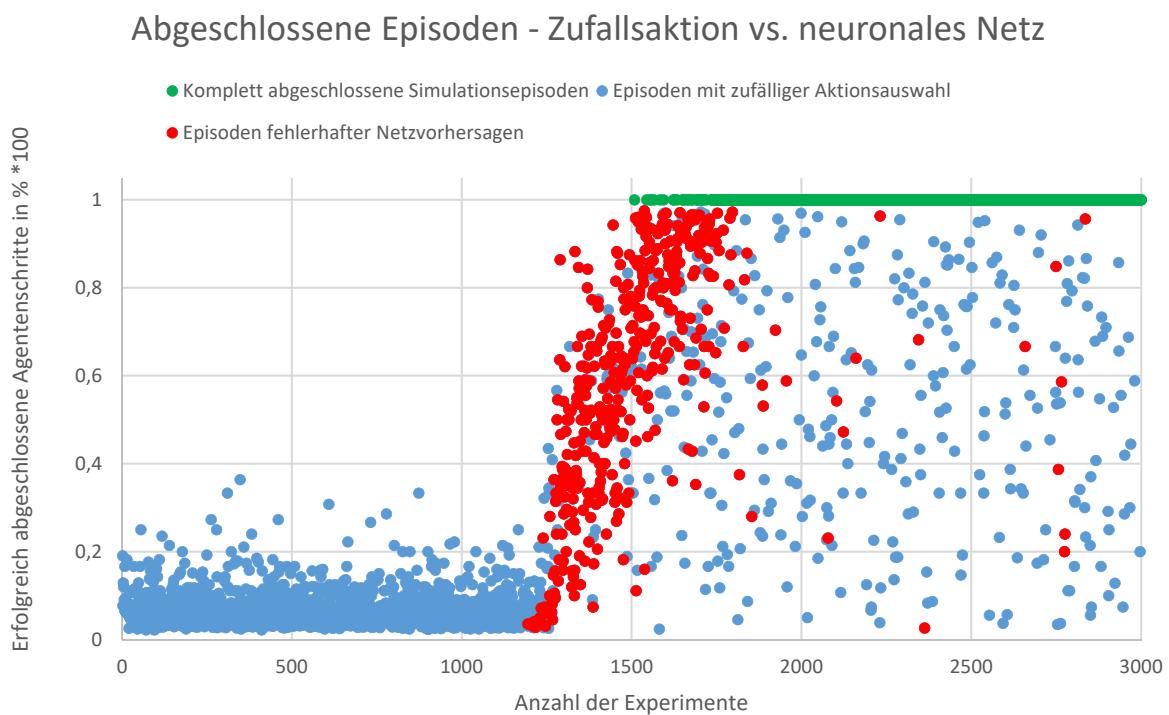


Abbildung 7-3: Fehlgeschlagene Episoden – Netzfehler vs. Zufallsfehler

Um diese Annahme zu prüfen, wird der Trainingsumfang deutlich erhöht auf 15.000 Simulationsläufe. Die Simulationsläufe werden durch die zufällige Experimenterzeugung des Experimentverwalters in Plant Simulation erstellt. Der zuvor durchgeführte Trainingslauf wird wiederholt und gibt in Abbildung 7-4 zu erkennen, dass durch den erhöhten Trainingsumfang die vorhergesagten Aktionen des neuronale Netzes nicht besser werden. Es gibt immer noch

vereinzelte Fehlentscheidungen des Netzes nach 15.000 Experimenten. Nach ca. 4.000 Experimenten wird das Netz wieder schlechter und die Qualität des Netzes schwankt. Bei Experiment 7.000 – 8.000 versagt das Netz sogar komplett und approximiert die meisten Schritte fehlerhaft. Wie bereits zuvor erwähnt, wird dieser Effekt als *overfitting* bezeichnet und im Rahmen dieser Arbeit nicht weiter untersucht.

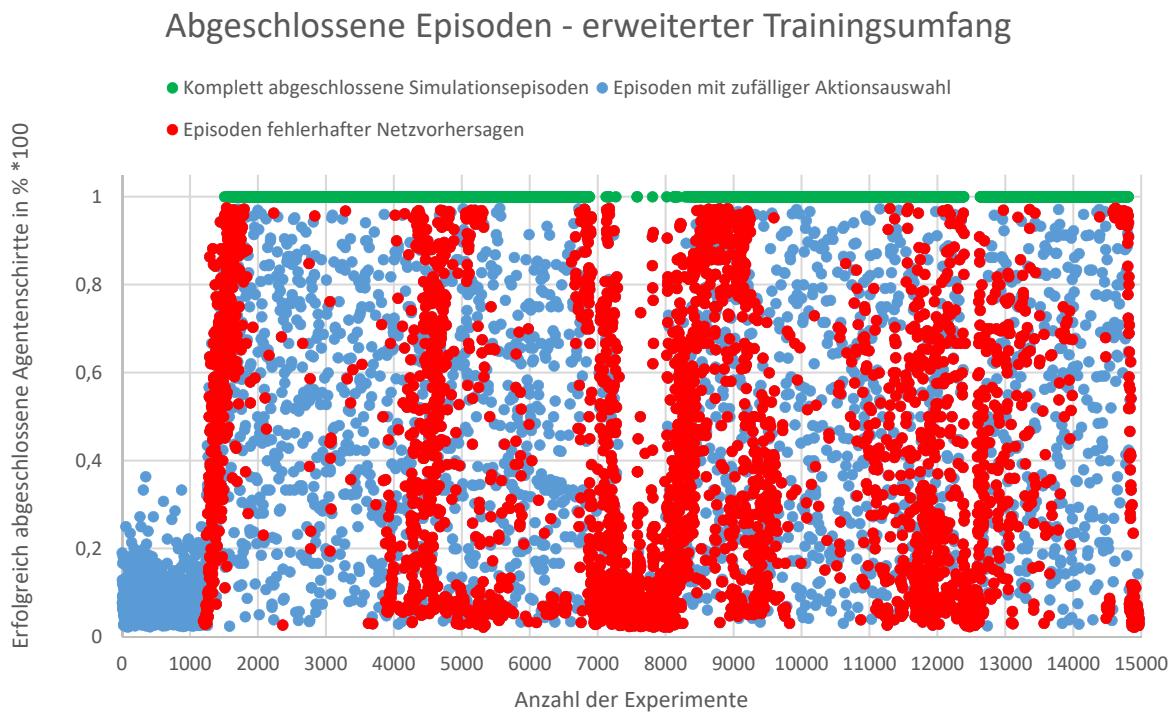


Abbildung 7-4: Vorhersagequalität mit erhöhtem Trainingsumfang

Um in einem weiteren Schritt der Untersuchung die zufällige Auswahl einer Aktion des RL-Agenten zu unterdrücken und die Aktionswahl ausschließlich dem neuronalen Netz zu überlassen, wurde der Hyperparameter Epsilon über das *Epsilon-Greedy*-Verfahren auf 0 heruntergesetzt. Dadurch gibt es keine zufälligen Aktionen des Agenten mehr. Abbildung 7-5 stellt dies dar. Auch ist hier zu erkennen, wie das neuronale Netz zu Beginn immer besser die korrekten Aktionen auswählt, um einen Simulationslauf komplett zu beenden. Jedoch steigen hier nach einigen Läufen die Fehlentscheidungen des Netzes ebenso bis bei Experiment 11.000 fast ausschließlich Fehlentscheidungen getroffen werden. Dieses Verhalten könnte anhand genauerer Netzanalysen untersucht und anhand von Methoden zur Eliminierung des *overfitting* in einer weiteren Arbeit optimiert werden.

Lernqualität

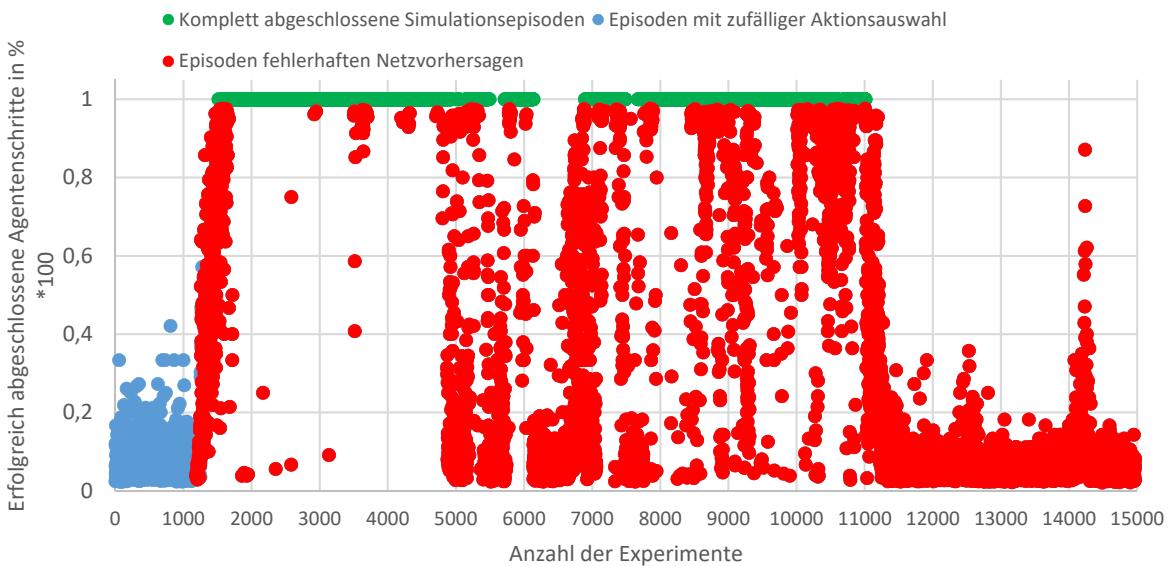


Abbildung 7-5: Vorhersagequalität ohne Zufallsauswahl

Bei Betrachtung des gleitenden Durchschnitts der korrekten Vorhersagen des Netzes über die letzten 100 Experimente ist zu erkennen, dass die Vorhersage des Netzes nur in ca. 98% der Fälle korrekt ist. Dies ist zudem lediglich zwischen den Experimentläufen 2.100 – 5.000 zu beobachten. Anschließend fällt die Vorhersagegenauigkeit wieder deutlich und bricht komplett ein. Diese Ungenauigkeit muss bei einem Benchmark des Agentennetzes berücksichtigt werden.

Korrekt getroffene Netzvorhersagen

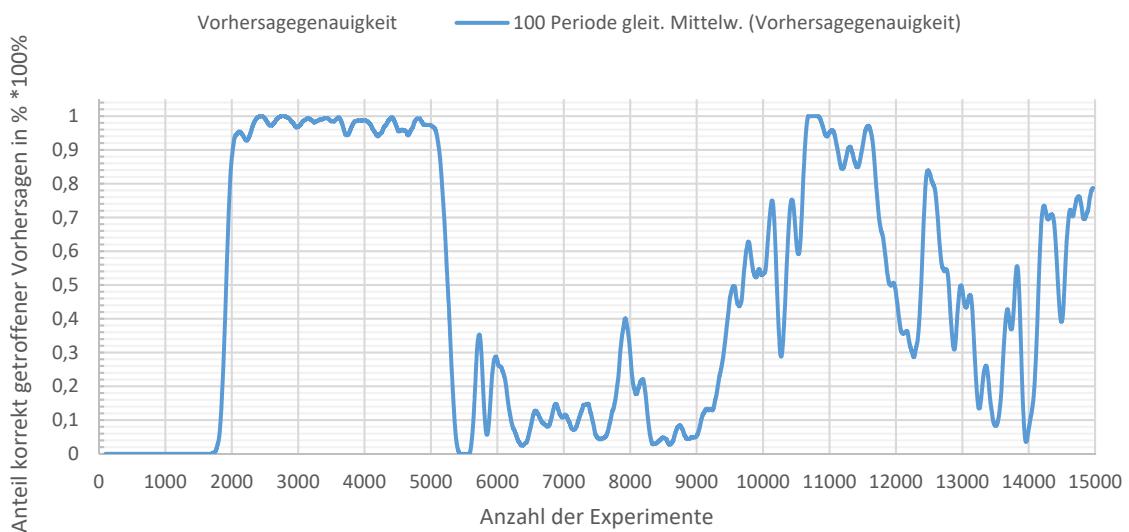


Abbildung 7-6: Korrekt getroffene Netzvorhersagen

7.4 Erprobung und kritische Betrachtung des lernenden Agenten

Um den trainierten Agenten nun für die Anwendung in einem Simulationsmodell vorzubereiten und anhand anderer Entscheidungsverfahren vergleichbar zu machen, muss dieser zunächst erprobt werden. Der Agent soll selbstständig in dem Simulationsmodell aus Kapitel 6.1 die Entscheidung über die Auftragsreihenfolge treffen. Hierfür wird das Training des neuronalen Netzes nach dem Erreichen eines guten Qualitätsniveaus abgebrochen und gespeichert. Dieses Netz wird nun vom Agenten genutzt, um eine Umlagerungsentscheidung zu treffen. Dabei findet kein Training des Netzes mehr statt. Die Bestimmung des Qualitätsniveaus wurde bei Experiment 4.500 festgelegt, da bis zu diesem Zeitpunkt das Netz nur minimale Fehlentscheidungen getroffen hat und der Anteil der Fehlentscheidungen ab diesem Zeitpunkt wieder ansteigt. Wie bereits in Kapitel 7.3 beschrieben, trifft das Netz trotz hohem prozentualen Anteil korrekt getroffener Entscheidungen dennoch zu einem kleinen Anteil Fehlentscheidungen, was beim Test des Agenten berücksichtigt werden muss und anhand zweier Lösungsansätze implementiert wird.

Im ersten Ansatz wurde für den Fall einer Fehlentscheidung des neuronalen Netzes eine Wartezeit eingeführt, die der Simulationsagenten verstreichen lässt, bis er eine neue Anfrage an das Netz sendet. Wählt der Agent eine unzulässige Aktion aufgrund einer Fehlinterpretation des neuronalen Netzes aus, wartet das Simulationsmodell für eine gewisse Zeitspanne, bevor eine erneute Abfrage des neuronalen Netzes gestartet wird. Dies soll dem Simulationsmodell ermöglichen, in der Zeit voranzuschreiten und eventuell neue Auftragszugänge in den Puffern oder fertiggestellte Aufträge auf den Arbeitsstationen zu erfassen.

Es zeigt sich, dass während des Simulationslaufs einzelne Arbeitsstationen dennoch keine Aufträge erhalten und sich dadurch deren Wartezeit erhöht, was zu einem reduzierten Durchsatz führt. Dies ist auf eine Fehlentscheidung des Agenten zurückzuführen. Bei wiederholten Netzabfragen konnte die Fehlentscheidung trotz berücksichtiger Wartezeit nicht ausgesetzt werden und der Agent kommt zu einem Stillstand.

Daher wurde in einem weiteren Schritt der Ansatz der Wartezeit mit einer Zufallsentscheidung über eine Umlagerung ersetzt. Sobald der Agent eine in der Simulationsumgebung unzulässige Aktion auswählt, tritt die Zufallsauswahl einer Aktion in Kraft. Diese wählt einen Auftrag aus einem vollen Puffer aus und lagert diesen auf eine zufällig freie Station um. Dies führt jedoch nach sehr kurzer Simulationszeit ebenfalls zu einem Abbruch des Agenten.

Trotz umfangreicher Tests und Experimenten führte die Fehlersuche zu keinem befriedigenden Ergebnis. Aufgrund dieser Problematik ist ein Benchmark des Agenten gegenüber anderen Entscheidungsverfahren, wie einer Heuristik, zum aktuellen Zeitpunkt

nicht sinnvoll und wurde an dieser Stelle nicht weiterverfolgt. Daher wird im Folgenden der Agent nochmals kritisch erörtert und ein möglicher Lösungsansatz aufgezeigt.

Aufgrund der soeben beschriebenen Problematik der ungenauen Vorhersagegenauigkeit des Netzes kann mit dem prototypischen Aufbau des Agentenkonzeptes zwar die Lernfähigkeit des Agenten verdeutlicht werden, jedoch ist ein Einsatz in einem Simulationsmodell noch nicht möglich. Auch bei deutlich erhöhtem Trainingsumfang konnte die Netzqualität nicht signifikant verbessert werden. Daher muss der in der vorliegenden Arbeit umgesetzte Ansatz des maschinellen Lernens für den praktischen Einsatz weiterentwickelt und verbessert werden.

Ein besonders kritischer Punkt des Ansatzes ist der Aufbau des Lernverfahrens in Verbindung mit der Simulation. Wird eine Fehlentscheidung des Netzes während des Trainings getroffen, wird das Simulationsexperiment abgebrochen. Dies tritt ein, wenn der Agent einen Auftrag aus einem leeren Puffer entnehmen möchte oder der Agent einen Auftrag auf eine bereits belegte Station umlagern möchte. Daher sind auch in den Trainingsdaten Situationen wie z. B. leere Pufferplätze unterrepräsentiert. So zeigt der Agent bei Szenarien mit überwiegend leeren Puffern Schwierigkeiten, einen korrekten belegten Puffer auszuwählen. Für eine Lösung gibt es zwei Wege: Entweder kann das Verhalten des Simulationsagenten so abgeändert werden, dass dieser eine unzulässige Aktion zulässt und hierfür eine Bestrafung erhält, jedoch den Simulationslauf des Experiments nicht abbricht. Eine weitere Lösung könnte eine erhöhte Anzahl der Beobachtungen pro Experiment sein. Jedoch deutet die in Kapitel 7.3 durchgeführte Analyse, mit einer Anzahl von 15.000 Experimenten an, dass die Anzahl der Beobachtungen hierfür deutlich erhöht werden müsste. So kann der Agent bestimmte Situationen mehrfach durchspielen, bis er eine geeignete Lösung gefunden hat. Das ermöglicht dem Agenten, das neuronale Netz auch auf selten eintretende Situationen zu trainieren.

Außerdem spielt eine Rolle, welches Verhalten des Agenten in der Simulation notwendig ist, um auf fehlerhafte Netzvorhersagen richtig zu reagieren. Abschließend ist festzuhalten, dass die Funktionalität und die Lernfähigkeit des entwickelten Konzeptes nachgewiesen wurden, jedoch der Einsatz dieses prototypischen Agenten in einem Simulationsmodell momentan nicht möglich ist.

8 Zusammenfassung und Ausblick

Aufgrund der zu Beginn der Arbeit beschriebenen Entwicklungen in der Produktion und Fertigung hin zu hoch automatisierten, dezentralen, wandlungsfähigen und flexibel vernetzten Logistik- und Produktionsstrukturen, stehen die Unternehmen vor der Herausforderung, diese Paradigmen anhand neuer Materialflussteuerungskonzepte umzusetzen. Die Dezentralisierung erfordert lokale, intelligent handelnde Einheiten, die modular einsetzbar sind. Dafür stellt sich der agentenbasierte Ansatz unter Verwendung neuronaler Netze als besonders relevant heraus. Gleichzeitig erhöht sich das Bedürfnis, Verfahren und Abläufe zur Steuerung intelligenter, dezentraler Logistik- und Fertigungssysteme mittels Simulation von Materialflüssen in der Planungsphase abzusichern.

Infolge dessen wurde die Zielstellung dieser Arbeit formuliert, ein Konzept zum simulationsbasierten Einsatz des maschinellen Lernens für die agentenbasierte Steuerung von Materialflüssen zu entwickeln und prototypisch umzusetzen. Ziel war es, einen Prozess zu entwerfen, der maschinelles Lernen in der ereignisdiskreten Simulation von Materialflüssen ermöglicht. Das Konzept wird am Beispiel der Steuerung einer Auftragsreihenfolge für Produktionsressourcen erprobt und die Lernfähigkeit des Agenten nachgewiesen.

Im ersten Schritt werden die grundlegenden Verfahren des maschinellen Lernens erläutert, um ein geeignetes Verfahren für die agentenbasierte Steuerung herauszuarbeiten. Hierfür ist das bestärkende Lernen eines Agenten als geeignet identifiziert und die relevanten Eigenschaften näher beschrieben worden.

Auf Basis der grundlegenden Funktions- und Arbeitsweise eines bestärkenden Agenten wird das Konzept für dessen Einsatz in der Materialflusssimulation entwickelt. Daraus entsteht der Ansatz, zwei miteinander interagierende Prozesse, Fertigungssimulation und Lernen einer geeigneten Steuerungsstrategie, in unterschiedlichen Softwareumgebungen zu implementieren. Der Simulationsagent wird als Bestandteil der Simulationsumgebung definiert und übernimmt die Steuerung der Abläufe des Modells. Der bestärkende Agent nimmt hingegen die Auswahl der Aktionen vor und trifft diese mit Hilfe eines neuronalen Netzes.

Für die Umsetzung des entwickelten Konzeptes wird das kommerzielle, in der Praxis weit verbreitete Simulationswerkzeug Plant Simulation eingesetzt. Der bestärkende Agent mit Steuerungsmöglichkeiten ist in dem Open Source Framework Keras unter Verwendung der Programmiersprache Python umgesetzt. Die Kommunikation und Interaktion dieser beiden Einheiten wird anhand eines WebSocket-Protokolls realisiert.

Nachdem die Abläufe und die Funktionalität beim Training des Simulationsagenten verifiziert und validiert sind, wird die Lernfähigkeit des bestärkenden Agenten untersucht und analysiert.

Dieser erfüllt die Aufgabe, Aufträge aus Eingangspuffern in einer geeigneten, rüstoptimierten Reihenfolge auf die nachgelagerten Fertigungsressourcen umzulagern. Dabei zeigt sich, dass der bestärkende Agent bereits nach wenigen Simulationsläufen einen sehr schnellen Lernfortschritt erzielt. Als Maß für den Lernfortschritt werden die erfolgreich abgeschlossenen Umlagerungen herangezogen. Nun stellt sich die Frage, wie gut der Agent die Aufgabe zur Bildung einer geeigneten Reihenfolge erlernt. Hierfür wird der gleitende Durchschnitt der 100 zuletzt erhaltenen Belohnungen je Umlagerungsschritt untersucht. Das Resultat ergibt eine mittlere Belohnung von 38 Punkten von maximal möglichen 40 Punkten, was einer Vorhersagegenauigkeit von 95 % entspricht. Allerdings bricht die Vorhersagegenauigkeit nach einigen Schwankungen komplett ein, wenn das neuronale Netz kontinuierlich weiter trainiert wird.

Mit dem trainierten neuronalen Netz des Agenten ist nun eine Erprobung in einem Simulationsmodell durchgeführt worden. Dabei kann der Agent durch das bestärkende Lernen das neuronale Netz so trainieren, dass dieses eine Taktik zur rüstoptimalen Umlagerung der Aufträge entwickelt, jedoch für den Einsatz in einem Simulationsmodell nicht genug ausgereift ist. Fehlentscheidungen des neuronalen Netzes führen zu wiederholten Abbrüchen der Simulationsläufe und lassen eine sinnvolle Bewertung sowie Vergleich des Agenten gegenüber anderen Entscheidungsverfahren nicht zu.

Mit dem in der vorliegenden Arbeit entwickelten Konzept ist ein Prozess geschaffen worden, der den Einsatz von simulationsbasierten maschinellem Lernen zur agentenbasierten Steuerung von Materialflüssen ermöglicht. Es wurde gezeigt, dass anhand dieses Ansatzes der Agent in einer Simulationsumgebung selbstständig eine optimale Steuerungsstrategie unter Verwendung von neuronalen Netzen erlernen kann.

Um dieses Konzept bei komplexeren Simulationsmodellen einzusetzen, bedarf es einer Weiterentwicklung der bestehenden Funktionalitäten. Dabei ist die konsequente Standardisierung der einzelnen Funktionen des Simulationsagenten zu priorisieren. Besonders die Schnittstelle via WebSocket erfordert eine vereinfachte und robuste Lösung, wodurch die Flexibilität des Simulationsagenten erhöht werden kann. Dies ermöglicht ebenso eine weniger komplexe Datenverarbeitung innerhalb des lernenden Agenten. Außerdem kann durch eine standardisierte und automatisierte Erfassung der Umwelt der Programmieraufwand bei der Implementierung des Simulationsagenten in einer fremden Umgebung verringert werden.

Des Weiteren sollte aufgrund der Sensibilität des gewählten Lernverfahrens gegenüber den Hyperparametern das Training des lernenden Agenten durch experimentelle Untersuchungen optimiert oder weitere Lernverfahren in Betracht gezogen werden. Ebenso bleibt das Problem

des *overfittings* bei zu langen Lernzeiträumen im Rahmen dieser Arbeit unberücksichtigt. Um den Agenten während des gesamten Simulationszeitraums ohne Verschlechterung der Vorhersage des Netzes lernen zu lassen, sind Methoden zur Reduzierung der Komplexität notwendig.

Anhang A: Übersicht der technischen Entwicklungsumgebung

Simulationsagent	Lernender Agent
Plant Simulation 15.1.4	Anaconda 3.8 64-bit
Plant Simulation Interface Package	Python 3.7.6 64-bit
	Keras 2.3.1
	Virtuelle Umgebung „PlantSimENV“ (digitaler Anhang)

Anhang B: Plant-Simulation-Programmcode

Anhang B.1: Die Schrittfunktion mStep

```
1  -- diese Methode wird von der "init" des Modells ausgelöst
2  --wenn Teile in Puffer sind, beginnt die Simulation
3
4  --Deklaration der Variablen
5  --lokale Variablen für die Eingangs- und Ausgangsobjekte des Agenten
6  var in, out:integer
7  --Variable für die zwischenspeicherung der Belohnungstabelle
8  var actTable:table
9
10 --Zuweisung der Untertabelle der Belohnungen für die aktuelle Episode-----
11 actTable := tblMemory_Rewards[1,SimEpisode]
12 -----
13
14 --Speichere die Anzahl möglicher Schritte des Agenten ab vor jedem Experimentlauf-----
15 For var p : 1 to tblInput_objects.Ydim
16   possible_actions := possible_actions + tblInput_objects[1,p].numMu
17 next
18 -----
19
20 --Endlosschleife solange alle Puffer vor dem Agenten mit Aufträgen befüllt sind
21 while ~.BufferA.occupied = true OR ~.BufferB.occupied = true OR ~.BufferC.occupied = true OR ~.BufferD.occupied = true OR ~.BufferE.occupied = true
22
23   --Zählervariable für alle erfolgreichen Schritte des Agenten-----
24   AgentEpisode += 1
25
26   |
27   print("waiting for machines not occupied anymore")
28
29   --warte bis eine der Nachfolger des Agenten frei ist für den nächsten Auftrag -> führe Lauf fort---
30   waituntil ~.M1.empty = true OR ~.M2.empty = true OR ~.M3.empty = true prio 1
31
32
33   --schreibe den aktuellen Zustand der Umgebung in eine Tabelle-----
34   mWriStates
35
36
37   --hole eine Aktion via Socket vom RL-Agenten (der Zustand wird übertragen aus der Tabelle)-----
38   --Wenn der Agent nicht trainieren, sondern nur das NN befragen soll
39   If predict = true
40     mSocket_predict_action
41   else
42     --hole die Aktion für das Training des Agenten (Exploration/Exploitation)
43     mSocket_get_action
44   end
45
46
47   --halte die Simulation im EventController an um auf eine Rückmeldung des RL-Agenten zu warten
48   eventController.stop
49   print("waiting for action")
50
51
52   --warte bis die Variable act_received wahr ist wenn der RL-Agent eine Aktion zurückgegeben hat---
53   waituntil act_received = true
54   act_received := false --setze Variable zurück für den nächsten Schritt
55   eventController.start --starte den Eventcontroller wieder
56
57
58   --Suche nach den Eingangs- und Ausgangsobjekten in Abhängigkeit der gewählten Aktion-----
59   in := tblActionSpace[1,action] --input object
60   out := tblActionSpace[2,action] --output object
61
62
63   --schreibe die Aktion in eine Tabelle-----
64   tblMemory_Actions.writeRow(1,tblMemory_Actions.Ydim+1, action, action_type, in, out)
65
66
67   --führe die Aktion aus und berechne die Belohnung für die Aktion-----
68   reward := mDoAction(in, out)
69
70
71   --schreibe die Belohnung in eine Tabelle-----
72   actTable.writeRow(1,actTable.Ydim+1,action,reward)
73
74
75   --schreibe den Folgezustand (neuer Zustand nach Aktion) in eine Tabelle-----
76   mWriNewStates
77
78
79   --gebe die Zustände und Aktionen an den RL-Agenten für das Training und speicher Sie-----
80   If predict = false
81     mSocket_write_memory --remember in Erfahrungsspeicher
82   end
83
84
85   --if done=true (impossible action chosen -> simempisode ends) quit the loop and start all over--
86   if done = true
87     --kommentierte Befehle sind nur für Training auskommentiert (Zum Testen Kommentarzeichen entfernen)
88     --if predict = false
89       exitloop
90     --else
91       --messageBox("prediction failed", 1, 1)
92       --waituntil ~.BufferA.occupied = true OR ~.BufferB.occupied = true OR ~.BufferC.occupied = true OR ~.BufferD.occupied = true OR ~.BufferE.occupied = true
93       --end
94     end
95   end
96
97
98   --Schreibe die Statistikwerte am Ende eines Experimentlaufs in Tabellen und beende den Simulationslauf.
99   tblMemory_Rewards[2,tblMemory_Rewards.Ydim] := total_reward
100  mStatistics
101  endsim
102
```

Anhang B.2: Die Belohnungsfunktion des Simulationsagenten

```

1  --Übergebenen Parameter von der DoAction-Methode (Eingangs- und Ausgangsobjekte des Agenten)
2  param inputObject, outputObject:object -> real
3
4  --Deklaration der Variablen-----
5  var reward:real
6  var procTime:time
7  var factorProcTime:real
8
9
10 -- Wenn das Eingangsobjekt leer ist oder das Ausgangsobjekt belegt -> Bestrafung -100 Punkte-----
11 if inputObject.empty = true OR outputObject.occupied = true
12   --Bestrafung
13   reward -= 100
14 else
15   --Ansonsten vergabe Belohnung für eine zulässige Aktion
16   reward += 25
17
18 --Wenn die Aktion ideal ist, da Rüstoptimiert vergabe erhöhte Belohnung
19 if inputObject.empty = false AND outputObject.isSetupFor = inputObject.cont.name
20   reward += 15
21 else
22   --Wenn nicht gebe keine zusätzliche Belohnung
23   reward -= 0
24 end
25 end
26
27 --berechne noch die Gesamtbelohnung-----
28 total_reward += reward
29
30
31 --Rückgabe der Belohnung an die Aufrufende Methode "DoAction-----
32 return reward
33

```

Anhang B.3: Ausführen der gewählten Aktion

```

1  /* method for performing the chosen action and calculating the reward for the action*/
2
3  --Übergabeparameter der Eingangs- und Ausgangsobjekte
4  param in, out:integer -> real
5
6  --Deklaration der Variablen-----
7  var inputObject, outputObject:object
8  var reward:real
9  var step:boolean
10
11
12 --readout the objects for the action-----
13 inputObject := tblInput_objects["input objects", in]
14 outputObject := tblOutput_objects["output objects", out]
15
16
17 --calculate the reward for the chosen action-----
18 reward := mCalculateReward(inputObject, outputObject)
19
20
21 --perform action and move parts prec -> succ only when possible-----
22 if inputObject.empty = false AND outputObject.occupied = false
23   inputObject.contentsList[1,1].move(outputObject)
24   done := false
25 else
26   --Aktion ist nicht zulässig -> Episode endet (würde Fehler in PlantSim auslösen)
27   done := true
28   agent_failed := true
29   --letzte Aktion war nicht erfolgreich
30   AgentEpisode -= 1
31 end
32
33
34 --Wenn alle Puffer geleert wurde, vergabe 100 Punkte extra für erfolgreiches Ende-
35 if ~BufferA.empty = true AND ~BufferB.empty = true AND ~BufferC.empty = true
36   AND ~BufferD.empty = true AND ~BufferE.empty = true AND total_reward > 0
37   total_reward += 100
38   done := true
39 end
40
41
42 --Rückgabe der Belohnung aus der Funktion mCalculateReward an die mStep Schrittfunktion
43 return reward
44

```

Anhang B.4: Datentransfer von Plant Simulation via Socket

```

1  --Methode zum senden der Zustände, Aktion, Belohnungen, Folgezustände
2
3  --Deklaration der Variablen-----
4  var state:integer[]
5  var action:integer[]
6  var reward:integer[]
7  var new_state:integer[]
8  var state_str:string
9  var xReal:real
10 var task:integer
11 var xInt:integer
12 -----
13
14 --Erstellung des Bytearrays zum Transfer zum RL-Agenten-----
15 --bestimme Aufgabe=2 : steht für Training des RL-Agenten und hänge diese an das Array
16 task := 2
17 state.append(task)
18 --hänge einen Wert für das Beende der Episode an-----
19 --wenn die Episode beendet wurde done=true hänge Zahl 1 an, wenn nicht hänge Zahl 0 an
20 if done = true
21   state.append(1)
22 else
23   state.append(0)
24 end
25 -----
26
27 state.append(tblMemory_States.Xdim) --number of state variables for state
28 state.append(1)--number of actions passend (always 1)
29 state.append(1)--number of rewards passend (always 1)
30 state.append(tblMemory_NewStates.Xdim) --number of state variables for newstate
31 state.append(1)--number of total_rewards passed
32 state.append(1)--number of sim_Episodes passed
33 state.append(1)--number of sim_Episodes passed
34 -----
35
36 --hole Zustand von Tabelle und hänge an Array an-----
37 for var i := 1 to tblMemory_States.Xdim
38   xReal := tblMemory_States[i, tblMemory_States.Ydim]
39   state.appendValueOfType(xReal, "real32")
40 next
41 -----
42
43 --hole Aktion von Tabelle und hänge an Array an-----
44 xReal := tblMemory_Actions[1, tblMemory_Actions.Ydim] - 1 -- subtract 1 zero-based action list in python / Plantsim is 1-based
45 state.appendValueOfType(xReal, "real32")
46 -----
47
48 --hole Belohnung von Tabelle und hänge an Array an-----
49 xReal := tblMemory_Rewards[1,SimEpisode][2, tblMemory_Rewards[1,SimEpisode].Ydim]
50 state.appendValueOfType(xReal, "real32")
51 -----
52
53 --hole Folgezustand von Tabelle und hänge an Array an-----
54 for var i := 1 to tblMemory_NewStates.Xdim
55   xReal := tblMemory_NewStates[i, tblMemory_NewStates.Ydim]
56   state.appendValueOfType(xReal, "real32")
57 next
58 -----
59
60 --hole Gesamtbelohnung pro SimEpisode und hänge an Array an-----
61 state.appendValueOfType(total_reward, "real32")
62 -----
63
64 --hole SimEpisode und hänge an Array an-----
65 if SimEpisode > 254
66   state.appendValueOfType(SimEpisode, "real32") --pass numer of AgentEpisode
67 else
68   state.appendValueOfType(SimEpisode, "real32") --pass numer of AgentEpisode
69 end
70 -----
71
72 --sende Array an RL-Agent via Socket
73 Socket_Server.writearray(0, state)
74 -----

```

Anhang B.5: Empfangen der Daten aus Python

```

1  /*Methode für den Empfang der Daten des RL-Agenten. Es gibt zwei Aufgaben
2   für PlantSim, abhängig von den Empfangenen Daten:
3   1. print Informationen über den Status und die erledigte Aufgabe
4   2. Aktionstyp, den PlantSim durchführen muss (Zufall oder Vorhersage durch NN)*/
5
6  --Übergebenen Parameter
7  param adress:integer, receivedData:string
8
9  --Deklaration der Variablen-----
10 var task, data, dataOut, splitData, msg, rp:string
11 var splitPos:integer
12 var allMsg:list[string]
13 -----
14
15 --Initialisierung der Nachrichtenliste & Zuweisung der erhaltenen Daten-----
16 allMsg.create --initialize msg list
17 splitData := receivedData --assign data to new variable
18 -----
19
20 --suche nach dem Ende einer Nachricht im Array und teile diese auf-----
21 while pos("/", splitData) /= 0
22   splitPos := pos("/", splitData) --Position des Endmarkers
23   dataOut := omit(splitData,splitPos,strlen(splitData)) --schneide Daten bis zum ersten Marker
24   allMsg.append(dataOut) --füge alle Informationen an Liste an
25   splitData := omit(splitData,1,splitPos) --schneide Informationen vom dem Hauptdatenblock
26 end
27 -----

```

```

28  --forme die erhaltenen Informationen in Aufgabe (Task) und Daten (Data)
29  --task: t=text, a=action
30
31  For var j := 1 to allMsg.Dim
32      msg := allMsg[j] --hole einzelne Information aus Liste
33      task := omit(msg,2,strlen(msg)) --schneide alle Daten der Liste weg um nur die Aufgabe zu erhalten
34      data := omit(msg,1,1) ---schneide erste Eintrag der Nachricht weg um Daten zu bekommen
35      rp := allMsg[2] --schneide ersten zwei Einträge weg um Aktionstyp zu erhalten (random or prediction)
36  --Text Information
37  If task = "t"
38      If data = "connection enabled"
39          connection_enabled := true --setze Variable auf true um fortzufahren
40      Elseif data = "Agent initialized"
41          DQN_created := true --warte bis Agent DQN-Netzwerk erzeugt hat
42      End
43      Print data
44
45  --Aktionen
46  Elseif task = "a"
47      --transformiere Aktion String in Integer für PlantSim
48      action := str_to_num(data) + 1 --addiere 1 zuer Listenindex, da python zero-based [0-14] Plant Sim ist 1-based [1-15]
49      action_type := str_to_num(rp)
50      act_received := true --setze Variable auf true, zum forfahren der mStep Schrittfunction
51  End
52 Next
53 -----

```

Anhang C: Python-Programmcode

Anhang C.1: Python-Pakete für den RL-Agenten

Benötigte Pakete für den RL-Agenten

```
In [ ]: #####  
# Die Datei "agent.py" ist ein Software-Agent (RL-Agent) der den Lernprozess steuert #  
# Nachdem alle benötigten Pakete importiert wurden, wird die Klasse "Agent" erzeugt #  
# Die "main" Funktion ist eine Endlosschleife, wartet auf Daten des Simulationsagenten #  
# Sie wird wiederholt, bis der Abbruchsbefehl vom Simulationsagenten kommt #  
# Der RL-Agent hat verschiedene Aufgaben zu erfüllen, unterschieden durch If-Abfragen #  
#####  
# Pakete für Debugging  
import logging  
# Pakete für die TCP Verbindung zu Plant Simulation (Socket)  
import socket  
import struct  
# Pakete für Berechnungen  
import numpy as np  
import collections  
import random  
# Pakete für das DQN  
from keras.models import *  
from keras.layers import *  
from keras.optimizers import *  
from keras.callbacks import TensorBoard  
#import selbst programmierte Klassen  
from dqn import *  
from convdata import *
```

Anhang C.2: Initialisierung der Agentenklasse

Initialisierungsfunktion des RL-Agenten

```
In [ ]: #####  
# Alle Hyperparameter für das Training und für das neuronale Netz #  
# werden von Plant Simulation übergeben.  
# Funktionen des Agenten: "get action", "train", "replay", "remember",#  
#####  
class Agent:  
  
    def __init__(self, init_data):  
        # Variablen der Umwelt  
        tup = (int(init_data[0][0]),)  
        self.observations = tup  
        self.actions = int(init_data[1][0])  
  
        # Variablen des Agenten  
        self.replay_buffer_size = int(init_data[2][0])  
        self.batch_size = int(init_data[3][0])  
        self.train_start = int(init_data[4][0])  
        self.epsilon = round(init_data[5][0], 4)  
        self.epsilon_decay = round(init_data[6][0], 4)  
        self.epsilon_min = round(init_data[7][0], 4)  
        self.gamma = round(init_data[8][0], 4)  
        self.memory = collections.deque(maxlen=self.replay_buffer_size)  
  
        # DQN Variablen  
        self.state_shape = self.observations  
        self.learning_rate = init_data[9][0]  
        self.hid_layer = int(init_data[11][0])  
        self.act_fkt = "relu"  
        # erzeuge neuronales Netz (policy Netz)  
        self.model = DQN(self.state_shape, self.actions, self.learning_rate, self.hid_layer, self.act_fkt)  
        # 2. neuronales Netz für Experience Replay (Zielnetz)  
        self.target_model = DQN(self.state_shape, self.actions, self.learning_rate, self.hid_layer, self.act_fkt)  
        # Zusätzliche Variablen  
        self.total_rewards = []
```

Anhang C.3: Die Funktion zur Auswahl einer Aktion

Funktion zur Aktionsauswahl

```
In [ ]: # Funktion für die Auswahl einer Aktion Zufall/Vorhersage  
def get_action(self, state):  
    # in epsilon Prozent der Fälle  
    if np.random.rand() <= self.epsilon:  
        # gib eine Zufallszahl im Bereich aller Aktionen (0-14)  
        return np.random.randint(self.actions), 0  
    else:  
        # hole eine vorhergesagte Aktion vom DQN-Model  
        return np.argmax(self.model.predict(state)), 1
```

Anhang C.4: Trainingsfunktion

Trainingsfunktion des RL-Agenten

```
In [ ]: # Funktion für die Trainingsschritte:
# speichere die gewählte Aktion "a" in Zustand "s" und die erhaltene Belohnung "r" endend in Zustand "s'"
def train(self, PSinput):
    sim_episode = 0.0
    done = bool(PSinput[1])
    # wandle erhaltene Daten um und speichere diese in Variablen
    state, action, reward, next_state, total_reward, sim_episode = convert.get_memory(PSinput)
    # formatiere Zustand für das Netzwerk um (8,1) -> (1,8)
    state = np.reshape(state, (1, state.shape[0]))
    next_state = np.reshape(next_state, (1, next_state.shape[0]))
    #sichere Variablen in den Erfahrungsspeicher (Replay Buffer)
    self.remember(state, action, reward, next_state, done)
    # rufe Funktion "replay" um vom vorherigem Schritt zu lernen
    self.replay()

    if done:
        self.total_rewards.append(total_reward)
        mean_reward = np.mean(self.total_rewards[-10:])
        self.target_model.update_model(self.model)
        print("Episode: ", sim_episode, " Total Reward: ", total_reward, " Epsilon: ", self.epsilon, " Ended: ",
              done, " Datenmenge: ", len(self.memory))
```

Anhang C.5: Funktion zur Speicherung der Daten im Erfahrungsspeicher

Funktion zur Speicherung der Daten im Erfahrungsspeicher (Replay Buffer)

```
In [ ]: def remember(self, state, action, reward, next_state, done):
    # Füge aktuellen Zustand, Aktion, Belohnung, Folgezustand dem Überlaufspeicher hinzu
    self.memory.append((state, action, reward, next_state, done))
    # wende die epsilon greedy an: reduziere epsilon bei jedem Schritt
    # bis epsilon min erreicht ist
    if len(self.memory) > self.train_start:
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Anhang C.6: Die Trainingsfunktion zur Aktualisierung der Q-Werte

Trainingsfunktion des RL-Agenten

```
In [ ]: def replay(self):
    # starte Training nur, wenn genügend Daten im Erfahrungsspeicher sind
    if len(self.memory) < self.train_start:
        return
    #ziehe einen Datenauszug aus dem Erfahrungsspeicher
    minibatch = random.sample(self.memory, self.batch_size)
    #speichere diesen in Listenvariablen
    states, actions, rewards, states_next, dones = zip(*minibatch)

    # [s1, s2, s3, s4, s5] ursprüngliches Listendesign
    # np.array([[s1], [s2], ...]) Design erwartet von Keras -> concatenate(numpy array)
    states = np.concatenate(states)
    states_next = np.concatenate(states_next)
    # Vorhersage aus dem live Netz für aktuelle Q-Werte
    # und Vorhersage aus dem Zielnetz für die Q-Werte der Folgezustände
    q_values = self.model.predict(states)
    q_values_next = self.target_model.predict(states_next) #q' und a' Werte
    #aktualisiere Q-Werte für die gerade ausgeführte Aktion des Auszugs
    for i in range(self.batch_size):
        a = actions[i]
        done = dones[i]
        # wenn Schritt zum Ende geführt hat, werden nur die unmittelbaren
        # Belohnungen berücksichtigt, ansonsten werden die Folgezustände
        # diskontiert und aufsummiert
        if done:
            q_values[i][a] = rewards[i]
        else:
            q_values[i][a] = rewards[i] + self.gamma * np.max(q_values_next[i], axis=0)

    # trainiere das Modell mit den Q-Werten und den Zuständen des Datenauszugs
    self.model.train(states, q_values)
```

Anhang C.7: Die Hauptfunktion und der Aufbau der Socket-Verbindung

Die Hauptfunktion des RL-Agenten

```
In [ ]: while True:  
    try:  
        if __name__ == "__main__":  
            logger = logging.getLogger()  
            # starte Verbindung mit Socket zum Datenempfang  
            client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #TCP  
            server_addr = ("127.0.0.1", 30000) #hostname, port  
            client_socket.connect(server_addr)  
            init = True  
  
            # Endlosschleife zum Datenempfang, Datenverarbeitung und erledigen der Aufgaben  
            while True:  
                # sende eine Nachricht an PlantSim bei der Initialisierung  
                if init == True:  
                    init = False  
                    client_socket.send(bytes("t"+"connection enabled/", "utf8"))  
                    print("connection enabled")  
  
                # Socket auf Empfang, wartend auf Dateneingänge  
                PSinput = client_socket.recv(1024)  
  
                # Teste die Verbindung bevor Daten gesendet/empfangen werden  
                if len(PSinput) <= 0:  
                    break  
  
                # erzeuge ein Objekt "convert" von der Klasse ConvData  
                convert = ConvData(PSinput)  
                # extrahiere die Aufgabe aus dem PlantSim Input  
                task = convert.get_task(PSinput)
```

Anhang C.8: Die Aufgaben des RL-Agenten

Die Aufgaben

```
In [ ]:  
    ##### initialisiere den Agenten #####  
    if task == 1:  
        # extrahiere nur die Initialisierungsdaten von PSinput  
        data = convert.get_init_data(PSinput)  
        # erzeuge ein Objekt "agent" der Klasse "Agent" und übergebe die Daten  
        agent = Agent(data)  
        # sende Nachricht an PlantSim (zur Fortführung in PlantSim)  
        client_socket.send(bytes("t"+"Agent initialized"+"/", "utf8"))  
        print("Agent initialized") # Drucke in der Konsole die Nachricht  
    #####  
  
    ### speichere Daten in Erfahrungsspeicher und trainiere Netz ##  
    elif task == 2:  
        agent.train(PSinput)  
    #####  
  
    ##### hole Aktion (Zufall/Vorhersage) von Agent #####  
    elif task == 3:  
        # extrahiere Zustand aus Daten  
        state = convert.get_data(PSinput)  
        # umformen des Tupel von (8,1) nach (1,8)  
        state = np.reshape(state, (1, state.shape[0]))  
        # hole Aktion von Agent  
        action, rp = agent.get_action(state)  
        # sende Aktion an PlantSim ("a" für Aufgabe in PlantSim, "/" als Endmarker für Nachricht)  
        client_socket.send(bytes("a"+str(action)+"/"+str(rp)+"/", "utf8"))  
    #####
```

```

##### hole Aktion (Vorhersage) vom gleadenen Netz #####
elif task == 4:
    # extrahiere Zustand aus Daten
    state = convert.get_data(PSinput)
    # umformen des Tupel von (8,1) nach (1,8)
    state = np.reshape(state, (1, state.shape[0]))
    # hole Aktion von Agent
    action = np.argmax(agent.model.predict(state))
    rp = 1
    # sende Aktion an PlantSim ("a" als Aufgabe in PlantSim, "/" als Endmarker für Nachricht
    client_socket.send(bytes("a"+str(action)+"/"+str(rp)+"/", "utf8"))

#####
##### sichere Agentennetz (model) #####
elif task == 5:
    path = PSinput
    agent.model.save_model("Pfad/dqn_PlantSimAgent.h5")
    print("Model saved")

#####
##### lade Agentennetz (model) #####
elif task == 6:
    path = PSinput
    agent.model.load_model("Pfad/dqn_PlantSimAgent.h5")
    print("Model loaded")

#####
##### beende Socket-Client Verbindung zum Simulationsagenten #####
elif task == 8:
    break
#####

```

```

#####
##### schließe Client-Socket #####
if task == 8:
    # sende Nachricht an PlantSim zum beenden der Verbindung
    client_socket.send(bytes("t"+"connection to socket successfully closed!"+"/", "utf8"))
    # schließe das Socket
    client_socket.close()
    break
#####

except Exception:
    logger.error("Fatal error in main loop", exc_info=True)

```

Anhang C.9: ConvData

```
In [ ]: import struct
import numpy as np

#Klasse um das Bytearray in eine Liste mit float Werten zu konvertieren
class ConvData:
    def __init__(self, PSinput):
        self.PSinput = PSinput

    #Funktion um die Aufgabe des RL-Agenten zu extrahieren und abzufragen
    def get_task(self, PSinput):
        #Aufgabennummer ist immer die erste Zahl im Array
        self.task = self.PSinput[0]
        return self.task

    #Funktion um die Daten zur Initialisierung zu formatieren und auszulesen
    def get_init_data(self, PSinput):
        #Parameter werden auf initialzustand gesetzt
        self.rawdata = PSinput[1:]
        self.intNum = 0
        #4 ist die Größe des Datenpaketes für eine Zahl
        self.intNumTo = 4
        self.data = []
        self.arrayLength = len(self.rawdata)
        self.arrayLength = int(self.arrayLength/4) #teile durch 4 für 32bit float
        for i in range(self.arrayLength):
            x = bytearray(self.rawdata[i*4:self.intNumTo])
            x = struct.unpack("f", x)
            self.data.append(x)
            self.intNumTo += 4 # 4bytes 32bit
        return self.data

    #Funktion um die Daten für einen Trainingsschritt zu extrahieren und umzuformen
    def get_data(self, PSinput):
        self.rawdata = PSinput[1:]
        self.intNum = 0
        self.intNumTo = 4
        self.data = []
        self.arrayLength = len(self.rawdata)
        self.arrayLength = int(self.arrayLength/4) #divide by 4 32bit float
        for i in range(self.arrayLength):
            x = bytearray(self.rawdata[i*4:self.intNumTo])
            x = struct.unpack("f", x)
            self.data.append(x)
            self.intNumTo += 4 # 4bytes 32bit
        self.data = np.asarray(self.data, dtype=np.float32)
        return self.data

    #Funktion um die Daten für einen Trainingsschritt zu extrahieren und umzuformen
    def get_memory(self, PSinput):
        self.rawdata = PSinput[8:]
        #Werte zum separieren (2=actual state, 3=action, 4=reward, 5=new state, 6=total reward, 7=sim episode)
        split = (PSinput[2], PSinput[3], PSinput[4], PSinput[5], PSinput[6], PSinput[7]) #Variablen für die Länge jeder Liste
        #initialisiere alle Listen um Daten aus übermitteltem Paket zu auszulesen
        self.result = []
        self.state = []
        self.action = []
        self.reward = []
        self.next_state = []
        rNum = 0

        for k in range(6):
            if k == 0:
                intNumTo = 4
            arrayLength = split[k] #Länge des Arrays der Zustands
            for i in range(arrayLength):
                x = bytearray(self.rawdata[rNum*4:intNumTo])
                x = struct.unpack("f", x)
                if k == 0: #lese den aktuellen Zustand aus dem Array
                    self.state.append(x)
                elif k == 1: #lese die Aktion vom Array
                    x = int(x[0])
                    self.action = x
                elif k == 2: #lese die Belohnung aus dem Array
                    x = round(x[0],2)
                    self.reward = x
                elif k == 3: #lese Folgezustand aus Array
                    self.next_state.append(x)
                elif k == 4: #lese Gesamtbelohnung aus Array
                    self.total_reward = x[0]
                elif k == 5:
                    self.sim_episode = x[0]
                intNumTo += 4 # 4bytes 32bit
            rNum += 1 #Schleifenvariable

        #formatiere die Listen der extrahierten Daten in NumPyArrays -> für Keras notwendig
        self.state = np.asarray(self.state, dtype=np.float32)
        self.next_state = np.asarray(self.next_state, dtype=np.float32)
        self.result = (self.state, self.action, self.reward, self.next_state, self.total_reward, self.sim_episode)

    #Rückgabewert sind Listen der Zustände, Aktionen, Belohnungen, Folgezustände, Gesamtbelohnung und SimEpisode
    return self.result
```

Literaturverzeichnis

Alpaydin, Ethem (2019): Maschinelles Lernen. 2., erweiterte Auflage. Berlin, Boston: De Gruyter (De Gruyter Studium).

Arthur Juliani (2016): Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond. Online verfügbar unter <https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df>, zuletzt geprüft am 26.07.2020.

Bauernhansl, Thomas; Hompel, Michael ten; Vogel-Heuser, Birgit (Hg.) (2014): Industrie 4.0 in Produktion, Automatisierung und Logistik. Wiesbaden: Springer Fachmedien Wiesbaden.

Ciubotaru, Bogdan; Muntean, Gabriel-Miro (Hg.) (2013): Advanced Network Programming – Principles and Techniques. London: Springer London (Computer Communications and Networks).

deepai.org (2019): ReLu. Online verfügbar unter <https://deepai.org/machine-learning-glossary-and-terms/relu>, zuletzt aktualisiert am 17.05.2019, zuletzt geprüft am 01.08.2020.

Eley, Michael (Hg.) (2012): Simulation in der Logistik. Berlin, Heidelberg: Springer Berlin Heidelberg (Springer-Lehrbuch).

Frochte, Jörg (2019): Maschinelles Lernen. Grundlagen und Algorithmen in Python. 2., aktualisierte Auflage. München: Hanser.

Görz, Günther; Schneeberger, Josef; Schmid, Ute (2013): Handbuch der Künstlichen Intelligenz. München: De Gruyter.

Günthner, Willibald; Hompel, Michael ten (2010): Internet der Dinge in der Intralogistik. Berlin, Heidelberg: Springer Berlin Heidelberg.

Hachtel, Günther; Holzbaur, Ulrich (2010): Management für Ingenieure. Technisches Management für Ingenieure in Produktion und Logistik. Wiesbaden: Vieweg + Teubner.

Hernandez-Leal, Pablo; Kartal, Bilal; Taylor, Matthew E. (2018): A Survey and Critique of Multiagent Deep Reinforcement Learning. Online verfügbar unter <https://arxiv.org/pdf/1810.05587.pdf>.

IBM Manufacturing (2018): Künstliche Intelligenz (KI) in der Fertigung. Ein Überblick und vier Anwendungen, die heute die Fertigung transformieren (IDW12363DEDE-01). Online verfügbar unter <https://www.ibm.com/downloads/cas/VWD75RJA>.

Inga Döbel et Al (2018): Maschinelles Lernen. Eine Analyse zu Kompetenzen, Forschung und Anwendung. Hg. v. Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V.

Jodlbauer, Herbert (2008): Produktionsoptimierung. Wertschaffende sowie kundenorientierte Planung und Steuerung. Zweite, erweiterte Auflage. Vienna: Springer-Verlag/Wien (Springers Kurzlehrbücher der Wirtschaftswissenschaften).

Kruse, Rudolf; Borgelt, Christian; Klawonn, Frank; Moewes, Christian; Ruß, Georg; Steinbrecher, Matthias (2011): Computational Intelligence. Wiesbaden: Vieweg+Teubner Verlag.

Lapan, Maxim (2020): Deep reinforcement learning hands-on. Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more. Second edition. Birmingham, UK: Packt Publishing Ltd.

Lillicrap, Timothy P.; Hunt, Jonathan J.; Pritzel, Alexander; Heess, Nicolas; Erez, Tom; Tassa, Yuval et al. (2015): Continuous control with deep reinforcement learning. Online verfügbar unter <https://arxiv.org/pdf/1509.02971>.

Lödding, Hermann (2016): Verfahren der Fertigungssteuerung. Berlin, Heidelberg: Springer Berlin Heidelberg.

Murphy, Kevin P. (2012): Machine learning. A probabilistic perspective (Adaptive computation and machine learning).

Murrenhoff, Anike; Roidl, Moritz; Hompel, Michael ten (2019): Steuerungskonzept für virtualisierte und lernfähige Materialflusssysteme.

Nandy, Abhishek; Biswas, Manisha (2018): Reinforcement learning. With Open AI, TensorFlow and Keras using Python. [Berkeley, Calif.], New York: Apress; Springer Science+Business Media.

Nyhuis, Peter; Wiendahl, Hans-Peter (2012): Logistische Kennlinien. Berlin, Heidelberg: Springer Berlin Heidelberg.

Patrick Dammann: Einführung in das Reinforcement Learning. Uni Heidelberg.

Petzka, Henning; Rüping, Stefan; Voss, Angelika; Döbel, Inga; Molina Vogelsang, Manuel; Neustroev, Dmitry et al. (2018): Vorhabensbezeichnung: Maschinelles Lernen - Kompetenzen, Anwendungen und Forschungsbedarf (Kurzbezeichnung: MaLe).

Schlussbericht des Vorhabens: "Maschinelles Lernen - Kompetenzen, Anwendungen und Forschungsbedarf" : Projektlaufzeit: 01.04.2017 bis 31.03.2018. Fraunhofer-Gesellschaft. München.

Rashid, Tariq (2017): Neuronale Netze selbst programmieren. Ein verständlicher Einstieg mit Python. 1. Auflage. Heidelberg: O'Reilly (Animals). Online verfügbar unter <https://ebookcentral.proquest.com/lib/gbv/detail.action?docID=5102090>.

Schenk, Michael; Wirth, Siegfried; Müller, Egon (2014): Fabrikplanung und Fabrikbetrieb. Berlin, Heidelberg: Springer Berlin Heidelberg.

Schuh, Günther; Stich, Volker (2012): Produktionsplanung und -steuerung 1. Berlin, Heidelberg: Springer Berlin Heidelberg.

Schwaiger, Roland; Steinwendner, Joachim (2019): Neuronale Netze programmieren mit Python. 1. Auflage (Rheinwerk Computing).

Sutton, Richard S.; Barto, Andrew (2018): Reinforcement learning. An introduction. Second edition. Cambridge, MA, London: The MIT Press (Adaptive computation and machine learning).

Tom Schaul; John Quan; Ioannis Antonoglou; David Silver (2015): Prioritized Experience Replay.

van Hasselt, Hado; Guez, Arthur; Silver, David (2015): Deep Reinforcement Learning with Double Q-learning. Online verfügbar unter <https://arxiv.org/pdf/1509.06461>.

Westkämper, Engelbert; Zahn, Erich (2009): Wandlungsfähige Produktionsunternehmen. Berlin, Heidelberg: Springer Berlin Heidelberg.

Wiendahl, Hans-Peter (2019): Betriebsorganisation für Ingenieure. 9., vollständig überarbeitete Auflage.

Wikipedia (2020a): Keras. Online verfügbar unter <https://de.wikipedia.org/w/index.php?title=Keras&oldid=201448819>, zuletzt aktualisiert am 30.06.2020, zuletzt geprüft am 01.08.2020.

Wikipedia (Hg.) (2020b): Socket (Software). Online verfügbar unter [https://de.wikipedia.org/w/index.php?title=Socket_\(Software\)&oldid=198672385](https://de.wikipedia.org/w/index.php?title=Socket_(Software)&oldid=198672385), zuletzt aktualisiert am 09.04.2020, zuletzt geprüft am 01.08.2020.

Wilke, Michael (2006): Wandelbare automatisierte Materialflusssysteme für dynamische Produktionsstrukturen. Zugl.: München, Techn. Univ., Diss., 2006. München: Utz (Fördertechnik - Materialfluss - Logistik).

Eidesstattliche Erklärung

Ich versichere, dass ich diese Masterarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Ulm, den

Unterschrift