



UNIVERSITÄT AUGSBURG

Fakultät für Angewandte Informatik

## Projektmodul

### Datenbasierter Entwurf eines Kraftreglers für den einachsigen Seilroboter

---

vorgelegt von: Julius Brandl und Valentin Höpfner

eingereicht am: 25. 07. 2022

Studiengang: Ingenieurinformatik

Anfertigung am Lehrstuhl: Regelungstechnik

Fakultät für Angewandte Informatik

Wissenschaftlicher Betreuer: Marcus Hamann

# Kurzfassung

Der Seilroboter *MoCaRo* verfügt über Aktorik, mit der Möglichkeit zur Kraftvorgabe, sowie Sensorik mit der Möglichkeit zur Kraftmessung. Eine Kraftregelung des gesamten Systems gestaltet sich dennoch als schwierig, da sich das System aus Seilwinde, Umlenkrollen und Seil stark nichtlinear verhält. Traditionelle Methoden zur Identifikation und Regelung dieses Systems wurden in der Vergangenheit erprobt und lieferten keine zufriedenstellenden Ergebnisse. Daher werden in diesem Projektmodul zwei datenbasierte Ansätze zur Modellierung und Regelung dieses Systems implementiert und analysiert. Zunächst wurde ein Aufbau entwickelt, in dem das System aus Winde, Rollen und Seil isoliert betrachtet werden kann. An diesem Aufbau kann ein Reinforcement-Learning-Ansatz direkt eine Regelungsstrategie erlernen. Beim zweiten Ansatz wird ein, auf neuronalen Netzen basierendes, Modell aus der ANARX-Klasse entworfen. Aus dieser Modellstruktur kann daraufhin ebenfalls ein Regler abgeleitet werden.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Nichtlinearitäten . . . . .	2
1.2 Related Work . . . . .	3
1.3 Aufbau der Arbeit . . . . .	4
<b>2 Aufbau</b>	<b>5</b>
2.1 Seilroboter . . . . .	5
2.2 Teststand . . . . .	5
2.2.1 Mechanischer Aufbau des Teststands . . . . .	6
2.2.2 Elektrischer Aufbau des Teststands . . . . .	8
<b>3 Methoden</b>	<b>10</b>
3.1 Reinforcement Learning . . . . .	10
3.1.1 Reglerauslegung mittels Reinforcement Learning . . . . .	12
3.1.2 Konfiguration des Environments . . . . .	14
3.2 Systemidentifikation mit neuronalen Netzen . . . . .	17
3.2.1 NARX-Modelle . . . . .	18
3.2.2 NARX als neuronales Netz . . . . .	18
3.2.3 ANARX als neuronales Netz . . . . .	19
3.2.4 Umwandlung in Zustandsraumform . . . . .	20
3.2.5 Regelungsansatz auf Basis von ANARX . . . . .	22
3.2.6 Training . . . . .	24
<b>4 Software</b>	<b>28</b>
4.1 Softwarekomponenten für RL . . . . .	28
4.1.1 Architektur der RL Environments . . . . .	29
4.1.2 Simulation . . . . .	32
4.2 Softwarebibliothek für (A)NARX . . . . .	33
4.2.1 Funktionalitäten . . . . .	33
4.2.2 Architektur . . . . .	34

4.2.3	Verwendung . . . . .	35
4.3	Weitere Software . . . . .	36
4.3.1	Tensorboard . . . . .	37
4.3.2	Weights and Biases . . . . .	37
<b>5</b>	<b>Ergebnisse</b>	<b>39</b>
5.1	DDPG . . . . .	39
5.1.1	Testaufbau . . . . .	39
5.1.2	Wahl der Reglerarchitektur . . . . .	41
5.1.3	Wahl der Observation . . . . .	41
5.1.4	Wahl des Rewards . . . . .	44
5.1.5	Hyperparameter . . . . .	46
5.1.6	Überführung auf den Hardwareaufbau . . . . .	46
5.2	(A)NARX . . . . .	49
5.2.1	Aufbereitung der Daten . . . . .	49
5.2.2	Wahl der Hyperparameter . . . . .	50
5.2.3	Training . . . . .	51
5.2.4	Einsatz am realen System . . . . .	53
5.3	Vergleich . . . . .	56
5.3.1	Naiver Ansatz . . . . .	57
5.3.2	Sprünge . . . . .	57
5.3.3	Sinus . . . . .	58
5.3.4	Fazit . . . . .	58
<b>6</b>	<b>Ausblick</b>	<b>61</b>
<b>Literatur</b>		<b>63</b>

# 1 Einleitung

Seilroboter finden eine breite Anwendung. So wird beispielsweise die SkyCam<sup>1</sup> in Stadien eingesetzt, um Veranstaltungen ohne Drohnen oder Hubschrauber zu filmen. Dabei überzeugt diese mit einer hohen Nutzlast bei einer gleichzeitig großen möglichen Dynamik. Ein Seilroboter wird ebenfalls verwendet, um den Empfänger des größten Radioteleskops der Welt (FAST)<sup>2</sup> über dem Parabolspiegel mit 300 m Durchmesser auszurichten. Auch hier spielt die große mögliche Nutzlast eine wichtige Rolle. Für eine schnelle Dynamik und genaue Positionierung des Endeffektors des Seilroboters ist es essentiell die am Endeffektor wirkenden Kräfte zu regeln. So muss dafür gesorgt werden, dass alle Seile jederzeit auf Spannung gehalten werden. Außerdem wird durch die Kraftregelung auch die Interaktion mit dem Menschen ermöglicht. So kann beispielsweise bei aktiver Kraftregelung der Endeffektor von einem Menschen gefahrlos durch den Arbeitsraum gezogen und positioniert werden [17]. Damit ist ein einfaches, aus der Industrierobotik bekanntes, *teach-in* möglich.

Als Zwischenschritt, bevor eine Kraftregelung für den ganzen Seilroboter entsteht, soll ein Regler für ein einziges Motormodul entworfen werden. Bereits die Kraftregelung eines einzigen Moduls gestaltet sich dabei als schwierig. So konnte kein PID-Regler entworfen werden, welcher stabil und ausreichend dynamisch ist. Deshalb sollen in dieser Arbeit zwei verschiedene datenbasierte Verfahren zur Auslegung eines Kraftreglers getestet werden. Dazu kommt einerseits ein Reinforcement Learning (RL) Ansatz zum Einsatz, welcher, ohne vorherige Modellbildung des System, direkt einen Regler entwerfen soll. Andererseits wird ein zweites Verfahren untersucht, welches das System zuerst mittels eines neuronalen Netz modellieren soll. Anschließend kann durch die speziell gewählte Netzstruktur daraus ein Regler entworfen werden.

---

<sup>1</sup><http://www.skycam.tv/>

<sup>2</sup><https://fast.bao.ac.cn/>

## 1.1 Nichtlinearitäten

Das betrachtete System hat im Wesentlichen zwei Nichtlinearitäten, die sich mit klassischen Methoden der physikalischen Systemidentifikation nur schwer modellieren lassen. Diese machen einen datenbasierten Ansatz attraktiv, was auch der Grundgedanke hinter diesem Projektmodul war. Beide Nichtlinearitäten lassen sich anhand einer Treppenfunktion, wie sie in Abbildung 1.1 dargestellt ist, gut erkennen:

**Kriechverhalten** Oben in Abbildung 1.1 ist das Kriechverhalten des DMS-Signals nach einem Sprung dargestellt. Die Richtung des Kriechens ist in nahezu allen Fällen entgegengesetzt zur Richtung des Sprunges. Es konnte bei Messungen auch nicht festgestellt werden, dass das Signal gegen einen Endwert konvergiert. Auch mehr als 60 Sekunden nach einem Sprung lässt sich noch ein Kriechen feststellen. Eine Ursache für das Kriechen könnte in der Längung des Seils liegen. Dafür würde sprechen, dass sich das Kriechen über das Positionssignal des Motors erfassen lässt (mehr dazu in Abschnitt 5.2.3).

**Zusammenhang zwischen Motorkraft und gemessener Kraft:** An der Treppenfunktion unten in Abbildung 1.1 kann man deutlich erkennen, dass der Zusammenhang zwischen der Kraftvorgabe am Motor und der Messung des Dehnmessstreifen(DMS) nichtlinear ist. Physikalisch könnte dies beispielsweise an nichtlinearen Reibungsverlusten in den Umlenkrollen liegen. Ein weiterer Erklärungsansatz wäre, dass der Kraftregler im Motor keinen Kraftsensor besitzt, sondern die Kraft nur anhand des Stroms schätzen kann.

Neben den Nichtlinearitäten lässt sich in Abbildung 1.1 gut das schnelle Einschwingen des DMS-Signals erkennen. Diese Dynamik ist so schnell, dass sie mit der alten Messeinheit überhaupt nicht erfasst werden konnte (mehr dazu in Abschnitt 2.2.2). Nicht nur für klassische Methoden stellt diese Kombination aus sehr schneller Schwingung und sehr langsamem Kriechen eine Herausforderungen dar: Um das System ausreichend mit datenbasierten Methoden zu beschreiben, ist sowohl eine hohe Abtastfrequenz (für das Schwingen) als auch eine lange Messreihe (für das Kriechen) nötig. Dadurch werden die Datensätze groß und der Einsatz von Machine-Learning-Methoden entsprechend rechenintensiv.

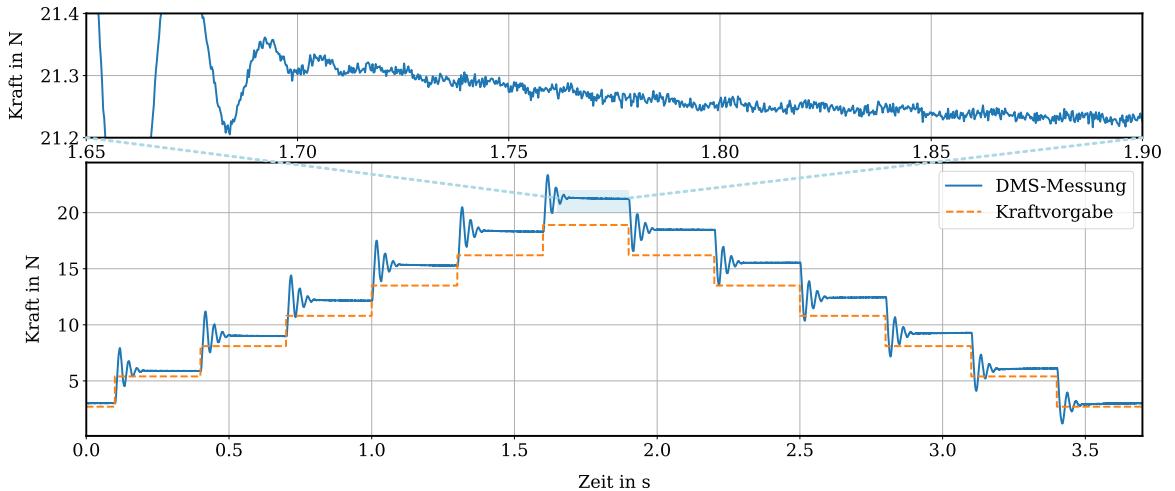


Abbildung 1.1: Die Nichtlinearitäten des Systems: Kriechverhalten (oben) und Zusammenhang zwischen Motorkraft und gemessener Kraft (unten)

## 1.2 Related Work

Kraftregelung ist seit langer Zeit ein wichtiger Teil in der Robotik. Speziell in Seilrobotern ist ein Kraftregler wichtig, damit alle am Endeffektor angebrachten Seile auf Spannung gehalten werden können. Dazu müssen Algorithmen entworfen werden, welche beschreiben, welche Sollkraftgrößen an jeder einzelnen Winde anliegen sollen [6][1][26]. Dazu muss in jeder Winde, und damit auch für jeden Motor, ein eigenständiger Kraftregler vorhanden sein. In [19] wird dafür das System eines Windenmoduls vollständig identifiziert. Dieses besteht neben dem Motor aus Umlenkrollen. Ebenfalls wird bei der Identifikation sowohl die Reibung der Umlenkrollen [18] als auch die Längung des Seils beachtet. Anhand dieses Modells wird als Kraftregler eine Vorsteuerung verbunden mit einem I-Regler vorgeschlagen. Im Unterschied zu dem in dieser Arbeit verwendeten Aufbau wird der Motor mittels Positions vorgabe und nicht per Kraftvorgabe betrieben. In [23] und [20] wird ebenfalls ein vollständiges Modell erarbeitet, jedoch wird ein PID-Regler, verbunden mit einer Vorsteuerung, zur Kraftregelung verwendet. In den eben genannten Arbeiten, sowie in [17] und [16] ist neben dem Kraftregler ein parallel arbeitender Positionsregler verbaut. Somit kann beispielsweise eine Kraft in einer Richtung aufgebracht werden, während eine Trajektorie aus vordefinierten Punkten abgefahren wird. In dieser Arbeit ist der parallele Positionsregler aber nicht umsetzbar, da das Seil fest mit dem Aufbau verspannt wird. Einen anderen Ansatz verfolgt [4], dabei ist das Seil den Aufbau durch Entfernung des Kraftmessers zu vereinfachen und sich so zwei Umlenkrollen einzusparen. Die Kraftmessung erfolgt dabei mittels des Motorstroms. Auch datenbasierte Ansätze werden zur Modellierung eines Seilroboters verwendet. So

wird in [7] die inverse Kinematik des Seilroboters, unter der Beachtung der Seilelastizität und der Seilmasse, mithilfe eines neuronalen Netzwerks berechnet. In [24] wird hingegen mit einer ähnlichen Methode die Vorwärtskinematik berechnet.

## 1.3 Aufbau der Arbeit

Dieser Bericht beginnt mit einer Beschreibung des Messstandes, der im Rahmen dieses Projektes entworfen wurde (Abschnitt 2). Neben dem mechanischen Aufbau wird dort auch die verwendete Elektronik ausführlicher erklärt. Im anschließenden Kapitel 3 werden die verwendeten modellbasierten und modellfreien Identifikationsmethoden, die im Rahmen dieses Projektes zum Einsatz kamen, aus einer eher theoretischen Sichtweise erklärt. Die Software, mit der diese Methoden implementiert wurden, spielte für die Projektarbeit eine große Rolle. Aus einer softwaretechnischen Sichtweise wird diese in Kapitel 4 erklärt. Das Herzstück dieses Projekts ist aber sicherlich die Anwendung der Modellierungsmethoden und Softwarelösungen auf Daten, die mit dem Messstand erzeugt wurden. Was sich dabei für Herausforderungen gestellt haben und welche Erkenntnisse dabei gewonnen werden konnten wird in Kapitel 5 ausgeführt. Am Ende dieses Kapitels findet sich auch nochmal ein Vergleich der verschiedenen Methoden anhand einer Regelung. Dieser Bericht endet schließlich mit Kapitel 6, einem Ausblick.

## 2 Aufbau

Ein Teil dieses Projektmoduls war Entwurf und Realisierung eines Teststandes, der auch als Grundlage für alle Experimente und Daten der späteren Kapitel diente. Dieser Aufbau wird hier beschrieben.

### 2.1 Seilroboter

Am Lehrstuhl für Regelungstechnik der Universität Augsburg gibt es einen Seilroboter, der den Namen *MoCaRo* (Modular Cable Robot) trägt. Dessen Aktorik besteht aus bis zu 6 Windenmodulen, die über ein Seil, das über eine Umlenkrolle läuft, mit einem Endeffektor verbunden sind. Dieser Endeffektor kann durch Variation der Seillängen frei im Arbeitsraum positioniert werden. Der Aufbau des *MoCaRo* ist in Abbildung 2.1 dargestellt. In jedem Windenmodul ist über eine flaschenzugartige Konstruktion ein Dehnmessstreifen (DMS) zur Kraftmessung verbaut. Damit kann die auf das jeweilige Seil wirkende Kraft gemessen werden. Ziel dieses Projekts ist es, ein Modell zu erstellen, was den Zusammenhang zwischen der Kraft, die auf das Seil wirkt und der Kraft die am DMS wirkt, beschreibt. Dies soll perspektivisch als Grundlage für eine Kraftregelung des Endeffektors dienen. Um das System isoliert betrachten zu können wurde ein Teststand entworfen der aus nur einem Windenmodul, einer Umlenkrolle sowie einem fixierten Seil besteht. Dieser Teststand wird im folgenden Abschnitt 2.2 beschrieben.

### 2.2 Teststand

Bevor die verschiedenen Regleransätze für den gesamten Seilroboter entworfen werden, sollen diese an nur einem Windenmodul konzipiert und getestet werden. Dafür wurde ein Windenmodul aus dem Seilroboter entfernt und in einen eigenen Testaufbau eingefügt. Der Teststand soll transportabel gestaltet sein, damit dieser schnell an einem anderen Ort aufgebaut werden kann. Somit wäre es beispielsweise möglich mit dem Aufbau einen Praktikumsversuch zu entwickeln. Dafür muss nicht nur das Windenmodul am Teststand

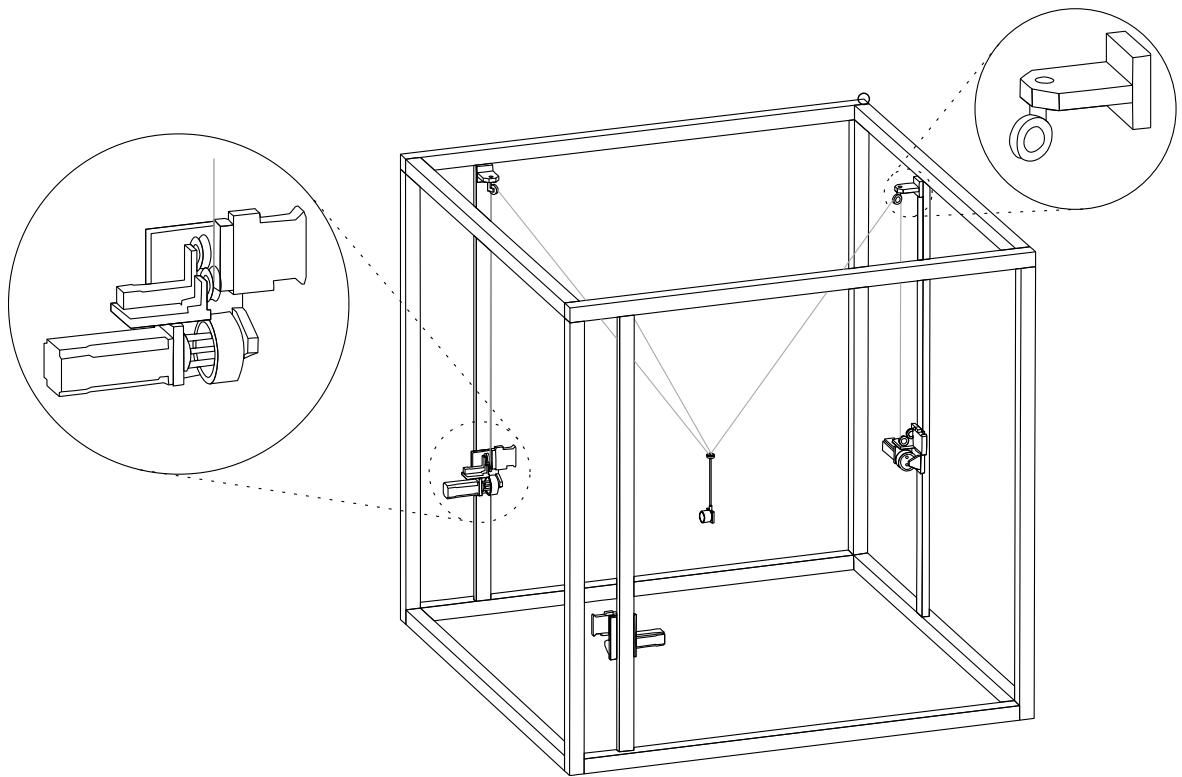


Abbildung 2.1: Aufbau des *MoCaRo* mit Windenmodul links und Umlenkrolle rechts,  
Abbildung mit Anpassungen übernommen aus[38]

befestigt werden, sondern auch die dazugehörige Auswerte- und Ansteuerungselektronik. Der mechanische als auch der elektrische Aufbau wird anschließend beschrieben.

### 2.2.1 Mechanischer Aufbau des Teststands

Der mechanische Aufbau des Teststands wird in Abbildung 2.2 präsentiert. Dabei kann dieser in drei Teile aufgeteilt werden: (1) ein Grundgerüst, das aus drei 80x80mm Aluminiumprofilen zusammengesetzt wird. Dabei bilden zwei Profile durch eine T-Anordnung einen stabilen Fuß. Darauf befindet sich, senkrecht stehend und mit Winkeln befestigt, ein drittes Profil an dem die restlichen Anbauten angebracht sind. Daran werden rückseitig (2) Netzteile und die Ansteuerungselektronik an Hutschienen befestigt. Vorderseitig an dem Profil wird hingegen das (3) Windenmodul und eine Umlenkrolle angebracht.

Das Windenmodul ist aus dem Seilroboter entnommen. Das Modul besteht aus einem Motor<sup>1</sup> der Firma Beckhoff mit einer Seiltrommel ( $d=10\text{cm}$ ) und einer Kraftmesseinheit.

---

<sup>1</sup><https://www.beckhoff.com/de-de/produkte/motion/rotatorische-servomotoren/am8100-servomotoren-fuer-kompakte-antriebstechnik/am8131-wfyz.html>

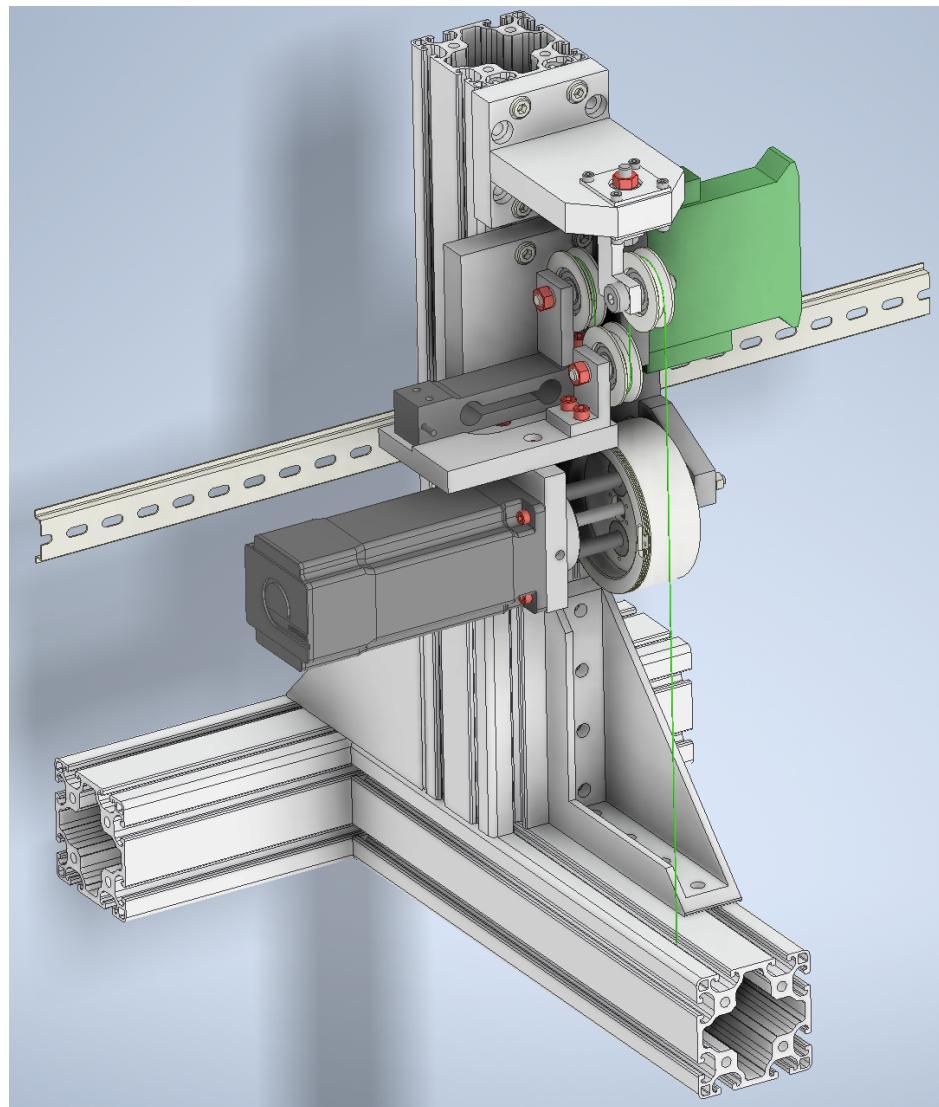


Abbildung 2.2: Aufbau des Teststands

Der Motor kann ca. eine Kraft von bis zu 1,3Nm aufbringen. Zur Kraftmessung wird ein Dehnungsmessstreifen (DMS) von Althen<sup>2</sup> verwendet, welcher einen Messbereich von bis zu 10kg besitzt. Zur richtigen Funktion muss der DMS auf Druck beansprucht werden. Um dies zu gewährleisten sind auf dem Windenmodul mehrere Umlenkrollen angeordnet. Eine weitere drehbare Umlenkrolle, welche ebenfalls aus dem Seilroboter entnommen wurde, leitet das Seil wieder zurück nach unten, wo es im Aluminiumprofil verankert wird.

<sup>2</sup><https://www.althensensors.com/de/sensoren/wiegetechnik-und-kraftaufnehmer/biegebalgen-scherstab-wiegezellen/15132/aobur-serie-biegebalgen-kraftaufnehmer/>

### 2.2.2 Elektrischer Aufbau des Teststands

Um den DMS auszulesen und den Motor anzusteuern ist Auslese- und Ansteuerungselektronik nötig. Zur Ansteuerung des Motors wird eine Beckhoff Motorklemme<sup>3</sup> verwendet, welche sich mit einer maximalen Frequenz von 12kHz ansteuern lässt. Der Motor wird durch eine Drehmomentvorgabe angesteuert. Alternativ ist der Motor auch mit einer Geschwindigkeits- oder Positionsvorgabe betreibbar.

Zur Auswertung des DMS wurde ursprünglich ein Analog-Digital-Wandler von Laumas genutzt<sup>4</sup>. Dieser besitzt jedoch nicht die nötige Dynamik um den realen Kraftverlauf am DMS zu analysieren. Deshalb wurde ein Wechsel auf eine Auswerteklemme von Beckhoff<sup>5</sup> vollzogen, welche die Auswertung des DMS mit bis zu 20kHz zulässt. In Abbildung 2.3 ist der Unterschied bei der Aufzeichnung der Dynamik zu erkennen. Dazu werden die Sprungantworten der beiden Auswerteeinheiten bei einem Momentsprung des Motors von 0,3 Nm auf 0,75 Nm verglichen. Die neue Auswerteeinheit zeichnet die Schwingung des Systems auf, während die alte Auswerteeinheit keinerlei Schwingung zeigt. Ebenfalls ist das alte System beim Erreichen des Endwerts wesentlich langsamer. Deshalb musste die alte Einheit ausgetauscht werden und die neue Auswerteeinheit wird in der restlichen Arbeit verwendet. Zur Kalibration der neuen Auswerteeinheit wurden verschiedene bekannte Gewichte an das Seil angehängt und die Auswerteeinheit ausgelesen. Daraufhin wurde mittels Regression der Zusammenhang zwischen Messwert und Gewicht bestimmt. Dabei wurde ein linearer Zusammenhang angenommen, mit welchem der Messwert verrechnet wird. Somit lautet die Umrechnung von Inkrementen des DMS in N:

$$F_N = (F_{Ink} - 202000) \cdot 5.6691 \cdot 10^{-7} \cdot 9.81 \quad (2.1)$$

Da die Kalibration im System mit Seil und Umlenkrollen stattgefunden hat, ist diese nicht sehr genau. Eine genauere Kalibration müsste bei einem ausgebauten DMS erfolgen.

Zur Ansteuerung der Klemmen wird eine Beckhoff-SPS<sup>6</sup> genutzt. Diese lässt sich mit MATLAB/Simulink programmieren. Dabei können die Programme mit einer Frequenz von bis zu 20kHz ausgeführt werden. Obwohl kürzere Zykluszeiten möglich wären,

---

<sup>3</sup><https://www.beckhoff.com/de-de/produkte/i-o/ethercat-klemmen/el7xxx-kompakte-antriebstechnik/el7211-0010.html>

<sup>4</sup><https://www.laumas.com/de/produkt/tlb-analoger-und-digitaler-waegetransmitter-rs485/>

<sup>5</sup><https://www.beckhoff.com/de-de/produkte/i-o/ethercat-klemmen/elmxxxx-messtechnik/elm3502-0000.html>

<sup>6</sup><https://www.beckhoff.com/de-de/produkte/ipc/pcs/c60xx-ultra-kompakt-industrie-pcs/c6030.html>

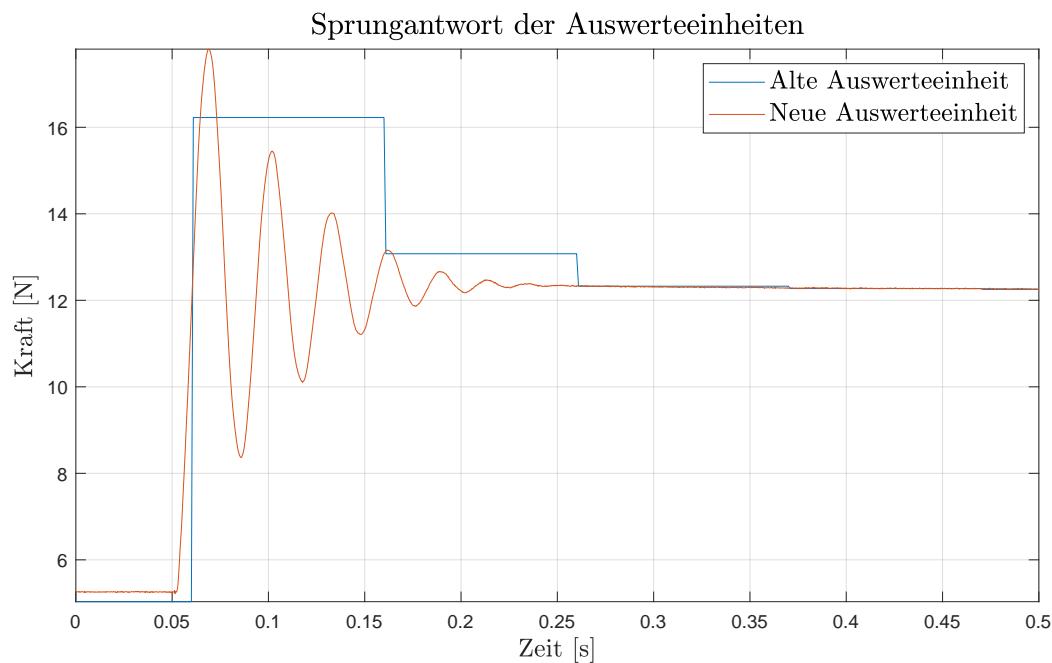


Abbildung 2.3: Vergleich der Auswerteeinheiten

wurde sich zur Synchronisation der Abtastzeiten zwischen Motor und DMS entschieden, weshalb das System mit einer Frequenz von 4kHz betrieben wird. Trotzdem ist es mittels Supersampling möglich, dass die DMS-Messklemme mit 20kHz den DMS abtastet. Hierbei wird mit den öfter abgetasteten Messpunkten der Mittelwert über die letzten fünf Messpunkte gebildet. Somit lässt sich das Messsignal mit nur einem sehr geringen Dynamikverlust im Vergleich zum Gesamtsystem vorfiltern.

# 3 Methoden

In diesem Abschnitt sollen die datenbasierten Methoden aufgezeigt werden, welche zur Kraftregelung des vorliegenden System genutzt werden. Dabei wird mit Reinforcement Learning zuerst ein modellfreier und anschließend mit ANARX ein modellbasierter Ansatz verwendet.

## 3.1 Reinforcement Learning

Es wurde bereits gezeigt, dass sich mit Reinforcement Learning (RL) verschiedene komplexe Probleme lösen lassen. So ist es beispielsweise in [25] möglich verschiedene Atari-Spiele nur mittels der Pixel als einzige Eingabe spielen zu lassen. In einer anderen Arbeit mit stärkerem Bezug zu physikalischen Größen und Regelungstechnik wird in [14] in einer Simulationsumgebung ein Regler für die Höhenkontrolle einer Drohne entworfen. In [36] wird ein RL-Regler entworfen, welcher zur Navigation eines unbenannten Wasserfahrzeugs verantwortlich ist. Inzwischen gibt es sogar bereits einen Benchmark, der RL-Regler in verschiedenen Szenarien vergleicht [8]. Des Weiteren wird in [3][37] gezeigt, dass sich RL auch bei nicht linearen System anwenden lässt. In [22] wird gezeigt, dass sich diese Ergebnisse auch auf reale physikalische Systeme übertragen lassen. Hier werden die Motoren eines Roboters angesteuert und die Endeffektorposition geregelt. Dabei wird direkt online auf dem System gelernt, somit muss kein Modell der Strecke erstellt werden. In [10] werden ganze Trajektorien, welche zum Lösen von Aufgaben von einem Roboter abgefahren werden, erlernt. Dies geschieht dabei sowohl in Simulation als auch mit echten Robotern. Aufgrund der beschriebenen Erkenntnisse soll RL auch für die hier benötigte Kraftregelung evaluiert werden.

Mittels RL soll eine Lösungsstrategie durch geschicktes Ausprobieren gefunden werden. Dabei interagiert ein Agent schrittweise mit einem Environment. Der Agent erhält in jedem Schritt eine Observation  $o_t$  vom Environment, welche einen Teil oder auch den gesamten Zustand  $s_t$  des Environment enthält. Mit dieser berechnet der Agent eine Action  $a_t$ , welche dann wieder auf dem Environment ausgeführt wird. Die Berechnungs-

vorschrift 3.1 des Agenten für die nächste Action nennt sich Policy  $\mu$ .

$$a_t = \mu_\theta(s_t) \quad (3.1)$$

Die Policy ist eine Funktion mit anpassbaren Parametern  $\theta$ . Diese Parameter sind z.B. die Gewichte eines neuronalen Netzes. Meist wird in der Literatur zur Beschreibung der Observation statt des besser passenden  $o$  das eigentlich unpassendere  $s$  verwendet. Deshalb folgt diese Arbeit der bekannten Schreibweise.

Außerdem gibt das Environment mit jeder neuen Observation auch einen Reward  $r_t$  zurück, welcher den aktuellen Zustand des Systems bewertet. Mittels Gleichung 3.2 lässt sich der gesamte erwartete Reward über einen Zustandsverlauf  $\tau$ , genannt Return  $R(\tau)$ , ausdrücken. Durch den Diskontierungsfaktor  $\gamma$  lässt sich steuern, wie stark der zukünftige Reward gewichtet werden soll. Je größer  $\gamma$  ist, desto weiter „denkt“ der Agent in die Zukunft.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \quad |\gamma \in (0, 1) \quad (3.2)$$

Ziel von RL ist es nun diejenige Policy  $\mu^*$  zu finden, welche den erwarteten Return maximiert. Daraus folgt:

$$\mu^* = \arg \max_{\mu} E_{\tau \sim \mu} [R(\tau)] \quad (3.3)$$

Ein Agent, der dieser Policy folgt, führt also immer die Aktion aus, welche den maximalen zukünftigen Return verspricht. Mittels  $\tau \sim \mu$  wird angegeben, dass der Zustandsverlauf anhand der aktuellen Policy gebildet wird.

Mathematisch kann das Zusammenspiel zwischen Actor und Environment als ein Markov Entscheidungsproblem (MDP) formuliert werden. Das MDP besteht dabei aus einem 5-Tupel  $(S, A, R, P, p_0)$ , mit:

- $S$ : Menge aller validier Zustände bzw. Beobachtungen.
- $A$ : Menge aller möglicher Aktionen.
- $R$ : Reward Funktion,  $r_t = R(s_t, a_t, s_{t+1})$ .
- $P$ : Übergangswahrscheinlichkeitsfunktion, Wahrscheinlichkeit  $P(s'|s, a)$  in Zustand  $s'$  zu gelangen, wenn in Zustand  $s$  die Aktion  $a$  gewählt wird.
- $p_0$ : Menge aller möglicher Startzustände.

Dabei muss das System die Markov-Eigenschaft, d.h. dass jeder Zustandsübergang nur vom aktuellen Zustand und der aktuellen Action abhängt, erfüllen. [28]

### 3.1.1 Reglerauslegung mittels Reinforcement Learning

Im Feld von RL gibt es viele verschiedene Algorithmen zum Erlernen der besten Policy  $\mu^*$ , welche zu jeder Observation die beste Action liefert. Eine Klasse dieser Algorithmen erlernt die Policy direkt, indem sie die Policy abhängig von den Parameter  $\theta$  ausdrücken und diese direkt per Gradientenabstieg trainieren. Deshalb werden diese Ansätze als *Policy Optimization* bezeichnet.

Andererseits ist es möglich die Policy indirekt zu erhalten, indem ein Approximator  $Q_\theta(s, a)$  für die optimale Action-Value-Funktion  $Q^*(s, a)$  erlernt wird. Wie in Gleichung 3.4 beschrieben, gibt diese Funktion an, welcher Return erwartet wird, wenn man, beginnend in einem Zustand  $s$ , eine beliebige action  $a$  ausführt und danach immer der optimalen Policy folgt. Der Zustandsverlauf wird wieder mittels  $\tau$  ausgedrückt.

$$Q^*(s, a) = \max_{\mu} E_{\tau \sim \mu}[R(\tau) | s_0 = s, a_0 = a] \quad (3.4)$$

Mithilfe der optimalen action-value Funktion kann ebenfalls die nächste beste action bestimmt werden:

$$a^* = \arg \max_a Q^*(s, a) \quad (3.5)$$

Hierbei wird diejenige action gewählt, welche den Return, wenn man sich nach der gewählten Action an die Policy hält, maximiert. Aufgrund des Namens der Funktion werden diese indirekten Ansätze *Q-Learning* genannt.

Ein großer Vorteil von *Q-Learning* ist, dass die Optimierung der Parameter fast immer off-policy stattfindet. D.h., dass alle gesammelten Datenpunkte, unabhängig von der damaligen Policy, für das weitere Training genutzt werden können. Bei *Policy-Optimization*-Algorithmen ist das Training meist on-policy, somit können nur Datenpunkte verwendet werden, welche mit der aktuellsten Policy generiert wurden. Off-Policy-Algorithmen sind deshalb wesentlich dateneffizienter, da gesammelte Daten oft mehrmals verwendet werden können.

Da es sich in dieser Arbeit um ein reales System handelt und sich dieses nicht parallelisieren lässt, ist diese Dateneffizienz sehr wichtig. Deshalb soll ein Off-Policy-Algorithmus eingesetzt werden. Außerdem ist das Ziel der Arbeit vorrangig zu zeigen, dass eine Reglerauslegung mittels RL möglich ist, weshalb vorerst auf komplexere RL-Algorithmen verzichtet werden soll. Hierdurch lässt sich gleichzeitig die Gefahr der starken Abhängigkeit von Hyperparametern bei RL-Algorithmen vermindern, wenn auch nicht lösen. Außerdem muss der Algorithmus einen kontinuierlichen Beobachtungs- und Aktionsraum unterstützen. Der einfachste Algorithmus, der diese Anforderungen

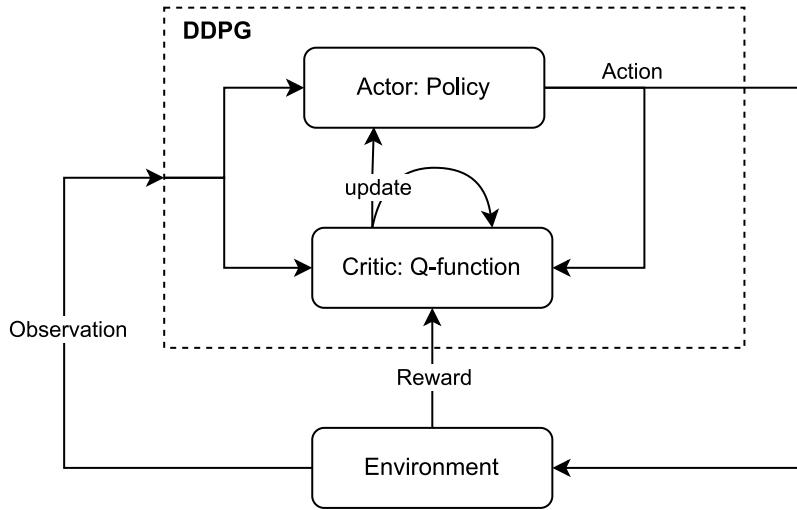


Abbildung 3.1: Schematischer Aufbau von DDPG [12]

erfüllt ist der *Deep Deterministic Policy Gradient (DDPG)* Algorithmus.

Ziel von DDPG ist es Deep Q-Learning mit einem kontinuierlichen Aktionsraum zu verknüpfen und somit einen RL-Algorithmus zu erstellen, welcher auch bei Regelungsproblemen, welche eine kontinuierliche Stellgröße erfordern, genutzt werden kann. Dies wurde in [21] zu DDPG bereits umgesetzt.

Der Algorithmus wird schematisch in Abbildung 3.1 dargestellt. Darin ist zu erkennen, dass sowohl *Q-Learning* als auch der *Policy-Optimization*-Ansatz genutzt werden. Zum Training müssen Transitionen, bestehend aus einem Tuple  $(s, a, r, s')$  generiert werden. Dazu wird die Reaktion des Environments auf verschiedene Actions getestet. Dabei erfolgen zur Erstellung die folgende Schritte: Das Environment gibt eine Observation  $s$  aus und DDPG generiert eine dazugehörige Action  $a$ . Diese wird wiederum auf das Environment angewandt. Hierdurch entsteht die nächste Observation  $s'$ . Gleichzeitig wird dabei ein Reward  $r$  erzielt. Anschließend wird die durchgeführte Transition abgespeichert.

Der DDPG-Teil des Schemas wird im folgenden Teil weiter beschrieben. Dabei wird der Algorithmus zum leichteren Verständnis vereinfacht, die Konzepte bleiben jedoch die gleichen. Eine genauere Erklärung findet sich beispielsweise in [27].

Im *Q-Learning*-Teil soll eine Approximation der optimale Action-Value-Funktion  $Q^*$  stattfinden. Dafür kann Gleichung 3.4 in die folgende, als Bellman-Gleichung bezeichnete, Gleichung umgeschrieben werden:

$$Q^*(s, a) = E[r(s, a) + \gamma \max_{a'} Q^*(s', a')] \quad (3.6)$$

Diese zerlegt den, in Gleichung 3.4 beschrieben, Return in zwei Summanden. Der erste Summand gibt den direkten Reward durch die nächste Action an. Der Zweite gibt den weiteren Return aus dem Ergebnis der Q-Funktion aus der darauf folgenden State-Reward-Kombination an. Das Ergebnis von Gleichung 3.4 und 3.6 bleibt jedoch dasselbe. Aus der Bellman-Gleichung lässt sich eine Loss-Funktion für das Training des neuronalen Netzes, welches den Approximator für  $Q^*$  darstellt, bilden. Dieses wird Netz wird Critic Network genannt. Der Loss wird folgendermaßen definiert:

$$L(\phi) = E \left[ \left( Q_\phi(s, a) - (r + \gamma \max_{a'} Q_\phi(s', a')) \right)^2 \right] \quad (3.7)$$

Dabei steht  $\phi$  für die veränderlichen Parameter des neuronalen Netzes. Während des Trainingsvorgangs werden die bereits beschrieben Transitionen in die Loss-Funktion eingesetzt. Diese Gleichung wird auch für das Training in Deep-Q Learning verwendet. Zur Bestimmung von  $a'$  muss außerdem die nächste Action bestimmt werden. Da im Fall eines kontinuierlichen Aktionsraums, wie er hier verwendet wird, die Auswertung der Gleichung 3.5 sehr aufwendig ist, wird auch hierfür ein neuronales Netz als Approximator der Funktion verwendet. Dieses Netz wird Actor Network genannt. Zur Optimierung des Actor Networks kann ein Gradientenabstiegsverfahren mit Bezug auf die Netzwerkparameter  $\theta$  angewendet werden. Damit kann folgende Gleichung gelöst werden:

$$\max_{\theta} E [Q_\phi(s, \mu_\theta(s))] \quad (3.8)$$

Fügt man nun den *Policy Optimization* Ansatz in den *Q-Learning* Ansatz ein erhält man für den Loss des Critic Networks:

$$L(\phi) = E \left[ \left( Q_\phi(s, a) - (r + \gamma \max_{a'} Q_\phi(s', \mu_\theta(s'))) \right)^2 \right] \quad (3.9)$$

Mit Hilfe eines geeigneten Optimierers kann der Loss minimiert und somit das Critic Network trainiert werden. Nachdem das Training abgeschlossen ist und die gelernte Policy in einer Produktivumgebung eingesetzt werden soll, kann der Critic Teil von DDPG verworfen werden. Zur Berechnung der nächsten Action ist ausschließlich der Actor Teil notwendig.

### 3.1.2 Konfiguration des Environments

Neben der Wahl des Algorithmus ist die Gestaltung des Reglers, die Wahl der beobachteten Variablen und die Berechnung des Rewards von großer Bedeutung. Auf

diese verschiedenen Designentscheidungen soll in diesem Abschnitt weiter eingegangen werden. In einem späteren Teil der Arbeit 5.1 werden diese Überlegungen mit experimentellen Daten veranschaulicht. Dazu wird neben dem physikalischen System eine Simulation erstellt, damit die Experimente schneller und mit weniger Hardwareaufwand durchgeführt werden können.

### **Wahl der Observations**

Die Wahl der Observatations hat einen enormen Einfluss auf die Performance des gelernten Reglers. Dabei muss entschieden werden, welche Signale dem RL-Algoritmus zur Verfügung gestellt werden und ob diese in irgendeiner Weise vorverarbeitet werden. Die unvorverarbeiteten Signale, welche in dieser Arbeit betrachtet werden, sind dabei die Führungsgröße  $w$ , die Stellgröße  $u$  und der Systemausgang  $y$ . Damit es dem System möglich ist einen zeitlichen Zusammenhang der Signale zu erlernen, können zusätzlich die gleichen Signale aus vergangenen Zeitschritten hinzugefügt werden.

Ein anderer Weg dem System diesen zeitlichen Zusammenhang mitzuteilen, ist es, explizit die Ableitungen der Signale zu berechnen. Dies hat den Vorteil, dass der Algorithmus schneller lernen wird. Gleichzeitig bringt dieses Vorgehen jedoch die Gefahr mit sich, dass dem System Informationen vorenthalten werden, welche möglicherweise zu besseren Ergebnissen führen würden. Neben dem zeitlichen Zusammenhang kann auch der Regelfehler 3.10 berechnet werden und dieser dem System direkt übergeben werden. Auch hier ist die Verwendung der letzten Zeitschritte oder der Ableitung des Fehlers denkbar.

$$e = w - y \quad (3.10)$$

Die Vor- und Nachteile bleiben bei dieser Vorgehensweise dabei die gleichen wie bei der Bildung der Ableitungen. So werden dem Netzwerk durch das Zusammenfassen zweier Rohsignale zu einem eventuell Informationen vorenthalten.

Die bereits beschriebenen Teilbeobachtungen können beliebig zur kompletten Observation zusammengefügt werden. Deshalb muss hier in einer Testphase bestimmt werden, welche Kombination von Teilbeobachtungen und welche dazugehörige Vorverarbeitung die besten Ergebnisse liefert.

### **Wahl des Rewards**

Durch einen geschickt gewählten Reward lässt sich die Lernzeit des Neuronales Netzes des Actor stark verkürzen. Der Reward sollte dabei möglichst in jedem Zeitschritt zurückgegeben werden. Da der Systemausgang der Führungsgröße möglichst fehler-

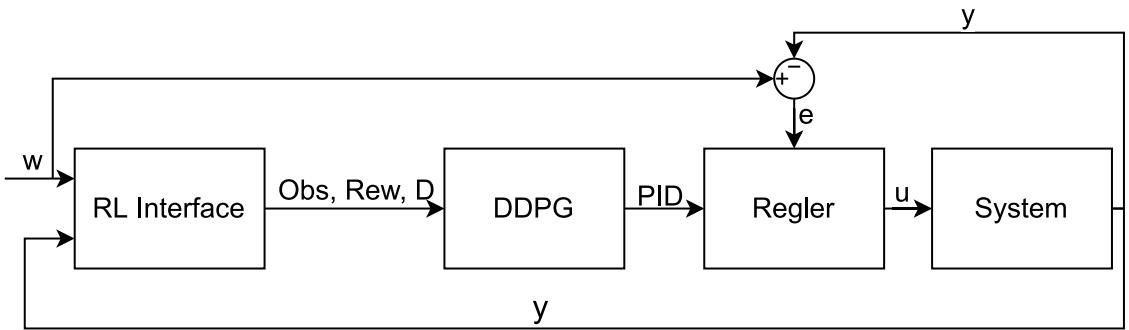


Abbildung 3.2: Erlernen der PID Parameter.

frei folgen soll, ist der absolute Fehler  $e_{abs} = \text{abs}(w - y)$  zwischen den Signalen ein offensichtlicher Anteil des Rewards. Um Stellenergie zu sparen und ein Schwingen des Systems zu unterbinden wird dem Reward außerdem eine Komponente hinzugefügt, welche die Änderung der Stellgröße  $u$  bestraft. Diese beiden Anteile können untereinander gewichtet werden. Einerseits ist dies mittels konstanten Gewichts möglich, andererseits ist die Gewichtung auch abhängig vom Fehler möglich. Des Weiteren ist vorstellbar die Einzelteile des Rewards mittels Funktionen zu beeinflussen. So kann die Änderung der Stellgröße mit einer Wurzelfunktion verrechnet werden, damit bereits einer leichten Schwingung um einen konstanten Wert entgegengewirkt werden kann. Außerdem soll die zusätzliche Addition eines diskreten Bonus auf den Reward bei unterschreiten verschiedener Fehlertoleranzen getestet werden.

### Wahl der Reglerarchitektur

Zur erfolgreichen Erstellung eines Reglers muss eine geeignete Architektur für den Regelkreis gefunden werden. Dabei kann in zwei Gruppen unterschieden werden: In der einen Gruppe wird ein klassischer Regler genutzt, dessen Parameter adaptiv angepasst werden. Die angepassten Parameter werden dabei mittels RL gelernt. Die Stellgröße  $u$ , die auf das System gegeben wird, wird dabei mit dem Regler berechnet. In dieser Arbeit wird als Vertreter dieser Gruppe ein PID Regler genutzt, dessen Parameter mit einer festen Zykluszeit geupdatet werden. Ein Vorteil dieses Ansatzes ist es, dass der Regler mit einer geringeren Zykluszeit arbeiten kann als die Auswertung des RL-Algorithmus stattfindet. Dieser Ansatz ist in einem Blockschaltbild in Abbildung 3.2 dargestellt und wird im weiteren Verlauf als *ControllerParams*-Ansatz bezeichnet. Das RL-Interface im Blockschaltbild ist dabei für die Berechnung der Observations und Rewards verantwortlich und bildet die Softwareseite des Hardwaresystems.

Der zweite Ansatz erlernt  $u$  direkt und ist deshalb nicht an die Dynamik des PID-

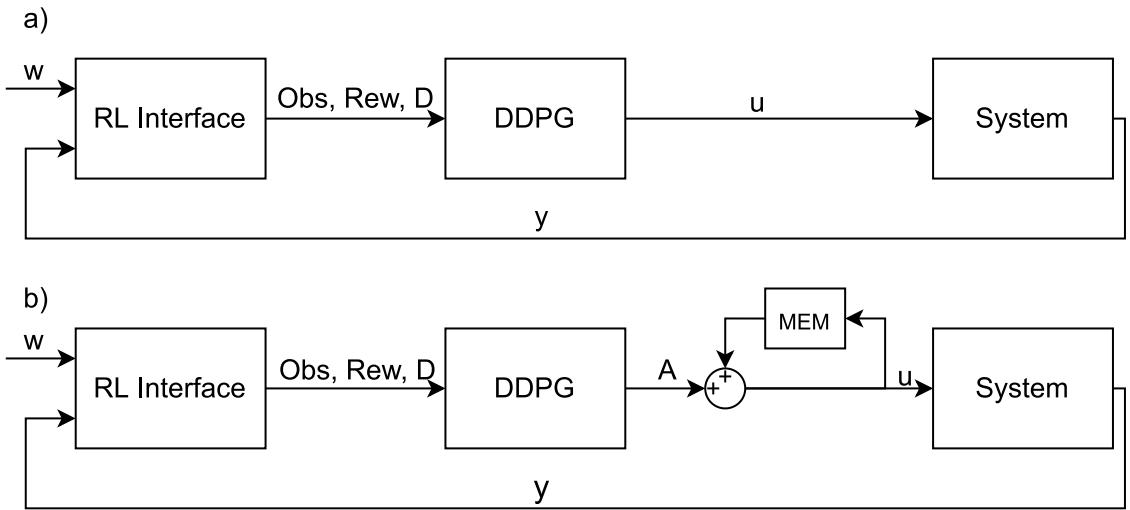


Abbildung 3.3: a) Direktes Erlernen der Stellgröße. b) Erlernen der Änderung der Stellgröße.

Reglers gebunden. Dieser Ansatz wird weiter in zwei Typen unterteilt. Im ersten Fall, abgebildet in Abbildung 3.3 a), wird in jedem Zeitschritt  $u$  berechnet und direkt auf das System gegeben. Hierbei besteht aber ein Problem:

Ist die Observation nur vom Regelfehler 3.10 abhängig hat der RL-Algorithmus keine Einsicht darüber wo sich das System im Arbeitsraum befindet und kann keine stationäre Genauigkeit erreichen. Da die Strecke auch keinen integrierenden Anteil besitzt, kann die stationäre Genauigkeit auch nicht durch die Strecke erreicht werden. Zur Lösung dieses Problems kann als zusätzlicher Teil der Observation die letzte Stellgröße hinzugefügt werden. Dabei ist aber nicht gegeben, dass das System den Zusammenhang zwischen  $u$  und  $u_{t-1}$  erlernt.

Ein andere Vorgehensweise ist, dieses Kriterium direkt in die Architektur einzubauen. So wird sich die letzte Stellgröße gespeichert und es wird nur die Änderung der Stellgröße gelernt. Diese Änderung wird anschließend auf die letzte Stellgröße aufaddiert. Dies ist in Abbildung 3.3 b) zu erkennen. Diese beiden Ansätze werden als *DirectControl* bezeichnet.

## 3.2 Systemidentifikation mit neuronalen Netzen

In der Einleitung wurden bereits einige Nichtlinearitäten erklärt, die das zu identifizierende System besitzt. Diese - und mögliche weitere - Nichtlinearitäten machen eine traditionelle lineare Systemidentifikation aufwendig. Mit dem Aufbau ist es allerdings

sehr leicht Daten aufzunehmen, was einen datenbasierten Ansatz sinnvoll erscheinen lässt. Wir haben uns für die Verwendung von Modellen aus der NARX -Klasse entschieden. Diese können die Nichtlinearitäten abbilden und lassen sich unter anderem als neuronale Netze formulieren und machen auf diesem Wege eine datenbasierte Identifikation möglich.

### 3.2.1 NARX-Modelle

NARX(*Nonlinear AutoRegressive eXogenous*)-Modelle sind nichtlineare autoregressive Modelle mit exogenem Eingang. Allgemein können Sie folgendermaßen dargestellt werden:

$$y_t = F(y_{t-1}, y_{t-2}, y_{t-3}, \dots, u_{t-1}, u_{t-2}, u_{t-3}) + \varepsilon$$

wobei  $y_t$  der Ausgang und  $u_t$  der exogene Eingang zum Zeitpunkt  $t$  sind. Die Variable  $\varepsilon$  ist ein Fehlerterm, der von uns vernachlässigt wird. Da der Systemausgang unter anderem von vergangenen Ausgängen abhängt, handelt es sich um eine Systemdarstellung auf Basis von Differenzengleichungen. Die Funktion  $F$  kann hier eine beliebige nichtlineare Funktion sein. In der Struktur dieser Funktion unterscheiden sich auch die verschiedenen Varianten von NARX-Modellen. In Frage kommen zum Beispiel radiale Basisfunktionen oder Sigmoidfunktionen. Viele dieser Varianten werden in [5] vorgestellt und diskutiert. Ein Regelungsansatz, der auf NARX-Modellen basiert wird in [9] vorgestellt. Ein etwas praktischerer Einsatz findet sich in [2]. Hier kommt dient ein NARX-Modell als Ersatz für das echte System zum Einsatz, um daran einen Machine-Learning basierten Regler zu trainieren. Allgemein finden NARX-Modelle oftmals Anwendung in der Analyse von Zeitreihen und der nichtlinearen Modellbildung. In unserem Projekt tritt an die Stelle von  $F$  ein neuronales Netz, das i.A. auch nur eine nichtlineare Funktion ist. Was für Vor- und Nachteile diese Herangehensweise im Vergleich zu „klassischen“ NARX-Modellen bietet, soll im folgenden Abschnitt gezeigt werden.

### 3.2.2 NARX als neuronales Netz

NARX auf Basis neuronaler Netze wird auch als NN-NARX bezeichnet. Im Vergleich zu anderen NARX-Methoden hat NN-NARX auch einige Nachteile, was die Methode in der klassischen Systemidentifikation eher unbeliebt macht[5]:

- Die Ergebnisse können schlechter interpretiert werden, da ein neuronales Netz mit nichtlinearen Aktivierungsfunktionen im Allgemeinen eine Blackbox ist, aus

der sich Systemeigenschaften kaum ableiten lassen.

- Die Auswahl der Lags (also der zeitverzögerten Ein- und Ausgänge) kann im Gegensatz zu anderen NARX-Methoden nur durch Probieren erfolgen. Hierbei ist der Suchraum groß, und die Auswertung jeder Kombination langwierig, da jeweils ein neuronales Netz trainiert werden muss.

Die Struktur eines NARX-Netzes ist in Abbildung 3.4 dargestellt. Die unterschiedlichen Methoden, mit denen ein solches Netz trainiert werden kann, werden in Abschnitt 5.2.3 erklärt.

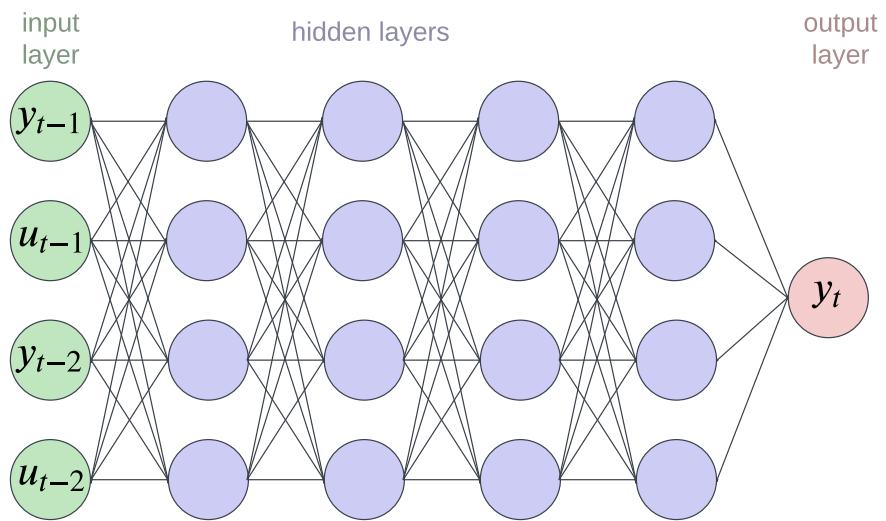


Abbildung 3.4: NARX als neuronales Netz

### 3.2.3 ANARX als neuronales Netz

ANARX (für *Additive*-NARX) ist eine Sonderform von NARX, die allgemein folgendermaßen dargestellt werden kann:

$$y_t = F_1(y_{t-1}, u_{t-1}) + F_2(y_{t-2}, u_{t-2}) + \cdots + F_n(y_{t-n}, u_{t-n}) + \varepsilon$$

$F_1, \dots, F_n$  sind wiederum beliebige nichtlineare Funktionen. Auch hier können wieder neuronale Netze für diese Funktionen eingesetzt werden. Wie die gesamte Struktur als neuronales Netz aussehen könnte ist in Abbildung 3.5 schematisch dargestellt. ANARX hat gegenüber NARX den entscheidenden Vorteil, dass es in eine Zustandsraumform umgewandelt werden kann. Dieser Zusammenhang macht die Methode für unsere

Zwecke sehr interessant, und wird in [15] bewiesen. In [30] und [32] werden verschiedene Ansätze vorgestellt, mit denen auf Basis eines solchen Modells Regler entworfen werden können. Einer dieser Ansätze wird in Abschnitt 3.2.5 genauer erklärt. Wie gut dieser Ansatz am echten System funktioniert, wird in Abschnitt 5.2.4 erläutert. Die Idee hinter der Umwandlung in die Zustandsraumform wird im nächsten Abschnitt 3.2.4 kurz erklärt.

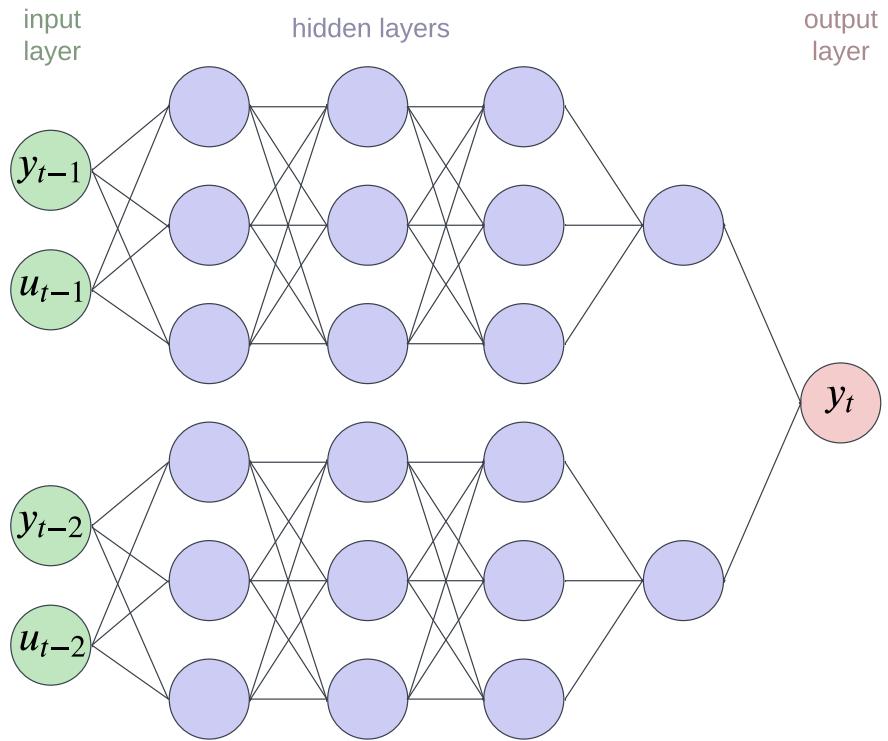


Abbildung 3.5: ANARX als neuronales Netz

### 3.2.4 Umwandlung in Zustandsraumform

Im letzten Abschnitt war bereits die Rede davon, dass sich die *ANARX*-Modellstruktur in eine (i.A. nichtlineare) Zustandsraumdarstellung umwandeln lässt. Diese Umwandlung funktioniert folgendermaßen: Allgemeine *ANARX*-Modelle der Ordnung  $k$  haben die Form

$$y(t_{n+1}) = N_1(y(t_n), \mathbf{u}(t_n)) + N_2(y(t_{n-1}), \mathbf{u}(t_{n-1})) + \cdots + N_k(y(t_{n-k+1}), \mathbf{u}(t_{n-k+1})),$$

wobei  $N_1, \dots, N_k$  beliebige Funktionen (in unserem Fall neuronale Netze),  $y(t)$  der Modellausgang und  $\mathbf{u}(t)$  die Modelleingänge zum Zeitpunkt  $t$  sind.

Um dieses Modell in Zustandsraumform anzugeben, darf der nächste Ausgang  $y(t_{n+1})$  und der nächste Zustand  $x(t_{n+1})$  nur vom Zustand  $\mathbf{x}(t_n)$  und den Eingängen  $\mathbf{u}(t_n)$  im Zeitschritt  $t_n$  abhängen. Wählt man die Zustände und den Ausgang wie folgt, ist diese Forderung erfüllt:

$$\begin{aligned} y(t_{n+1}) &= x_1(t_{n+1}) = x_2(t_n) + N_1(y(t_n), \mathbf{u}(t_n)) \\ x_2(t_{n+1}) &= x_3(t_n) + N_2(y(t_n), \mathbf{u}(t_n)) \\ &\vdots \\ x_{k-1}(t_{n+1}) &= x_k(t_n) + N_{k-1}(y(t_n), \mathbf{u}(t_n)) \\ x_k(t_{n+1}) &= N_k(y(t_n), \mathbf{u}(t_n)) \end{aligned}$$

Setzt man die zeitverschobenen Zustände aus dieser Gleichung in den Ausdruck für  $y(t_{n+1})$  ein wird die Äquivalenz zur Formel klar.

Zu beachten ist, dass diese Form des ANARX-Modells zwar die Definition einer Zustandsraumdarstellung erfüllt, die Zustände allerdings nicht physikalisch interpretierbar sind.

Benötigt man die Ableitungen dieses Modells nach den Zuständen bzw. Eingängen (also die Jacobimatrix) kann diese i.A. auch nicht analytisch angegeben werden. Im Fall von *NN-ANARX* können die expliziten Gradienten jedoch typischerweise über die Autograd-Funktion des verwendeten Machine-Learning-Frameworks bestimmt werden. In PyTorch könnte das wie folgt aussehen:

```
import torch
from ANARX import LAGNET

# Beispieldaten
x = torch.tensor([2.], requires_grad=True)
u = torch.tensor([3.], requires_grad=True)

# Hier wird als Beispiel ein Subnetz aus der ANARX-Klasse erzeugt
# N kann eine beliebige tensorwertige Funktion sein
N = LAGNET(2, n_hidden=2, layersize=3, afunc=torch.tanh, bias=True)

result = N(torch.cat((x,u))) # Auswertung von N
result.backward() # Beim Backward-Pass bestimmt Autograd die Gradienten

# Die Gradienten werden in der .grad-Property der Tensoren gespeichert
print(x.grad)
```

```

print(u.grad)

# Die Jacobimatrix kann direkt mit diesem Befehl bestimmt werden
J = torch.autograd.functional.jacobian(N, torch.cat((x,u)))

# Die Ergebnisse der Methoden stimmen berein
assert torch.equal(J.squeeze(), torch.cat((x.grad,u.grad)))

```

### 3.2.5 Regelungsansatz auf Basis von ANARX

Es gibt eine Reihe von Ansätzen um Systeme mithilfe von ANARX-Modellen zu regeln. In [31] wird beispielsweise eine Ausgangs-Feedback-Linearisierung auf Basis von ANARX-Modellen mithilfe eines weiteren neuronalen Netzwerks vorgestellt. In [35] findet sich ein Ansatz zur Zustandsrückführung basierend auf der Zustandsraumdarstellung von ANARX-Modellen. Der hier vorgestellte Ansatz stammt aus [30]. Da für diesen Ansatz neben den ANARX-Modellparametern nur der Ausgang des echten Systems benötigt wird, handelt es sich um eine Art der Ausgangsrückführung. Nun sei  $\mathbf{x}(t)$  der Zustand eines mittels ANARX beschriebenen Systems, entsprechend der in Abschnitt 3.2.4 beschriebenen Form. Der Ausgang  $y(t)$  des Systems, wird also folgendermaßen bestimmt:

$$y(t_{n+1}) = x_2(t_n) + N_1(y(t_n), \mathbf{u}(t_n)).$$

Ziel des Regleransatzes ist es nun,  $\mathbf{u}(t)$  so zu bestimmen, dass  $y(t)$  einen Zielwert  $\eta(t)$  annimmt. Es soll also gelten:

$$\eta(t_{n+1}) = x_2(t_n) + N_1(y(t_n), \mathbf{u}(t_n))$$

bzw.

$$(x_2(t_n) + N_1(y(t_n), \mathbf{u}(t_n))) - \eta(t_{n+1}) = 0.$$

Da alle Parameter und Argumente dieser Formel zur Laufzeit im Zeitschritt  $t$  bekannt sind bzw. mithilfe des Modells bestimmt werden können, kann der (im Sinne des Modells) optimale Eingang  $\mathbf{u}(t)$  als Ergebnis einer Optimierung bestimmt werden. Diese Vorgehensweise hat trotz ihrer bestechenden Einfachheit einige Nachteile:

1. Da der optimale Systemeingang  $\mathbf{u}$  im Sinne des ANARX-Modells bestimmt wird, hängt die Güte des Reglers direkt von der Güte des Modells ab.
2.  $\mathbf{u}(t)$  wird so bestimmt, dass das Modell im nächsten Zeitschritt den Wert  $\eta_{t+1}$

erreicht. Bei Sprüngen von  $\eta$ , denen das echte System nicht folgen kann, ergeben sich also nicht-realisierte Werte für  $\mathbf{u}$ .

3. In jedem Zeitschritt muss ein Optimierungsproblem für  $\mathbf{u}$  gelöst werden. Dies ist unter Umständen rechenaufwändig und macht damit den Einsatz auf eingebetteten Systemen unattraktiv.
4. Da im Allgemeinen nichts über die Funktion  $N_1$  bekannt ist, ist es i.A. nicht garantiert, dass das Optimierungsproblem für  $\mathbf{u}$  eine Lösung besitzt.

Um die letzten beiden Probleme zu lösen wird in [30] vorgeschlagen, für  $N_1$  eine lineare Funktion zu verwenden. In diesem Fall besitzt die Gleichung oben immer eine Lösung, die auch direkt bestimmt werden kann. Die Funktionen  $N_2, \dots, N_k$  können weiterhin beliebige nichtlineare Funktionen sein. In unseren Experimenten hat sich das lineare Netz am ersten Lag nicht negativ auf die Modellqualität ausgewirkt. Diese

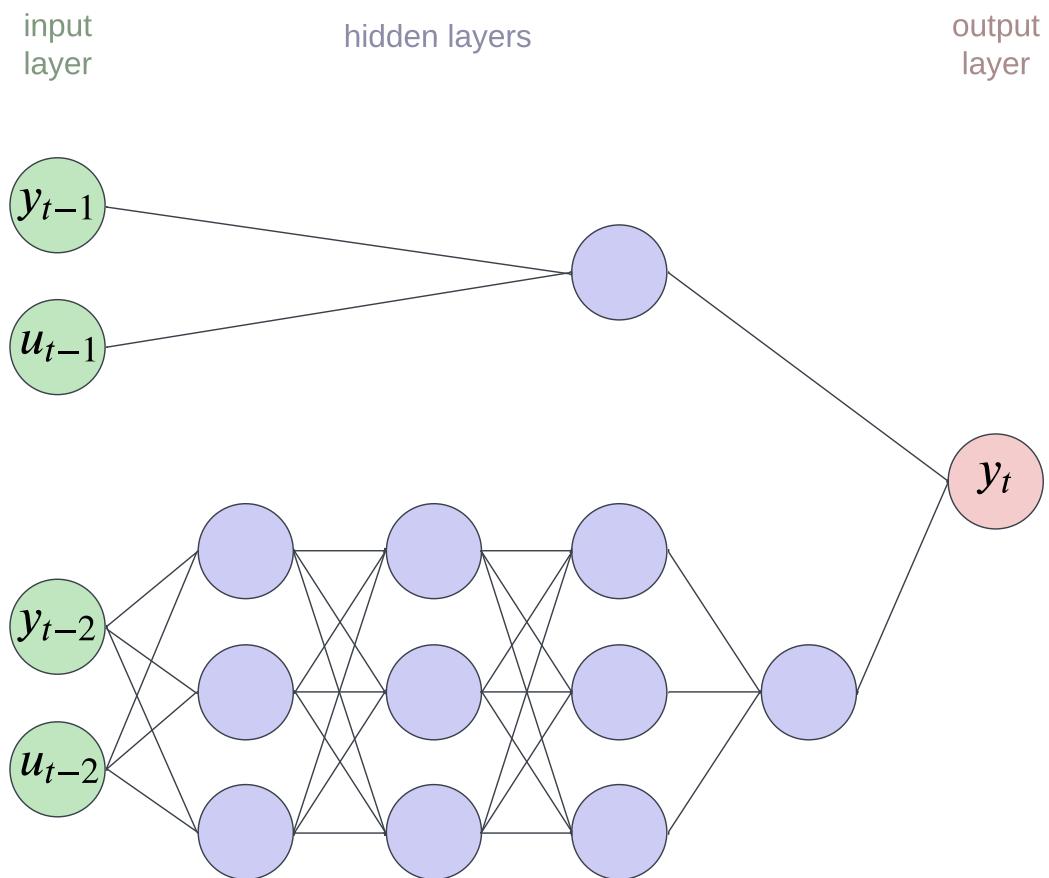


Abbildung 3.6: SANARX als neuronales Netz

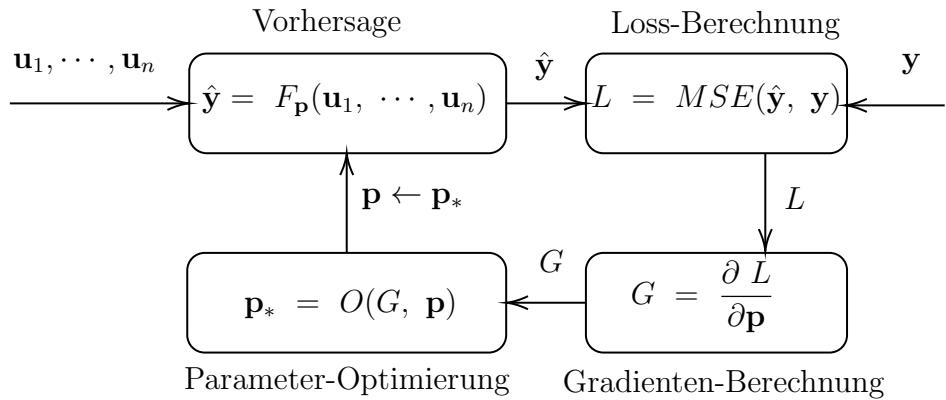


Abbildung 3.7: Trainingsschleife

Modellstruktur wird in [30] *NN-SANARX* genannt und ist in Abbildung 3.6 schematisch dargestellt. Der geschilderte Regelungsansatz kam auch in diesem Projekt am echten System zum Einsatz (siehe Abschnitt 5.2.4).

### 3.2.6 Training

Der grundlegende Ablauf des Trainings für neuronale Netze in NARX-Struktur ist in Abbildung 3.7 dargestellt:

**Vorhersage** Anhand des (initial zufälligen) Modells wird eine Modellvorhersage  $\hat{\mathbf{y}}$  berechnet.

**Loss-Berechnung** Es wird eine Metrik(Loss)  $L$  bestimmt, die die Abweichung zwischen Modellvorhersage  $\hat{\mathbf{y}}$  und echtem Systemausgang  $\mathbf{y}$  charakterisiert.

**Gradienten-Berechnung** Die Ableitung  $G$  von  $L$  nach den Modellparametern  $\mathbf{p}$  wird bestimmt.

**Parameter-Optimierung** Ein Optimierungsverfahren passt die Parameter  $\mathbf{p}$  anhand von  $G$  an. Die Modellvorhersage sollte dem realen Systemausgang nun näher kommen.

Diese Schleife wird typischerweise wiederholt, bis die Loss-Metrik nicht mehr kleiner wird.

*Loss- und Gradientenberechnung* sowie die *Parameteroptimierung* sind in PyTorch bereits implementiert. Für die Optimierung gibt es die Wahl zwischen verschiedenen Optimierungsverfahren. In unserem Fall kommt *ADAM*[13] zum Einsatz. Bei der Berechnung des Loss kann man ebenfalls zwischen verschiedenen Funktionen wählen.

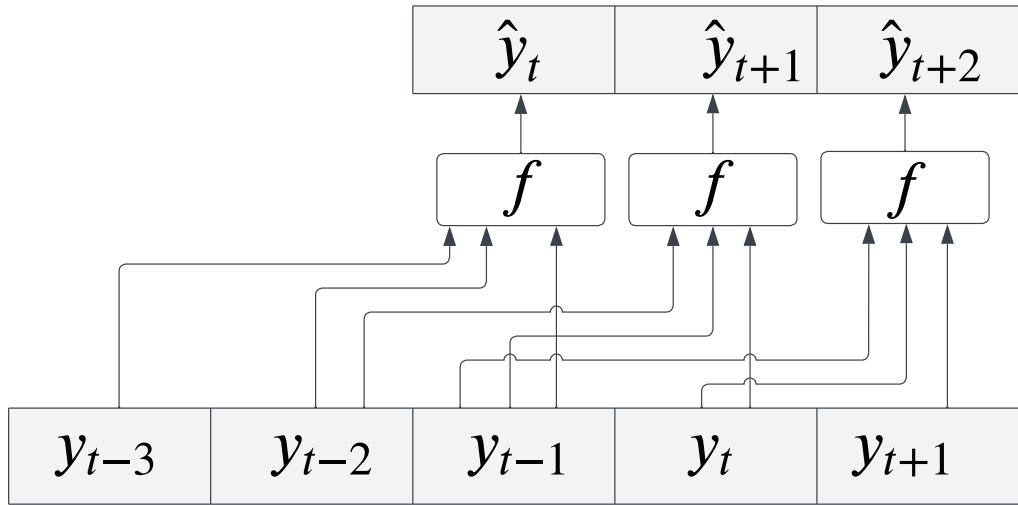


Abbildung 3.8: Schematische Darstellung der Open-Loop-Prädiktion

Hier wurde stets die, für Zeitreihen gut geeignete, mittlere quadratische Abweichung (MSE) gewählt. Gradienten kann PyTorch mittels *Autograd*[29] explizit berechnen. In Abschnitt 3.2.4 findet sich dazu ein Beispiel.

Zur Berechnung einer Vorhersage für das Training des Netzes kann grundlegend zwischen zwei Varianten unterschieden werden. Beide Varianten bringen ihre eigenen Vor- und Nachteile mit und sollen im Folgenden kurz beschrieben werden.

### Open-Loop Training

Der Unterschied zwischen Open-Loop- und Closed-Loop-Training (Abschnitt 3.2.6) liegt im Prädiktionsschritt. Genauer wird danach unterschieden, welche verzögerten Ausgänge für die Prädiktion verwendet werden. Beim Open-Loop Training wird der „simplere“ Ansatz gewählt, schlicht die Ausgangsmessung aus den Trainingsdaten zu verwenden (vgl. Abbildung 3.8). Das hat mehrere Vorteile:

- Die Eingangsdaten der jeweiligen Prädiktionsschritte sind von Beginn des Trainings an korrekt. Im Gegensatz dazu startet das Closed-Loop-Training (aufgrund der zufälligen Initialisierung des Netzes) mit falschen Zwischenergebnissen, die sich dann erst über das Training den richtigen Werten annähern.
- Alle zur Berechnung benötigten Daten stehen bereits von Beginn an zur Verfügung. Somit ist die Berechnung parallelisierbar und kann so deutlich beschleunigt werden.

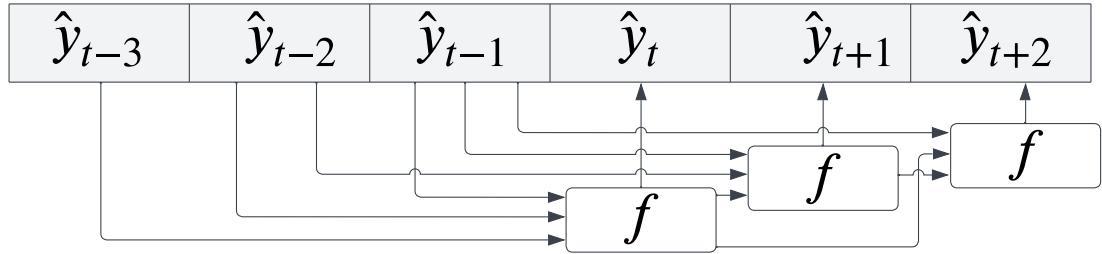


Abbildung 3.9: Schematische Darstellung der Closed-Loop-Prädiktion

- Es ist keine Backpropagation-Through-Time[11] nötig. Daher ist diese Methode etwas leichter zu implementieren und zu verstehen.

Weil aber immer nur ein Schritt in die Zukunft gelernt wird, ist die Methode auch weniger stabil. In unseren Experimenten hat sich gezeigt, dass Open-Loop-Training sinnvoll zu Beginn des Trainings eingesetzt werden kann, um schnelle Fortschritte zu erzielen. Ein wirklich gutes Trainingsergebnis erreichten wir allerdings nur, wenn wir danach noch Closed-Loop „nachtrainiert“ haben.

### Closed-Loop Training

Wie im letzten Abschnitt angedeutet, benötigt man Closed-Loop-Training für die bestmöglichen Trainingsergebnisse. Beim Closed-Loop-Training werden  $n$  Zeitschritte vorausberechnet (im Gegensatz zu nur einem Zeitschritt beim Open-Loop-Training) und dann über all diese Zeitschritte ein Loss berechnet (vgl. Abbildung 3.9). Damit die Gradienten dann über  $n$  Schritte hinweg korrekt bestimmt werden können, ist *Backpropagation-Through-Time*[11] notwendig. Dabei wird die Rückwärtssauswertung über mehrere Netzauswertungen hinaus ausgeführt. Gegenüber dem Open-Loop Training hat das folgende Vorteile:

- Das Trainingsverhalten ist deutlich stabiler, da immer direkt für die kompletten Trainingsdaten optimiert wird, und nicht sequentiell für Ausschnitte.
- Das Netz muss aus fehlerbehafteten Prädiktionen weitere möglichst präzise Prädiktionen machen. Damit wird im Training eine gewisse Robustheit gegen Fehler bei den Zwischenergebnissen erlernt.
- Dementsprechend ist die Trainingsmethode auch weniger anfällig für Overfitting, da die tatsächliche Systemdynamik gelernt werden muss.

Das Closed-Loop-Training sowie die Closed-Loop-Vorhersage sind aus diesem Grund allerdings auch nicht parallelisierbar: das Ergebnis zum Zeitschritt  $t_{n+1}$  ist vom Ergebnis zum Zeitschritt  $t_n$  abhängig. Für die Gradientenberechnung muss außerdem der Berechnungsweg als Baumstruktur im Speicher abgelegt werden [29]. Für lange Datenreihen wird beim Training also entsprechend viel Arbeitsspeicher benötigt. In unseren Experimenten hat das Closed-Loop-Training - wenn auch langsamer - stets deutlich bessere Ergebnisse als das Open-Loop-Training geliefert.

# 4 Software

In diesem Kapitel soll die erstellte Softwarelösung vorgestellt werden. Die Software für die RL-Komponenten ist dabei vollständig von dem Teil für (A)NARX unabhängig.

## 4.1 Softwarekomponenten für RL

Im Folgenden werden die verschiedenen Komponenten, welche softwareseitig für die Durchführung der RL-Experimente benötigt werden, vorgestellt. Die Komponenten werden in Abbildung 4.1 präsentiert. Diese zeigt die Softwareteile gegliedert nach der Hardware auf der sie ausgeführt werden.

Der User startet die Anwendung indem er einen *Runner* startet, welcher die Konfiguration von DDPG vorgibt. Das *Runner*-Skript ist dabei, wie jeder Softwareteil auf dem Entwickler-PC, in Python 3.8 geschrieben. Python bietet sich an, da bereits viele verschiedene Pakete für Deep Learning als auch für RL zur Verfügung stehen. In diesem Skript werden neben der Einstellungen aus Kapitel 3.1.2 unter anderem auch die Gestalt des Actor- und Critic-Networks und die Zeit, wie lange trainiert werden soll, festgelegt. Außerdem muss gewählt werden, ob der Trainingsvorgang auf dem Demonstrator oder in Simulation stattfinden soll.

Wenn das Training mit Daten vom Demonstrator stattfindet, sorgt das *Hardware Interface* dafür, dass die über das Beckhoff Protokoll *ADS* übertragenen Daten der SPS in einem für DDPG geeigneten Format zu Verfügung stehen. Außerdem bildet das Interface die Brücke zwischen dem Echtzeitsystem der SPS und dem nicht echtzeitfähigen Entwickler-PC. Der Entwickler-PC arbeitet mit einer durchschnittlichen Frequenz von 100 Hz. Diese wurde gewählt da bei schnelleren Frequenzen die Ausgabe der Berechnung des nächsten Systemeingangs einem zu großen Jitter unterliegt.

Die SPS hingegen arbeitet mit einer Frequenz von 4 kHz. Die Programmierung der SPS erfolgt mittels MATLAB/Simulink. Aufgabe der Software auf der SPS ist es den Motor anzusteuern, den DMS auszulesen und die Daten in einen FIFO-Puffer abzulegen, sodass diese über *ADS* ausgelesen werden können. Außerdem erstellt das *Simulink+Modell* die Führungsgröße auf welche geregelt werden soll. Des Weiteren beinhaltet das Modell

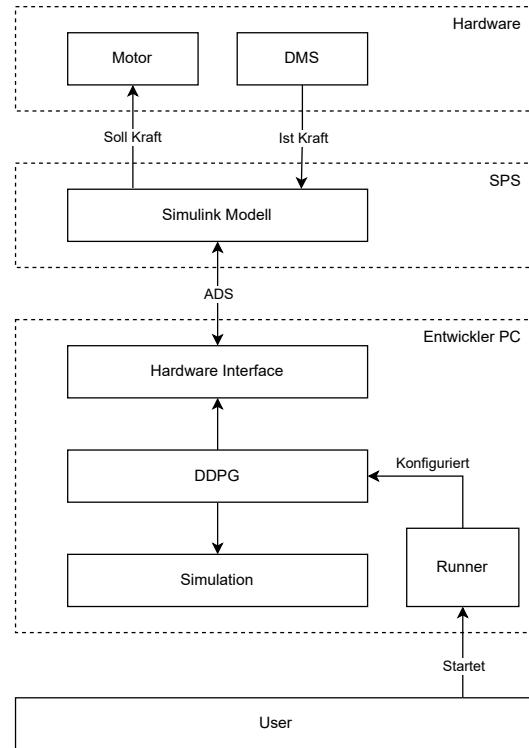


Abbildung 4.1: Schematischer Ablauf der Software

ein aus dem Seilroboter ausgekoppeltes vereinfachtes Sicherheitssystem. Dieses sorgt dafür, dass keine zu hohen Stellgrößen auf das System gegeben werden können. Außerdem kann ein Arbeitsraum vorgegeben werden. Wird dieser verlassen bremst der Motor selbstständig ab. Dieses Sicherheitssystem ist notwendig, da sich das Seil beim Trainingsvorgang aufschwingen und dadurch von den Seilrollen springen kann.

### 4.1.1 Architektur der RL Environments

In diesem Abschnitt soll weiter auf die Architektur der Software des Entwickler-PCs eingegangen werden. Das Hauptaugenmerk liegt dabei darauf, dass die Daten mit denen der Algorithmus arbeitet leicht austauschbar sind, sodass es einfach ist, zwischen Mess- und Simulationsdaten zu wechseln.

Anstatt DDPG selbst zu implementieren wird das Framework stable-baselines3 (sb3)<sup>1</sup> verwendet. Dieses stellt verschiedene RL Algorithmen zur Verfügung. Seit der Version 3 von stable-baselines wird PyTorch<sup>2</sup> als Backend für Anwendungen mit neuronalen Netzen verwendet. In dieser Arbeit wird zwar nur DDPG genutzt, aber durch einheitliche

<sup>1</sup><https://stable-baselines3.readthedocs.io>

<sup>2</sup><https://pytorch.org>

Interfaces in sb3 ist es schnell möglich auch andere RL-Algorithmen zu testen. Zur Nutzung von sb3 ist außerdem das Paket gym<sup>3</sup> erforderlich. Gym gibt dabei ein Interface vor, mit welchen die verschiedenen Environments für RL-Probleme standardisiert werden. Neben der Möglichkeit eigene Environments zu definieren werden auch einige Beispiele bereitgestellt. Darunter sind auch klassische Beispiele aus der Regelungstechnik wie das Aufschwingen eines inversen Pendels.

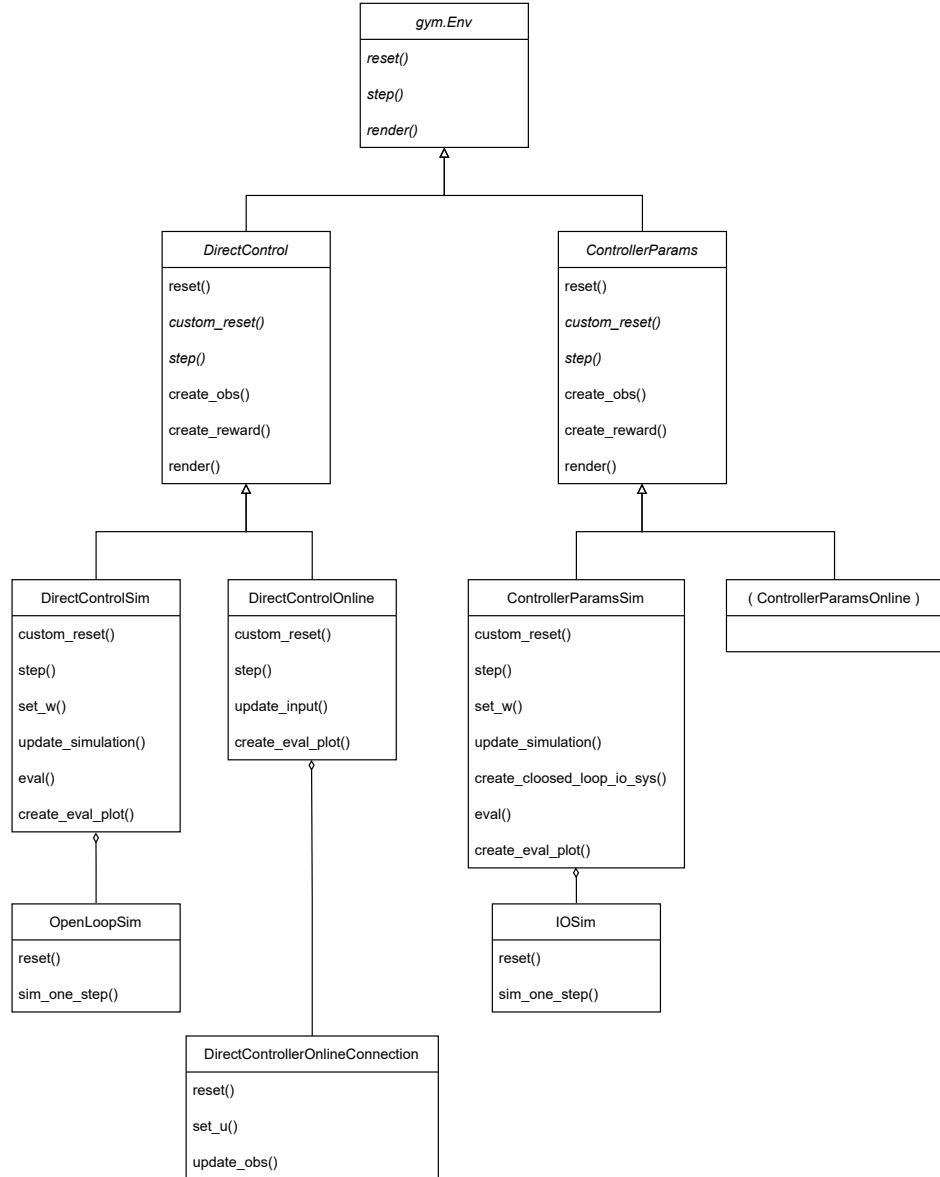


Abbildung 4.2: Schematischer Aufbau von DDPG

In Abbildung 4.2 ist ein Klassendiagramm der Software auf dem Entwickler-PC dargestellt. Oben ist die abstrakte *gym.Env* Klasse zu erkennen, von welcher alle weiteren

<sup>3</sup><https://gym.openai.com>

Environments erben müssen. Dazu ist folgendes nötig:

- Definition der Dimensionen und Grenzen des Aktions- und Beobachtungsraums als Klassenvariablen
- Implementierung von `reset()`: Zurücksetzen des Environments nach Abschluss einer Episode; gibt eine Observation zurück.
- Implementierung von `step()`: Ausführen eines Schritts auf dem Environment; Nimmt Action als Eingabe; Gibt Observation, Reward, Done (ist Episode abgeschlossen) und Info (mögliche extra Infos) zurück.
- Implementierung von `render()`: Möglichkeit schrittweise eine Bildschirmausgabe auszugeben.

Da sich die Berechnung der Observations, Rewards und des Systemausgangs zwischen den Ansätzen *DirectControl* und *ControllerParams* stark unterscheiden werden diese in zwei verschiedene abstrakte Klassen aufgeteilt. Beide Klassen beinhalten dabei Methoden mit denselben Aufgaben aber unterschiedlicher Implementierungen. Dies beinhaltet zuallererst die Erstellung des Rewards und der Observations. Außerdem wird die `reset()`-Methode implementiert um allgemeine Dinge zurückzusetzen. Das Reinitialisieren von simulations- oder hardwarespezifischen Einstellungen erfolgt in einer dedizierten Methode `custom_reset()`, welche von den Kindern implementiert werden muss.

Die Klassen, welche tatsächlich instantiiert werden können, erben von *DirectControl* und *ControllerParams*. Diese Klassen sind speziell auf den Einsatz in der Simulation oder auf dem realen System abgestimmt. Dabei implementieren die Klassen die `step()`-Methode, welche sich je nach Einsatzzweck unterscheidet. Da jedoch die Erstellung der Observations und Rewards in der Elternklasse stattfindet, ist sichergestellt, dass hier kein Fehler beim Wechsel von der Simulation auf die reale Hardware stattfindet. Des Weiteren besitzt jede Klasse eine dazugehörige Simulation oder eine Online Connection, welche als Interface zwischen SPS und Entwickler-PC dient. Zur Evaluation und Veranschaulichung des aktuellen Trainingsstands besitzt jede Klasse eine Methode welche eine Episode aufnimmt und anschließend plottet.

### 4.1.2 Simulation

Zur schnelleren Identifikation von geeigneten und ungeeigneten Konfigurationen und zur Schonung des Hardwareaufbaus wurde eine Simulation implementiert. Dazu wird die Toolbox *Python Control Systems Library*<sup>4</sup> eingesetzt, welche unter anderem die Fähigkeit besitzt lineare als auch nichtlineare Systeme zu simulieren. Dabei ist die Definition des Systems entweder als Übertragungsfunktion oder auch im Zustandsraum möglich.

Zur Simulation des System wird ein lineares Systemmodell verwendet. Da die Simulation nur zur Vorauswahl dienen soll, ist ein solches Modell ausreichend. Der Gedanke dabei ist, dass, wenn kein Regler mit bestimmten Einstellungen für ein lineares System gefunden werden kann, werden diese Einstellungen auch nicht für ein nicht lineares System funktionieren. Im Falle, dass eine geeignete Konfiguration gefunden wurde, kann diese anschließend auf dem Aufbau getestet werden. Außerdem ist das Ziel des Ansatzes mit RL ein System zu regeln, ohne dass ein Modell erstellt werden muss. Deshalb soll hier nicht viel Zeit für die Erstellung eines Modell genutzt werden. Zur Erzeugung dieses vereinfachten Modells wird ein System 2. Ordnung verwendet. Dieses wurde anschließend mit der *Matlab System Identification Toolbox* auf eine Sprungantwort des Aufbaus gefittet.

Zur Simulation dieses Modells kann eine Schrittweite gewählt werden. Davon unabhängig ist außerdem eine Frequenz für das Eingangs- als auch für das Messsignal festlegbar. Standardmäßig ist hier eine Simulationsfrequenz *sim\_freq* von 12kHz, eine Sensorfrequenz *y\_freq* von 4kHz und eine Eingangsfrequenz *u\_freq* von 100Hz festgelegt. Die Sensor- und Eingangsfrequenz entsprechen dabei denen des realen Aufbaus.

Neben den eben genannten Gemeinsamkeiten bestehen aber starke Unterschiede zwischen den Simulationen für *DirectControl* und *ControllerParams*. Die Simulation von *DirectControl* ist performanter, da hier bis zur Neuberechnung des Systemeingangs nur die offene Kette simuliert werden muss. Somit wird das System jeweils für *sim\_freq/u\_freq* Schritte berechnet und dann angehalten. Daraufhin erfolgt das Update des Systemeingangs und die nächsten Schritte werden simuliert. Dabei muss darauf geachtet werden, dass sich bei jedem Update des Systemeingangs der Zustand der Simulation gemerkt wird, damit diese mit dem richtigen Startzustand weitergeführt wird.

Dagegen ist die Simulation für die *ControllerParams* aufwendiger. Hier wird mittels der Toolbox ein sogenanntes *Input/Output Systems*-Objekt des Modell aufgebaut. Dies funktioniert ähnlich wie in Simulink, die grafische Oberfläche ist jedoch nicht vorhanden.

---

<sup>4</sup><https://python-control.readthedocs.io/en/0.9.1/>

Dabei wird der Regler anhand der Übertragungsfunktionen der P, I und D Komponenten erstellt und mit dem restlichen System verknüpft. Diese Reglerübertragungsfunktionen werden anschließend bei dem Update durch DDPG angepasst und das Modell neu aufgebaut. Auch hier muss darauf geachtet werden, dass sich bei jedem Updateschritt die entsprechenden Zustände gemerkt und anschließend als Startzustände wiederverwendet werden.

## 4.2 Softwarebibliothek für (A)NARX

Im Rahmen dieses Projektmoduls wurde eine Softwarebibliothek entwickelt, welche die Erstellung, das Training und die Auswertung von (A)NARX-Modellen auf Basis des Machine-Learning-Frameworks *PyTorch*<sup>5</sup> ermöglicht. Diese soll im folgenden Abschnitt kurz vorgestellt werden.

### 4.2.1 Funktionalitäten

In der Bibliothek sind folgende Funktionalitäten implementiert:

1. Die Erzeugung beliebiger Konfigurationen von (A)NARX-Netzen, sowie die automatische Aufbereitung der Trainingsdaten für diese. Parameter sind zum Beispiel die Anzahl der Eingänge, die Anzahl der Lags, die Anzahl der Hidden-Layer, die verwendete Aktivierungsfunktion und viele weitere, die für Hyperparameterstudien interessant sein könnten.
2. Das Training mit beliebigen Zeitreihendaten, sowohl Open-Loop als auch Closed-Loop.
3. Die Closed-Loop-Vorhersage von Ausgängen anhand beliebiger Eingangsdaten.
4. Die Möglichkeit bereits trainierte Netze zu speichern und für späteres Training bzw. Auswertungen wieder zu laden.
5. Die Möglichkeit trainierte ANARX-Netze in die Zustandsraumform (Abschnitt 3.2.4) umzuwandeln und auch in dieser Form Zeitreihen vorherzusagen.

Diese Kombination von Funktionalitäten, die für dieses Projekt unerlässlich waren, bietet keine der vorhandenen Toolboxen. Durch die Verwendung von *Pytorch* ist es sogar

---

<sup>5</sup><https://pytorch.org>

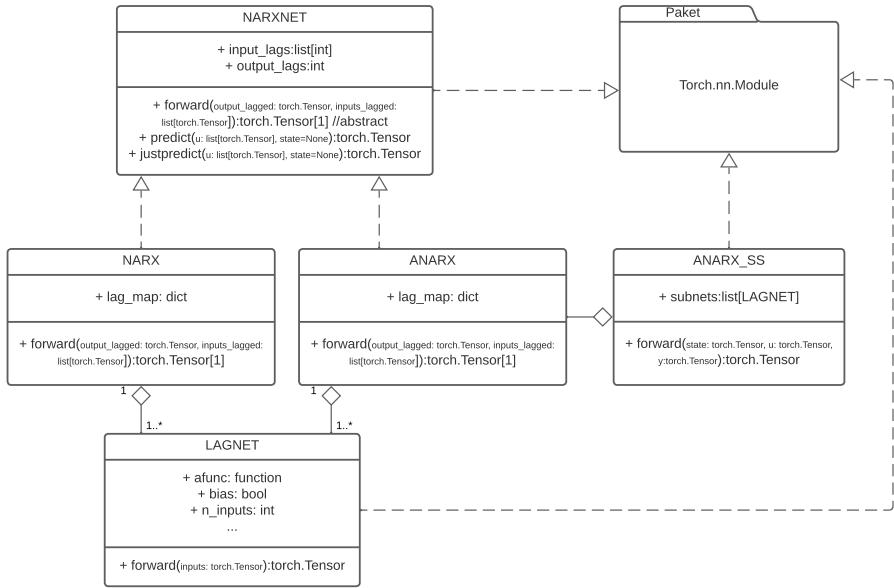


Abbildung 4.3: Aufbau der (A)NARX Softwarebibliothek

möglich alle Klassen der Bibliothek auch im ONNX-Format<sup>6</sup> zu exportieren, um sie auf nahezu beliebigen Endgeräten auszuwerten. So können die trainierten Modelle parallel zum realen System betrieben werden. Dies ermöglicht in erster Linie Regleransätze 3.2.5, es wäre aber z.B. auch ein Einsatz zur Fehlererkennung denkbar.

## 4.2.2 Architektur

In Abbildung 4.3 ist die Architektur der implementierten Bibliothek dargestellt. Es ist leicht zu erkennen, dass alle Klassen Ableitungen der `Torch.nn.Module`-Klasse sind. Damit können alle relevanten *PyTorch*-Funktionen wie Gradienten- und Lossberechnung, Laden und Speichern sowie der Export verwendet werden. Die `NARXNET`-Klasse besitzt die Funktionen `predict()` und `justpredict()`. Diese implementieren eine Mehrschrittvorhersage, die für Closed-Loop-Training -Vorhersage benötigt wird. Der einzige Unterschied zwischen den Funktionen ist, dass beim Aufruf von `predict()` Gradienten berechnet werden können, was für das Training nötig ist. Beim Aufruf von `justpredict()` nicht, für Vorhersagen ist das ausreichend. Die `forward()`-Funktion müssen die Kinder der `NARXNET`-Klasse selbst implementieren, da in dieser Funktion die konkrete Netzstruktur definiert wird. In der `forward()`-Funktion der `ANARX`-Klasse werden dann zum Beispiel die jeweiligen Subnetze (`LAGNET`-Klasse) ausgewertet. Die `ANARX_SS`-Klasse übernimmt die Subnetze eines Objekts vom Typ `ANARX` und stellt eine

<sup>6</sup><https://onnx.ai/>

`forward()`-Funktion bereit, mit der diese in der Zustandsraumform (Abschnitt 3.2.4) ausgewertet werden können. Training ist in dieser Form zwar nicht sinnvoll, allerdings ist diese Form für Auswertungen zur Laufzeit in Kombination mit dem in Abschnitt 4.2.1 erklärten ONNX-Export sehr nützlich.

### 4.2.3 Verwendung

Die Bibliothek integriert sich gut in typische Machine-Learning-Workflows: sie lässt sich komfortabel in *Jupyter-Notebooks* verwenden, und bietet gute Anbindungen für automatisierte Hyperparameterstudien (siehe auch Abschnitt 4.3.2). Im Folgenden soll ein kurzes Beispiel gegeben werden, wie eine Anwendung der Bibliothek aussehen könnte.

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import scipy.io
from torch.utils.data import TensorDataset, DataLoader
from tqdm import tqdm
from ANARX import ANARX
from utilities import lag_matrix

# Load some input data
data = scipy.io.loadmat("data/1803")
input = data["u2_t"]
out = data["y_t"]
# Create Torch Tensors and prepare data for OL-Training
out = torch.Tensor(out).squeeze()
input = torch.Tensor(input).squeeze()
in_lagged = lag_matrix(input, 6)
out_lagged = lag_matrix(out, 14)

# Create a SISO-SANARX model with 14 output lags and 6 input lags
model = ANARX(14, [6], n_hidden=3, layersize= 10,
               afunc=torch.tanh, SANARX=True)

# Open loop training
optim = torch.optim.Adam(model.parameters())
crit = nn.MSELoss()
dataset = TensorDataset(in_lagged, out_lagged, out)
```

```
loader = DataLoader(dataset, batch_size=20, shuffle=False)
for epoch in tqdm(range(200)):
    for il, ol, os in loader:
        optim.zero_grad()
        result = model(ol, [il])
        # print(result)
        loss = crit(result, os)
        loss.backward()
        optim.step()

# Closed Loop Training
optim = torch.optim.Adam(model.parameters())
crit = nn.MSELoss()
for epoch in tqdm(range(5)):
    optim.zero_grad()
    pred = model.predict([input])
    loss = crit(pred, out)
    loss.backward()
    optim.step()

# Compare model prediction to actual output in a plot
plt.plot(out.detach().numpy())
plt.plot(model.justpredict([input]).detach().numpy())

# Save the Model
torch.save(model, "example.pt")
```

## 4.3 Weitere Software

Neben Machine-Learning-Frameworks wie *PyTorch*<sup>7</sup> oder *KERAS*<sup>8</sup> hat die Machine-Learning-Community im Laufe der Zeit einige wichtige Tools entwickelt, die es erlauben das Training von neuronalen Netzen zu überwachen und zu automatisieren. Im Folgenden Abschnitt wird genauer auf zwei solcher Tools eingegangen, die im Rahmen dieses Projektmoduls viel eingesetzt wurden.

---

<sup>7</sup><https://pytorch.org/>

<sup>8</sup><https://keras.io/>

### 4.3.1 Tensorboard

Zur Überwachung der einzelnen Trainingsdurchgänge wird Tensorboard<sup>9</sup> verwendet, welches ursprünglich für Googles ML Framework Tensorflow entwickelt wurde. Inzwischen ist die Integration des Frameworks in *PyTorch* oder *sb3* aber möglich. Tensorboard bietet die Möglichkeit verschiedene Metriken während des Trainingsvorgangs zu loggen und diese in Echtzeit darzustellen. Diese Metriken müssen nicht zwingend numerische Werte sein, sondern können z.B. auch aus Abbildungen bestehen. So wurde während des Trainings von DDPG einerseits der Reward pro Episode geloggt. Andererseits wurde alle 5 Episoden ein Evaluierungsvorgang durchgeführt und eine dazugehörige Abbildung erstellt. Die erstellten Metriken lassen sich im Browser betrachten, wofür ein lokaler Server gestartet werden muss. Hier ist es außerdem möglich verschieden Trainingsruns miteinander zu vergleichen.

### 4.3.2 Weights and Biases

Bei der Durchführung von Parameterstudien ist Tensorboard nur bedingt geeignet, da zwar Runs miteinander verglichen werden können, aber es ist nur umständlich möglich die dazu passenden Konfiguration einzusehen und wieder herzustellen. Weights and Biases (wandb)<sup>10</sup> bietet genau diese vermissten Fähigkeiten. Für jeden Run der ausgeführt wird, wird die genutzte Konfiguration mit abgespeichert und kann wieder eingesehen werden. Im Gegensatz zum Tensorbaord erfolgt das Speichern der Ergebnisse in der Cloud, wobei die kostenlose akademische Lizenz mehr als genug Speicherplatz (100 GB) bietet.

Zudem ist es möglich sogenannte *sweeps* für einzelne Parameterstudien durchzuführen. Dafür müssen in einer Konfigurationsdatei die möglichen Werte der Parameter definiert werden. Anschließend wird der Sweep gestartet und alle möglichen Konfiguration werden ausgeführt und geloggt. Dabei kann die Kombinationsbildung entweder per Zufall oder per Rastersuche geschehen. Außerdem ist es möglich, dass die Runs auf verschiedenen Rechner ablaufen. Somit kann schnell die Rechenkapazität erhöht werden, ohne dass sich um die Synchronisierung der Systeme gekümmert werden muss. Die Ergebnisse dieser Runs werden auf einem Dashboard zusammengefasst und können dort ausgewertet werden. Dabei waren besonders die interaktiven *Parallele Koordinaten*-Plots hilfreich. Ein solcher ist hier zu finden: [https://wandb.ai/jubra97/Projektmodul2-ergebnisse\\_chapter\\_runner/sweeps/cm03q8y1](https://wandb.ai/jubra97/Projektmodul2-ergebnisse_chapter_runner/sweeps/cm03q8y1). Zur weiteren Analyse der Logs

<sup>9</sup><https://www.tensorflow.org/tensorboard>

<sup>10</sup><https://wandb.ai>

ist es außerdem möglich diese wieder per Python-API herunterzuladen und eine tiefere Durchsicht in Python durchzuführen.

# 5 Ergebnisse

Im folgenden Kapitel werden die Ergebnisse der verschiedenen Ansätze verglichen. Zuerst erfolgt eine Fokussierung auf den RL-Regler und dessen Konfigurationsmöglichkeiten. Daraufhin werden die Resultate des (A)NARX-Ansatzes vorgestellt. Zuletzt erfolgt ein Vergleich der Ergebnisse der beiden Ansätze auf dem Hardwareaufbau.

## 5.1 DDPG

Im Folgenden sollen neben den erzielten Ergebnissen durch einen mittels DDPG ausgelegten Regler auch die Auswirkungen der verschiedenen Einstellungsmöglichkeiten, welche in 3.1.2 vorgestellt wurden, untersucht werden. Wegen der vielen verschiedenen möglichen Konfigurationen werden diese zuerst in der vereinfachten Simulationsumgebung getestet. Die Ergebnisse können in folgendem WandB Projekt weiter nachvollzogen werden: [https://wandb.ai/jubra97/Projektmodul2-ergebnisse\\_chapter\\_runner](https://wandb.ai/jubra97/Projektmodul2-ergebnisse_chapter_runner).

### 5.1.1 Testaufbau

Die Simulation nutzt das in Kapitel 4.1.2 beschriebene Simulationsmodell. Wegen der vielen möglichen Kombinationsmöglichkeiten der Einstellungen ist das Ausprobieren aller Kombinationen nicht möglich. Stattdessen soll von einer funktionierenden Grundeinstellung ausgehend gezeigt werden, welche Auswirkungen die verschiedenen Parameter auf das Ergebnis des DDPG-Ansatzes haben. Die Grundeinstellungen, welche sich auf die Konfiguration des Environments aus Kapitel 3.1.2 und die dazugehörigen Hyperparameter beziehen, lauten:

- Observation:  $e, \dot{e}, \dot{u}$
- Reward:  $-(e + 25 * \sqrt{\dot{u}}) + discrete\_bonus$
- Actor Net: 2 Layer à 100 Neuronen; ReLU als Aktivierungsfunktion<sup>1</sup>; ohne Bias

---

<sup>1</sup>Da der Ausgang des Netzes zwischen [-1, 1] liegen muss, ist die letzte Aktivierungsfunktion Tanh

- Critic Net: 2 Layer à 200 Neuronen; ReLU als Aktivierungsfunktion; mit Bias
- Reglerarchitektur: *DirectControl* mit Aufaddieren der Stellgröße

Zur Verifikation dieser Konfiguration wurden drei Modelle trainiert. In Abbildung 5.1 sind die Sprungantworten dieser drei Modelle zu erkennen. Der RMSE zwischen der Führungsgröße  $w$  und des Systemausgangs  $y$  beträgt zwischen 0.016 und 0.0175, die Anstiegszeit (10% - 90%) zwischen 0.105s und 0.109s und die Einschwingzeit (5%-Band um Zielwert) zwischen 0.17s und 0.175s. Hierdurch ist zu erkennen, dass die gewählten Einstellungen wiederholbar solide Ergebnisse liefern. Das Actor- als auch das Critic-Netzwerk sind bewusst groß gewählt um eine Beeinträchtigung der Performanz durch nicht ausreichende Netzparameter zu verhindern. In einem späteren Abschnitt sollen die Auswirkungen anderer Netzgrößen, bei einer festen Konfiguration des Environments, untersucht werden.

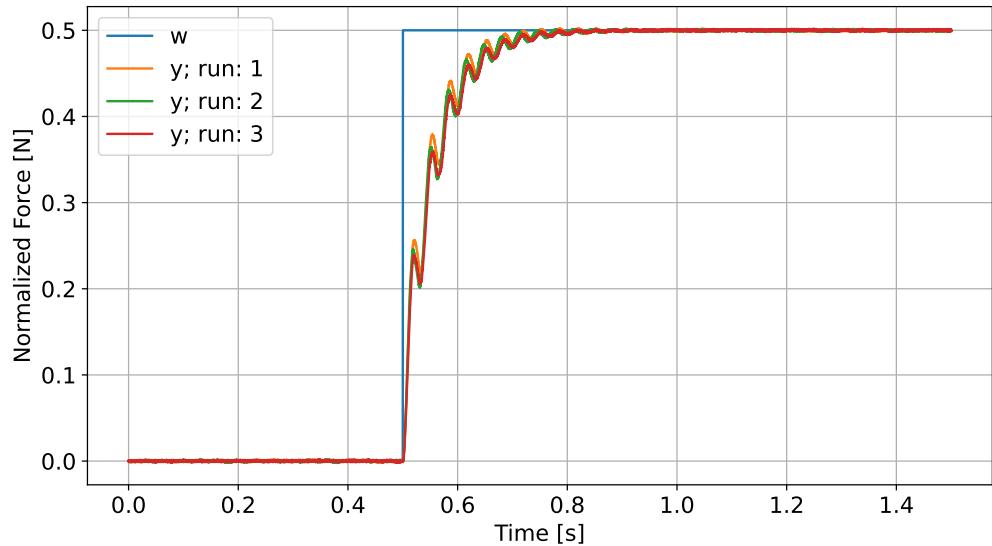


Abbildung 5.1: Vergleich der Sprungantworten dreier, mit der Grundkonfiguration trainierten, Modelle

Die weiteren verwendeten Hyperparameter, welche sich vorrangig auf DDPG beziehen, lauten:

- Trainingsdauer: 300.000 Schritte  $\hat{=} 2.000$  Episoden à 150 Schritte
- Action Noise: Normalverteilt; Linear abnehmend von  $\pm 0.1$  zu  $\pm 0.0003$  über die ersten 250.000 Schritte

- Learning Rate von 0.001
- Optimierer: Adam

Ausgehend von diesen Grundeinstellungen aus sollen nun die Auswirkungen der Umkonfigurationen gezeigt werden. Zur Evaluation der trainierten Netze wird der Regler mit verschiedenen Führungsgrößen angeregt. Insgesamt werden pro Netz 60 Trajektorien genutzt. Diese lassen sich wie folgt beschreiben:

- Sprünge von 0 zu Zielwert, insgesamt 20 Zielwerte zwischen 0 und 0.5 mit gleichen Abstand
- Rampe von 0 zu Zielwert, insgesamt 20 Zielwerte zwischen 0 und 0.5 mit gleichen Abstand, 0.25s Ansteigzeit
- Rampe von 0 zu Zielwert, insgesamt 20 Zielwerte zwischen 0 und 0.5 mit gleichen Abstand, 0.5s Ansteigzeit

Zur Berechnung der Anstiegs- und Einstellzeit wird bei den folgenden Betrachtungen der Mittelwert der Ergebnisse der Sprünge verwendet.

### 5.1.2 Wahl der Reglerarchitektur

Auch nach zeitaufwendigen Parameterstudien konnte für den *ControllerParams*-Ansatz keine erfolgreiche Konfiguration gefunden werden. Meist erfolgt durch das Training nur eine sehr geringe Änderung gegenüber den Startparametern. D.h., dass in seltenen Fällen ein Regler erstellt wurde, welcher keinen adaptiven Eigenschaften besitzt. Meist wurde jedoch ein Reglerparameter an das Maximum gezogen und entstand eine Dauerschwingung mit einer hohen Amplitude.

Ähnliches gilt auch für den *DirectControl*-Ansatz bei dem die Stellgröße für jeden Schritt neu berechnet werden soll. Auch hier konnte kein erfolgreiches Training erfolgen, der Systemeingang wird meist auf einen Extremwert (-1, 1) gezogen und ändert sich dann nicht mehr.

Im Gegensatz dazu ist das Training mit *DirectControl* und Erlernen der Änderung der Stellgröße erfolgreich. Deshalb wird im weiteren Verlauf auch nur dieser Ansatz betrachtet.

### 5.1.3 Wahl der Observation

Bei der Wahl der zu beobachteten Variablen gibt es zwei Hauptkriterien: Zum einen muss entschieden werden, ob der Fehler  $e$  oder die Soll-  $w$ , Stell-  $u$  und die Istgröße  $y$  einzeln

		False	True
Given Observation	e -	0.01 0.93	1.00 1.00
	e, $\dot{e}$ -	0.03 0.44	0.06 1.00
	e, $\dot{e}$ , $\ddot{e}$ -	0.11 0.23	0.31 0.59
	e, $\dot{e}$ , $\int e$ -	0.04 0.22	0.32 1.00
	e, $\dot{e}$ , u -	1.00 1.00	1.00 1.00
	e, $\dot{e}$ , y -	0.74 1.00	1.00 0.99
	e, $e_{t-1}$ -	0.00 0.75	0.06 1.00
	e, $e_{t-1}$ , $e_{t-2}$ -	0.07 0.47	1.00 1.00
	e, u -	1.00 1.00	1.00 1.00
	e, y -	1.00 1.00	1.00 0.97
w, $\dot{w}$ , u, $\dot{u}$ , y, $\dot{y}$ -	1.00 1.00	0.01 0.30	
w, u, y -	1.00 1.00	0.01 0.43	
$w_t, w_{t-1}$ - $u_t, u_{t-1}$ - $y_t, y_{t-1}$ -	1.00 1.00	0.01 0.65	
w, $w_{t-1}$ , $w_{t-2}$ - $u, u_{t-1}, u_{t-2}$ - $y, y_{t-1}, y_{t-2}$ -	0.01 0.25	0.01 0.45	

Abbildung 5.2: Vergleich verschiedener Observations: Der obere Eintrag zeigt die Anstiegszeit und der untere die Einschwingzeit. Links ist die jeweilige Observation angezeigt.

dem Netz übergeben werden. Zum anderen muss bestimmt werden, ob Ableitungen und Differentiale der Größen der Observation hinzugefügt werden sollen. Stattdessen ist es auch möglich die Observation mit dem zeitlichen Verlauf der Größen über die letzten  $n$  Zeitschritte zu erweitern. Verschiedene Kombinationen dieser Vorangehensweisen werden in Abbildung 5.2 gezeigt. Der jeweils obere Eintrag der Zellen zeigt die Anstiegszeit über die Evaluationsreihe und der untere Eintrag die Einschwingzeit.

Die Tabelle ist in der x-Achse außerdem danach gegliedert, ob das Actor-Netzwerk mit oder ohne Bias arbeitet. Der Bias ist ein zusätzlicher Eingang in jedes Neuron, welcher eine Konstante aufaddiert. Im Fall, dass die Observation den Fehler beinhaltet, ist dies aber nicht gewünscht. Denn dies würde bedeuten, dass bei  $e = 0$  eine Änderung der Stellgröße verursacht wird und somit ein konstanter Offset entsteht. Dieses Verhalten ist in der Tabelle daran zu erkennen, dass die Einschwingzeit 1s beträgt, was damit gleichzusetzen ist, dass der Zielwert nie erreicht wird. Dies passiert wenn der konstante Offset dafür sorgt, dass das 5%- Fehlerband um den Endwert nie erreicht wird.

Zum weiteren Vergleich sollen nun die jeweils besten Ergebnisse der Ansätze mit Fehler und Soll-, Stell- und Istgröße weiter untersucht werden. Diese werden um den Ansatz aus der Grundkonfiguration erweitert.

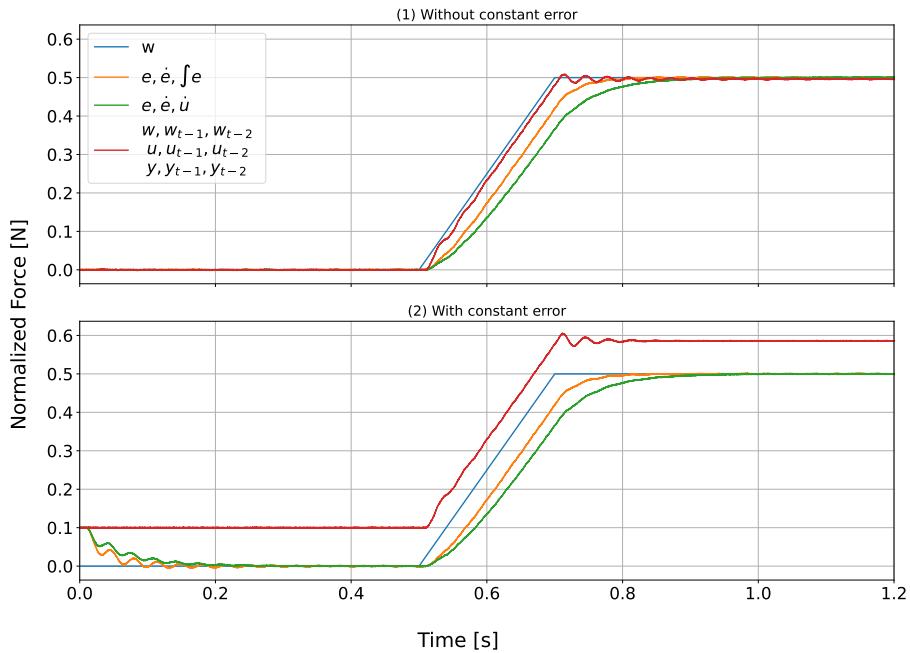


Abbildung 5.3: Vergleich verschiedener Observations auf eine Rampe als Systemeingang.  
Oben unter Trainingsbedingung. Unten mit einer künstlichen konstanten Störung.

In Abbildung 5.3 ist das Verhalten bei einer Rampe (0 auf 0.5 mit einer Anstiegszeit von 0.2s) als Sollgröße dargestellt. Im oberen Plot (1) ist zu erkennen, dass der Schleppfehler bei der direkten Vorgehensweise wesentlich geringer ist. Insgesamt besitzt dieser Ansatz eine schnellere Dynamik, wodurch auch ein leichtes Überschwingen entsteht. Außerdem erzielt der Ansatz ein sehr gutes Folgeverhalten. Das Konzept mit dem Fehler als Observation besitzt hingegen eine langsame Dynamik und kein Überschwingen. Eine Verringerung des Schleppfehlers scheint hier aber noch möglich.

Das große Problem des direkten Ansatzes tritt aber erst auf, wenn auf den Systemausgang ein künstlicher Fehler aufaddiert wird. Dies ist im Teil (2) der Abbildung zu erkennen. Dazu wird der Ausgang mit  $y = y + 0.1$  gestört. Durch die Störung entsteht ein konstanter Offset von ca. 0.1. Dies führt zum Schluss, dass durch reine Observation aus Soll-, Stell- und Istwerten eine Steuerung gelernt wird. Somit scheint das neuronale Netz die Durchführung eines Soll-Istwert-Vergleich nicht zu lernen. Wird dieser Vergleich aber explizit vorher durchgeführt wird zwingendermaßen ein Regler erlernt, welcher auch auf Störungen reagieren kann. Wie in (2) erkennen ist, kompensiert der erlernte Regler mit dem Fehler als Observation die Störung vollständig.

Tabelle 5.1: Auswahl der Parameter für die Konfiguration des Rewards

<i>osc_gain</i>	0	0.001	0.01	0.1	1	10	50	100
<i>error_fun</i>	square()	sqrt()	no_fun()					
<i>osc_fun</i>	square()	sqrt()	no_fun()					
<i>discrete_bonus</i>	True	False						
<i>osc_pen_dependent_on_error</i>	True	False						

### 5.1.4 Wahl des Rewards

Die Wahl des Rewards ist essentiell für einen erfolgreichen Trainingsprozess. Ein schlecht gewählter Reward kann dafür sorgen, dass mittels DDPG kein Regler gefunden werden kann. Außerdem lässt sich über den Rewards das Verhalten des Reglers beeinflussen. So kann etwa das Überschwingen des Reglers bei einem Sprung beschränkt werden. Die Rewardfunktion hat folgende Gestalt:

$$rew = \begin{cases} -(error\_fun(|e|) + osc\_gain \cdot osc\_fun(\frac{1}{|e|}|\dot{u}|)) & \text{if } osc\_pen\_dependent\_on\_error \\ -(error\_fun(|e|) + osc\_gain \cdot osc\_fun(|\dot{u}|)) & \text{else} \end{cases}$$

Dabei verrechnet sie den Fehler im aktuellen Schritt und die Änderung der Stellgröße. Durch die Verwendung der Stellgröße soll der Schwingungsneigung des Systems entgegen gewirkt werden. Die verwendeten Parameter sind in Tabelle 5.1 zusammengefasst. Dabei gibt *osc\_gain* eine Gewichtung zwischen dem Fehler und der Stellgrößenänderung an. Mittels *osc\_fun* und *error\_fun* kann eine Funktion gewählt werden, mit welcher eine noch feinere Gewichtung des Rewards erfolgen kann. So wird beispielsweise bei Verwendung der Wurzelfunktion für *osc\_fun* eine kleine Stellgrößenänderung überpropotional zu großen Stellgrößenänderung bestraft. Ist *osc\_pen\_dependent\_on\_error* aktiviert wird eine Stellgrößenänderung stärker bestraft, wenn das System einen kleinen Fehler besitzt. Die Idee hierbei ist, dass bei einem Sprung auf die Führungsgröße eine große Stellgrößenänderung erwünscht ist. Ist der Fehler jedoch gering führt eine große Stellgrößenänderung zu einem unerwünschten Schwingen um die Sollgröße. Ist außerdem *discrete\_bonus* aktiviert wird auf das Ergebnis der Funktion bei Unterschreitung bestimmter Fehlergrößen zusätzlich ein diskreter Bonus aufaddiert.

In Tabelle 5.1 sind die verschiedenen getesteten Komponenten der Rewardfunktion aufgeführt. Bei beliebiger Kombination ergeben sich 288 Möglichkeiten, welche auch alle getestet wurden. 215 dieser Kombinationen haben einen Regler erlernt, welcher eine Einschwingzeit von <1s besitzt. Bei Netzwerken, welche diese Eigenschaft nicht erfüllen, ist davon auszugehen, dass sie keinen Lernerfolg hatten. 67 der 73 Kombinationen

ohne Lernerfolg haben das Flag `osc_pen_dependent_on_error` aktiviert. Besonders in Verbindung mit einem hohen `osc_gain` ist erfolgreiches Training nicht möglich. So ist kein einziger Versuch mit einer Verstärkung von 100 gelungen. Aber auch drei Versuche mit einer Verstärkung von nur 0.001 konnten kein ausreichendes Ergebnis erzielen. Bei den restlichen sechs Kombinationen mit deaktiviertem `osc_pen_dependent_on_error` Flag ist ebenfalls eine hohe Verstärkung von 100 und 50 vorhanden, jedoch führt diese hohe Verstärkung nicht zwingend zu einem Fehlversuch. Erwähnwert ist außerdem, dass es mit deaktiviertem `osc_pen_dependent_on_error` Flag und aktiviertem `discrete_bonus` keinen einzigen Versuchen gab, bei dem gar nichts gelernt wurde.

Legt man nun sehr gute Ergebnisse fest, indem die Anstiegszeit  $< 0.01s$  und die Einschwingzeit  $< 0.25s$  betragen muss bleiben noch 54 Kombinationen übrig. Bei diesen gestaltet es sich als schwierig besondere Merkmale herauszuarbeiten. Wenige Auffälligkeiten gibt es aber: Das `discrete_bonus` Flag ist bei zwei von drei Kombinationen deaktiviert. Die `osc_fun` ist nur 20% der Fälle `square()`. Die restlichen 80% teilen sich die Einstellung ohne weite Verrechnung oder mittels `sqrt()`. Die `error_fun` ist hingegen in 25% der Kombinationen `sqrt()`. Mit ca. 43% ist hier `square()` am häufigsten Vertreten.

Die bevorzugten Funktionen zur Verrechnung sind erklärbar. Mittels der Bestrafung für hohe Geschwindigkeiten der Stellgröße sollen Schwingungen um den Sollwert verhindert werden. Wird jedoch diese Bestrafung quadriert, werden größere Änderung der Stellgröße überproportional stark bestraft. Dies soll aber nicht geschehen, da bei der Änderung der Istgröße eine hohe Dynamik des Reglers, und damit eine hohe Änderungsrate von  $u$ , erwünscht ist. Deswegen ist `square()` als `osc_fun` nicht geeignet. Hingegen ist `sqrt()` als `error_fun` ungeeignet. Durch die Funktion werden große Fehler überproportional gering bestraft. Dies ist aber in diesem Fall nicht passend, da es das Hauptziel ist den Fehler zu minimieren.

Insgesamt kann man zu dem Schluss kommen, dass `osc_pen_dependent_on_error` nicht verwendet werden sollte. Bei Aktivierung verstärkt sich die Gefahr eines nicht erfolgreichen Trainingsvorgangs stark. Auch bei den guten Ergebnisse ist dieser Parameter nicht überproportional oft aktiv. Außerdem kann festgehalten werden, dass `sqrt()` nicht als `error_fun` und `square()` nicht als `osc_fun` verwendet werden sollte. Der `discrete_bonus` macht das zwar Training robuster, allerdings haben vier der besten fünf Durchläufe das Flag deaktiviert. Dieses muss also zur letzten Optimierung ggf. deaktiviert werden. Zusammenfassend ist zu sagen, dass, wenn die in diesem Absatz genannten Anforderungen eingehalten werden, das Training gegenüber der Rewardfunktion sehr robust ist.

### 5.1.5 Hyperparameter

Einer der wichtigsten Hyperparameter für DDPG ist der Aufbau der Netze für das Actor- und das Critic-Netzwerk. Hierzu wurde eine Studie, mit dem Ziel die ideale Anzahl der Ebenen und die Anzahl der Neuronen pro Ebene herauszufinden, durchgeführt. Dazu wurden jeweils Netze mit ein oder zwei Ebenen und mit einer Anzahl zwischen ein und 400 Neuronen pro Ebene getestet. Besitzt das Actor Netzwerk nur 1 Neuron pro Ebene konnte in nur einer von zwölf Kombinationen ein gutes Ergebnis erzielt werden. Deshalb wurde sich für ein Actor-Netzwerk mit mindestens 10 Neuronen entschieden. Außerdem wurde der Einfluss des Action Noise untersucht. Durch diesen Vorgang kann auf die errechnete Action ein Rauschen aufaddiert werden um eine bessere Exploration zu gewährleisten. Dabei wurde ein konstantes gleichverteiltes Rauschen, ein lineare abnehmend gleichverteiltes Rauschen und ein *Ornstein Uhlenbeck* [34] Rauschen verwendet. Außerdem wurden verschiedene Verstärkungen des Rauschens untersucht. Insgesamt konnte dabei kein klarer Einfluss der Action Noise auf das Ergebnis des Lernvorgangs festgestellt werden. So hat die Performance nicht einmal dann klar abgenommen, wenn überhaupt kein Rauschen zur Exploration verwendet wurde.

### 5.1.6 Überführung auf den Hardwareaufbau

Zur Überprüfung der Simulationsergebnisse sollen diese nun auf den realen Aufbau übertragen werden. Aufgrund der ADS-Übertragung und der Auswertung der neuronalen Netze auf dem Entwickler-PC wird eine Updatefrequenz des Regler von 100 Hz verwendet. Des Weiteren wurden folgende Einstellungen verwendet:

- Observation:  $e, \dot{e}, \dot{u}$
- Reward:  $-(e + 0 - 1 * \sqrt{\dot{u}}) + discrete\_bonus$
- Actor Net: 2 Layer à 10 Neuronen; ReLU als Aktivierungsfunktion; ohne Bias
- Critic Net: 2 Layer à 200 Neuronen; ReLU als Aktivierungsfunktion; mit Bias
- Action Noise: Konstant mit 0.05
- Trainingsdauer: 50.000 Schritte  $\hat{=} 100$  Episoden à 500 Schritten bzw. 5s

Das Training auf dem realen System bleibt jedoch stark Startwertabhängig und ist in nur ca. 3 von 5 Durchläufen erfolgreich. Häufig muss der Trainingsdurchgang abgebrochen werden, weil es bei der Exploration zu starkem Aufschwingen kommt und hierdurch

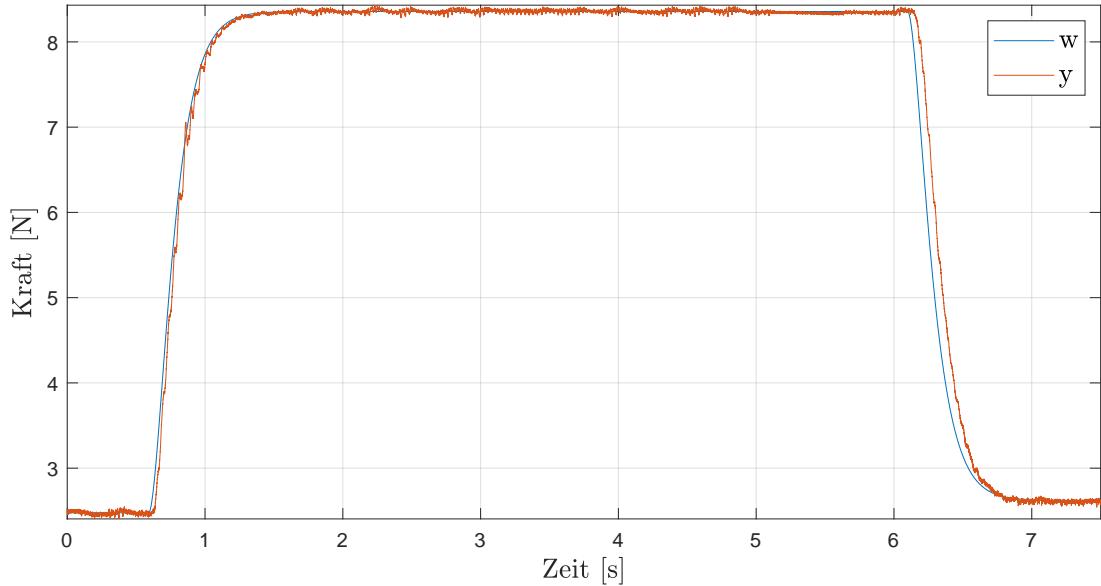


Abbildung 5.4: Systemantwort des geregelten Systems auf eine PT2-förmige Anregung

das Seil von den Führungsrollen springt. Ist das Training jedoch erfolgreich, erzielt der Regler sehr gute Ergebnisse.

In Abbildung 5.4 ist der Verlauf des Systemausgangs auf eine PT2-förmige Anregung der Führungsgröße zu erkennen. Der Schleppfehler beim Anstieg ist sehr gering und es kommt zu keiner bleibenden Regelabweichung. Beim Abfall der Kraft ist der Schleppfehler größer, da das Seil nicht auf Druck belastet werden kann und somit die Kraft nicht durch den Druck des Motors schneller abnehmen kann. Insgesamt ist eine sehr gute Reglerdynamik zu erkennen.

In Abbildung 5.5 ist außerdem die Robustheit des Reglers präsentiert. Um das System zu stören wird der DMS mit ca. 0.5Kg vorgespannt. Deshalb kann im gestörten Fall auch keine Kraft von unter 0.5Kg erreicht werden. Dies ist in Teil (a) an den Minima des Sinus zu erkennen. Außerdem ist auch hier wieder gut der Unterschied zwischen Zug und Druck auf das Seil zu erkennen. Während des Anstiegs des Sinus wird am Seil gezogen und der Schleppfehler ist sehr gering. Dagegen ist während des Abstiegs das Seil auf Druck belastet und kann nicht schneller nachgeben. Somit ist der Schleppfehler während des Abfalls der Sollkraft wesentlich größer. In Teil (b) ist die Stellkraft des Motors aufgezeigt. Da das gestörte System vorgespannt ist, fällt hier die Stellkraft geringer aus als beim ungestörten System. Wegen der Art und Weise wie die Vorspannung zustande kommt, lässt der Anteil der Vorspannkraft bei größerer Kräftewirkung auf den DMS nach. Dies ist dadurch zu erklären, dass mit einer größeren Kraft auch der DMS stärker

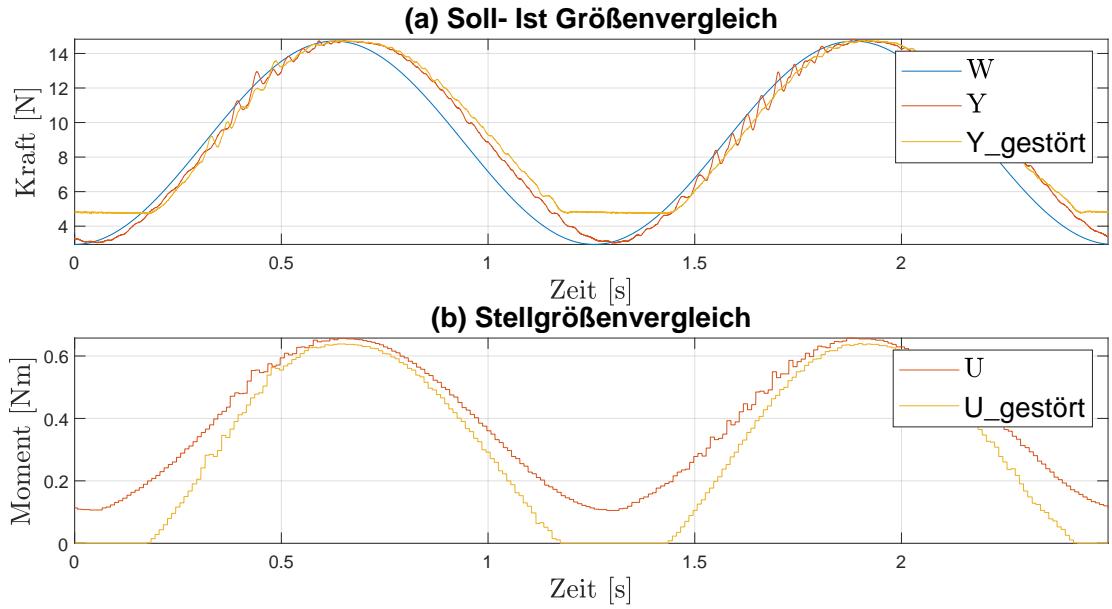


Abbildung 5.5: Systemantwort des geregelten (un)gestörten Systems auf eine Sinus-förmige Anregung: (a) Verlauf des Systemausgangs und der Führungsgröße. (b) Verlauf der Sollgröße

ausgelenkt wird. Die Vorspannung entsteht durch das Herabdrücken des DMS in Ruhe. Biegt sich der DMS nun durch die Kraft am Seil, lässt der in Ruhe eingestellte Druck nach. Deshalb ist die Stellkraft in den Maxima des Sinus fast identisch, während sie in den Minima deutlich voneinander abweicht.

Eine weitere Erkenntnis ist, dass keine Wiederholbarkeit gegeben ist. Das heißt, dass zwei erfolgreiche Trainingsdurchgänge mit derselben Konfiguration zu verschiedenen Ergebnissen führen können. In Abbildung 5.6 ist dies mithilfe einer Sprungantwort dargestellt. Dabei wurde der Regler *RL\_1* mit den gleichen Einstellungen wie der Regler *RL\_2* trainiert. Regler *RL\_1* hat dabei eine wesentlich stärkere Schwingungsneigung. Die Schwingung ist dabei so stark, dass die Einstellzeit ebenfalls um 0,3 s länger als bei Regler *RL\_2* ausfällt. Des Weiteren wurde geprüft, wie sich der Regler verändert, wenn die Kommunikation zur Auswertung des Actor-Netzwerks mit dem Entwickler-PC wegfällt. Dazu wurde das Netzwerk direkt auf der SPS ausgewertet. Der Regler *RL\_2\_SPS* ist der gleiche Regler wie der Regler *RL\_2* wird aber auf der SPS ausgewertet. Dadurch reagiert der Regler bereits einen Abtastschritt nach der Änderung der Stellgröße und nicht wie sonst erst nach zwei Abtastschritten (vgl. Abbildung 5.6 Detail). Insgesamt wird der Regler durch diese Herangehensweise dynamischer.

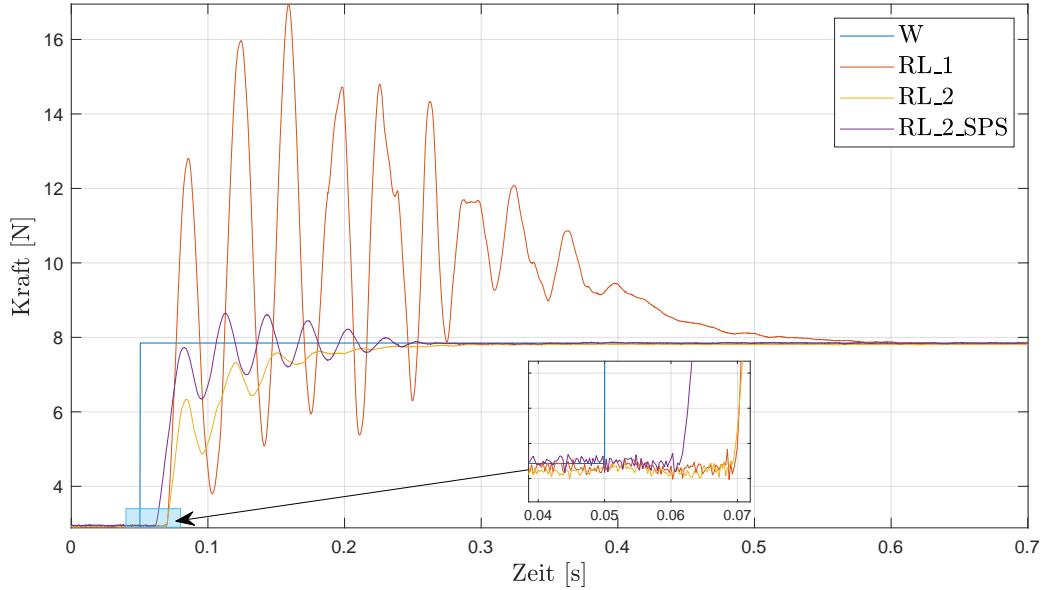


Abbildung 5.6: Sprungantwort verschiedener RL Regler

## 5.2 (A)NARX

In Abschnitt 4.2 wurde bereits die im Rahmen dieses Projektmoduls entwickelte Softwarebibliothek für neuronale Netze in der A(NARX)-Form vorgestellt. Diese Bibliothek wurde verwendet, um neuronale Netze auf Basis von Zeitreihen aus dem Teststand (siehe Abschnitt 2.2) zu trainieren. Wie die Hyperparameter für solche Netze gewählt wurden, wird in Abschnitt 5.2.2 erklärt. Anschließend wird in Abschnitt 5.2.3 darauf eingegangen wie die Netze trainiert wurden und wie gut sie als Modell die Dynamik des echten Systems replizieren können. Auf Basis eines solchen Modells wurde schließlich ein Regler (vgl. Abschnitt 3.2.5) für das reale System ausgelegt, der in Abschnitt 5.2.4 vorgestellt wird.

### 5.2.1 Aufbereitung der Daten

Für die Datenerhebung wurde dem Motor im Teststand (vgl. Abschnitt 2.2) eine Kraftvorgabe mit einer zufälligen Reihe von Sprüngen gemacht. Dabei wurde der gesamte Kraftbereich, den der Motor konstant liefern kann (ca. 0,95 Nm), abgedeckt. Dafür wurden zunächst die vorgegebene Sollkraft am Motor sowie die gemessene Kraft am DMS mit 4 kHz aufgezeichnet. Weil sich im Verlauf des Projekts gezeigt hat, dass die Modelle das Kriechverhalten nicht ausreichend replizieren konnten, wurde später zusätzlich der Positionswert des Motors aufgezeichnet. Für das Training wurden die

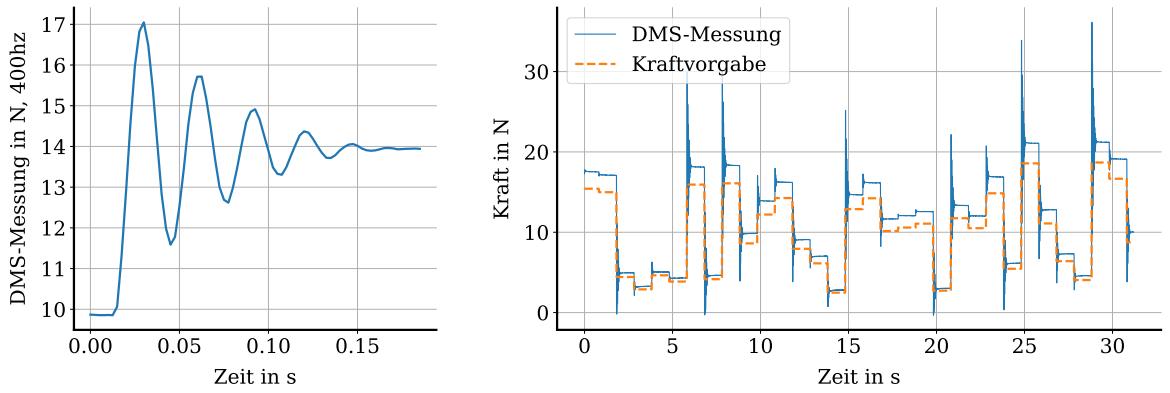


Abbildung 5.7: Mit 400Hz gemessene Sprungantwort des DMS (links), Gesamte Messreihe (rechts)

Daten normiert und die Auflösung auf 400 Hz reduziert. Dies ist die Auflösung, mit der das Schwingen des DMS-Messsignals nach Sprüngen gerade noch dargestellt werden kann. Die Anzahl der Messpunkte wurde reduziert, damit das Training des Modells wesentlich schneller abläuft. In Abbildung 5.7 ist diese Schwingung links mit einer Auflösung von 400 hz dargestellt. Auf der rechten Seite ist die gesamte Messreihe mit der Kraftvorgabe an den Motor und der resultierenden Kraft am DMS zu sehen. In einem Versuch das Kriechen des echten Systems besser zu modellieren (siehe auch Abschnitt 5.2.3), wurden Modelle zusätzlich mit dem Positionssignal des Motors trainiert. Um dabei zu verhindern, dass das Modell aus der Position fälschlicherweise die Höhe der Sprünge oder das Schwingverhalten lernt, wurden alle Sprünge aus den Positionsdaten entfernt. Abbildung 5.8 zeigt oben das ursprüngliche und unten das für das Training bereinigte Positionssignal einer Messreihe.

## 5.2.2 Wahl der Hyperparameter

Zur Wahl der Hyperparameter wurde eine Studie mithilfe von Weights and Biases (siehe Abschnitt 4.3.2) durchgeführt. Dabei wurden zunächst zufällige Kombinationen der Parameter aus Tabelle 5.2 getestet. Dabei wurden, über mehrere Rechner verteilt, im Laufe von etwa fünf Tagen etwa 100 verschiedene ANARX-Netze trainiert. Als sinnvolle Konfiguration, die zuverlässig gute Trainingsergebnisse lieferte, wurden die fett dargestellten Parameter gewählt. Basierend auf dieser wurde dann eine weitere Hyperparameterstudie, dieses mal mit dem Fokus auf der Wahl geeigneter Lags, durchgeführt. Es wurden dabei etwa 430 ANARX-Netze, mit zufälligen Kombinationen aus Eingangslags zwischen 1 und 10, und Ausgangslags zwischen 1 und 20 trainiert. Dabei hat sich ergeben, dass es für das betrachtete System sinnvoll ist, mehr als 4

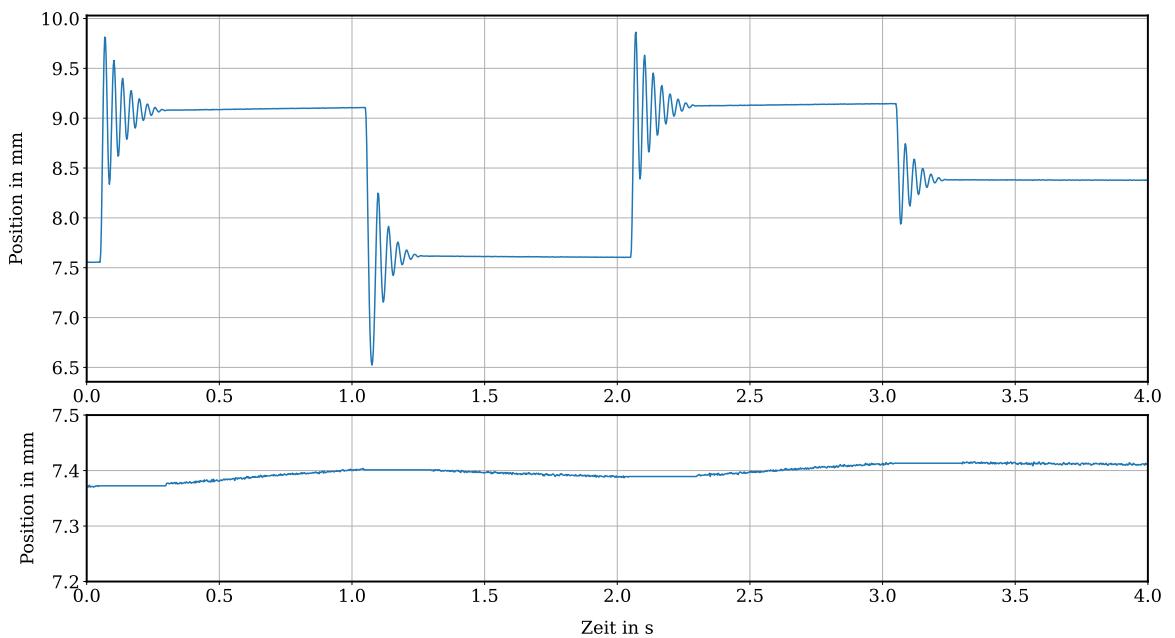


Abbildung 5.8: Gemessenes Positionssignal (oben), Für das Training bereinigte Positionssignal ohne Sprünge (unten)

Tabelle 5.2: Untersuchte Parameter des ersten Sweeps

Größe der Trainingsbatches	1	5	10	<b>50</b>	100	200	500	1000
Größe der Hidden-Layer	3	5	<b>10</b>	50	100			
Anzahl der Hidden-Layer	2	<b>3</b>	4					
Aktivierungsfunktion	<i>ReLU</i>	<b>tanh</b>						
Bias	<b>Ja</b>	Nein						

Eingangslags und mehr als 10 Ausgangslags zu verwenden. Das im nächsten Abschnitt vorgestellte Netz, das auch für die Regelung des Systems verwendet wurde, hat 14 Ausgangslags und 5 Eingangslags.

### 5.2.3 Training

Die ersten Trainingsversuche fanden mit der in Abbildung 5.9 dargestellten SISO-Struktur statt. Hier zeigte sich, dass das Modell nicht in der Lage war das Kriech-

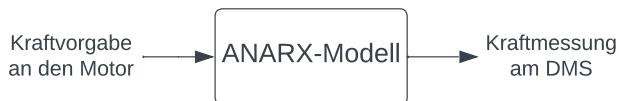


Abbildung 5.9: SISO-ANARX Modell

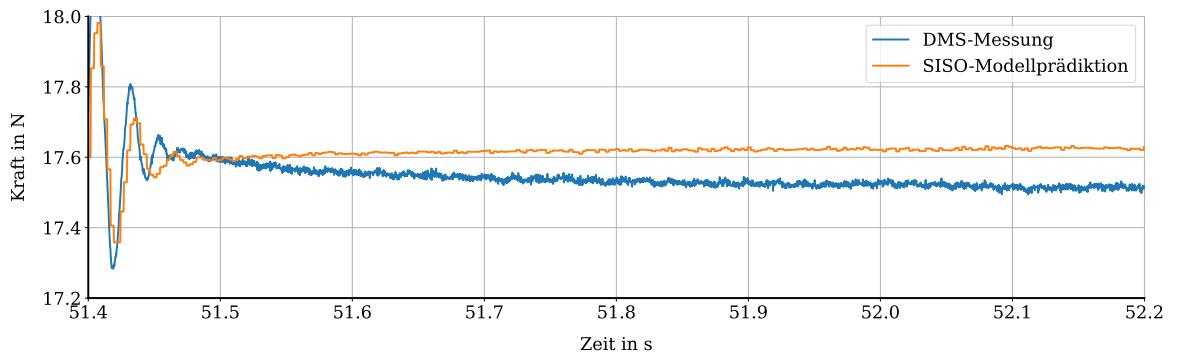


Abbildung 5.10: Kriechverhalten eines SISO-ANARX-Modell im Vergleich zum realen Messwert



Abbildung 5.11: MISO-ANARX Modell

verhalten des realen Systems nachzubilden. Die Prädiktion eines solchen Modells, ist in Abbildung 5.10 dargestellt. Es ist deutlich zu erkennen, wie das Modell nach dem Einschwingvorgang stationär bleibt, während das reale System kriecht. Um dieses Problem zu lösen wurde die Modellstruktur um einen Eingang - den Positionsmesswert des Motors - erweitert. Abbildung 5.11 zeigt das resultierende MISO-Modell. Eine Ursache für das Kriechen könnte die Längung des Seils sein. Diese kann über den Positionswert gut erfasst werden. In Abbildung 5.12 ist das Kriechverhalten eines MISO-Modells mit Positionseingang dargestellt. Man kann deutlich erkennen, dass das Kriechverhalten von diesem Modell deutlich besser dargestellt wird.

Beim Training hat es sich als gute Strategie erwiesen zunächst Open-Loop (vgl. 3.2.6) zu trainieren, bis sich die Ergebnisse auf den Validierungsdaten nicht weiter verbesserten. Anschließend wurde Closed-Loop (vgl. 3.2.6) wieder solang trainiert, bis sich die Performance auf den Validierungsdaten nicht weiter verbessert hat. Das bedeutete meist etwa 500 Episoden Open-Loop- und 200 Episoden Closed-Loop-Training. Als Loss-Funktion diente in allen Fällen die mittlere quadratische Abweichung, bzw. `torch.nn.MSELoss`<sup>2</sup>. Zusammenfassend kann man sagen, dass ANARX-Modelle das System sehr gut nachbilden können. Die endgültige Kraft am DMS sagen sowohl SISO- als auch MISO-Modelle sehr exakt voraus, wobei nur MISO-Modelle dem Kriechen folgen können. Auch das

<sup>2</sup><https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html>

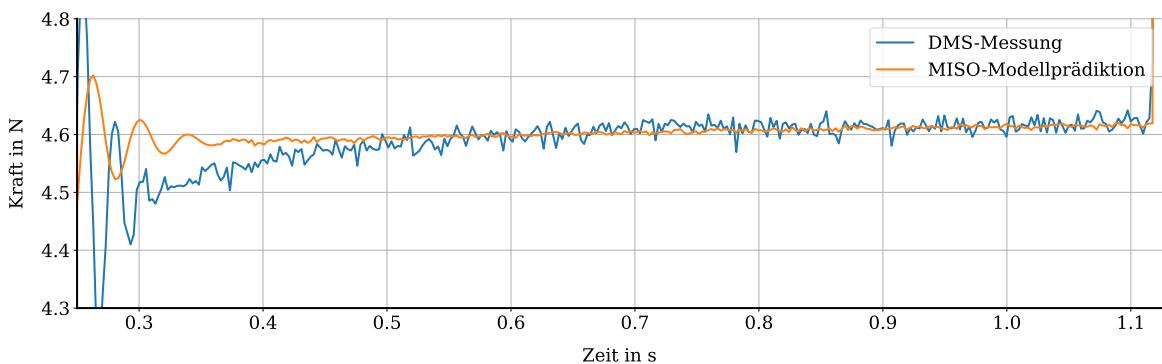


Abbildung 5.12: Kriechverhalten eines MISO-ANARX-Modell im Vergleich zum realen Messwert

Schwingen des realen Systems stellen die Modelle gut dar. Abbildung 5.14 zeigt die Parallelvoraussage eines SISO-Modells. Man kann unschwer erkennen, wie gut das System den Schwingungen folgt und auch die endgültige Kraft am DMS voraussagt.

### 5.2.4 Einsatz am realen System

Im folgenden Abschnitt wird beschrieben, wie der Regelansatz aus Abschnitt 3.2.5 auf das echte System übertragen wurde und welche Herausforderungen sich dabei ergeben haben. Zunächst ist es für eine Regelung nicht mehr möglich, das reale System als MISO-System zu modellieren, da der Positionswert des Motors zwar als Messung zur Verfügung steht, aber nicht vorgegeben werden kann. Für den Regelansatz aus Abschnitt 3.2.5 ist es allerdings nötig, dass alle Eingänge des Modells auch vorgegeben werden können. Für den Regler wird also das in Abbildung 5.9 dargestellte SISO-Modell verwendet.

Ein weiteres Problem ist, dass die Dynamik des echten Systems recht anfällig für Veränderungen ist. Das System reagiert beispielsweise abweichend, wenn das Seil anders auf die Winde gewickelt wurde oder zwischenzeitlich von der Rolle gesprungen war. Diese Eigenschaft macht es nötig, dass nicht viel Zeit zwischen der Aufnahme der Trainingsdaten und der Verwendung des Reglers vergeht. Für die Ergebnisse aus diesem Abschnitt vergingen nur einige Stunden zwischen der Datenaufnahme und Anwendung des Reglers. Um zu überprüfen, ob das Modell das reale System gut darstellt, bietet sich der in Abbildung 5.13 dargestellte Parallelbetrieb des Modells mit dem realen System an. Dabei erhält das Modell die zeitverzögerten Messwerte des realen Systems, und muss - wie im Reglerbetrieb auch - immer nur einen Schritt in die Zukunft voraussagen. In Abbildung 5.14 sind die Ergebnisse eines solchen Parallelbetriebs mit zufälligen

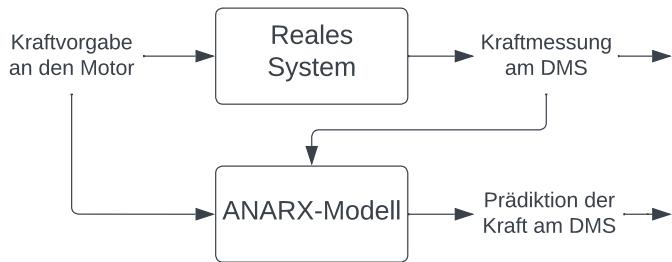


Abbildung 5.13: ANARX-Modell im Parallelbetrieb mit dem realen System als Blockschaltbild

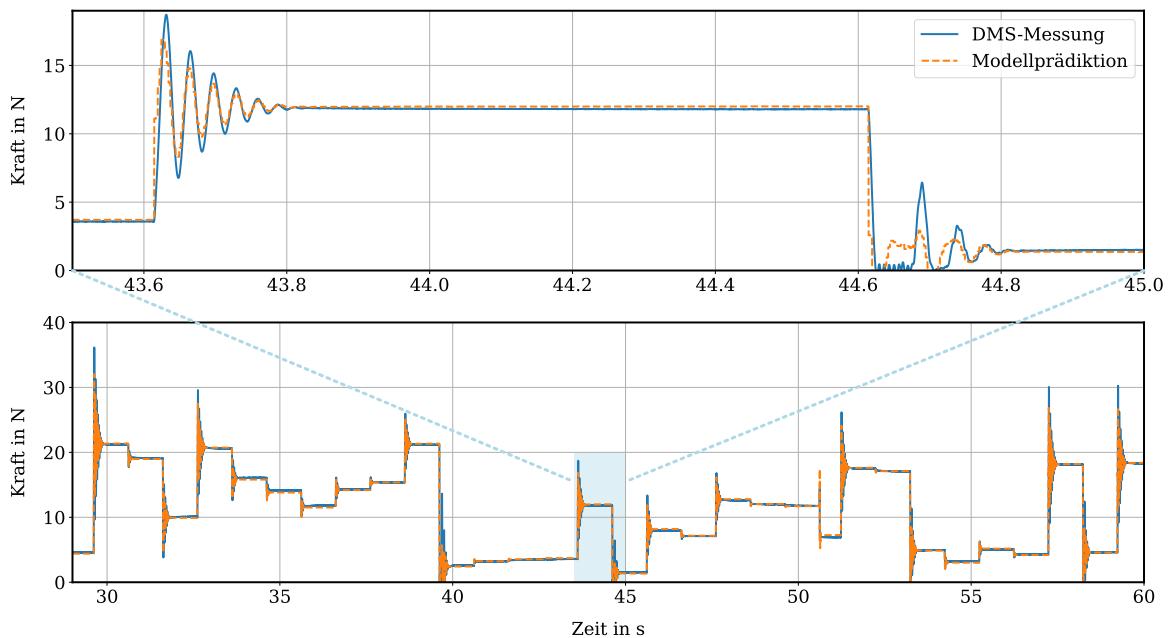


Abbildung 5.14: ANARX-Modell im Parallelbetrieb mit dem realen System: Ergebnisse

Sprüngen dargestellt. Man kann erkennen, dass das Modell der Systemtrajektorie prinzipiell sehr genau folgt. Die Schwingung nach Sprüngen wird sehr exakt modelliert. Das Kriechen des realen Systems kann das Modell nicht darstellen, da der dazu benötigte Positionswert fehlt. Rechts oben in Abbildung 5.14 ist ein Sprung zu sehen, bei dem das Schwingen dazu führt, dass keine Kraft mehr am DMS anliegt. Das Modell weicht zunächst deutlich von der Systemtrajektorie ab, erholt sich dann aber schnell genug um die letzte Schwingung sowie den Endwert exakt zu modellieren. Dieses Ergebnis ist sehr positiv, da in den Trainingsdaten nie eine solche Situation auftauchte. Das Modell zeigt sich hier also sehr extrapolationsfähig und resilient gegen lokale Fehler.

Mit einem für den Anwendungsfall sehr guten Modell kann nun die benötigte Stellgröße  $u$  bestimmt werden. Da das Modell in NN-SANARX-Struktur vorliegt muss die

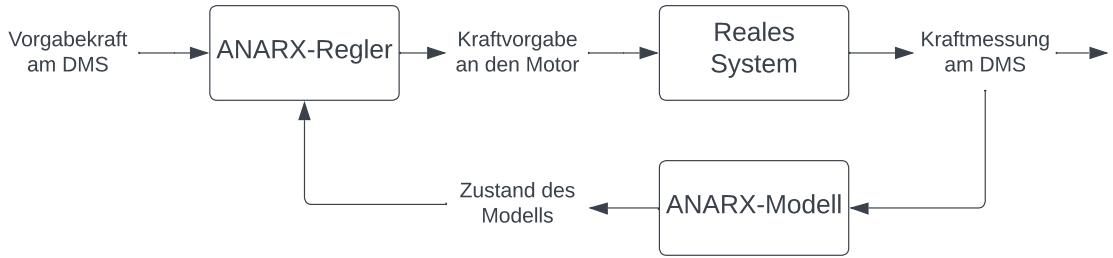


Abbildung 5.15: Blockschaldbild des Reglers auf Basis eines ANARX-Modells

Gleichung

$$\eta(t+1) = x_2(t) + N_1(y(t), \mathbf{u}(t))$$

nach (dem in diesem Falle eindimensionalen)  $u(t)$  umgestellt werden (vgl. Abschnitt 3.2.5), wobei  $\eta$  die Führungsgröße ist.  $N_1$  ist als `torch.nn.Linear`<sup>3</sup> implementiert, somit gilt

$$\eta(t+1) = x_2(t) + [u(t) \ y(t)]^T \cdot A^T + b \text{ bzw.}$$

$$\eta(t+1) = x_2(t) + u(t) \cdot a_1 + y(t) \cdot a_2 + b .$$

Nun lässt sich aus dem Modellzustand  $x_2(t)$ , dem Systemausgang  $y(t)$  sowie der Führungsgröße  $\eta(t+1)$  der im Sinne des Modells ideale Systemeingang  $u(t)$  wie folgt bestimmen:

$$u(t) = \frac{\eta(t+1) - x_2(t) - y(t) \cdot a_2 - b}{a_1} .$$

Die Implementierung und Berechnung dieses Regelgesetzes ist - wenn ein parallel zum System betriebenes ANARX-Modell vorhanden ist - trivial, da sich die Parameter  $a_1, a_2$  und  $b$  zur Laufzeit nicht mehr ändern. In Abbildung 5.15 ist das vollständige, geregelte System als Blockschaltbild dargestellt.

## ANARX-I

Beim Betrieb des beschriebenen Reglers am realen System war festzustellen, dass er keine stationäre Genauigkeit erreichen konnte. Die Ursache dafür liegt vermutlich in der Abweichung der Modellprädiktion vom tatsächlichen Systemausgang. So kann auch erklärt werden, warum dieses Verhalten in der Simulation nicht auftrat: Hier waren Modellausgang und Systemausgang identisch, da das Modell zur Simulation des Systems verwendet wurde. Eine simple Lösung des Problems stellt die Erweiterung des Systems um eine integrierende Kaskade dar. Die resultierende Struktur ist in Abbildung 5.16

<sup>3</sup><https://pytorch.org/docs/stable/generated/torch.nn.Linear.html>

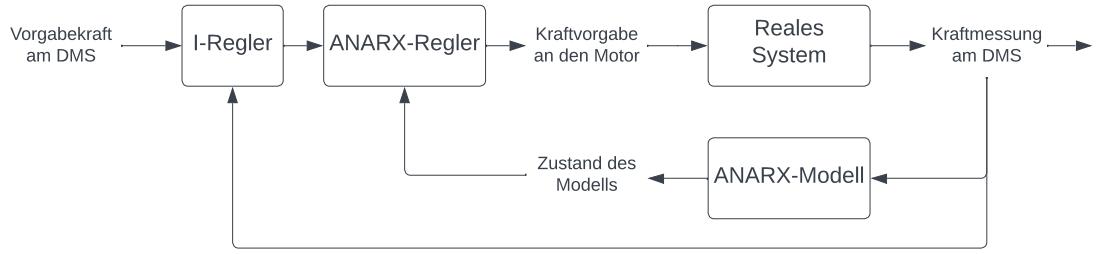


Abbildung 5.16: Blockschaltbild des Reglers auf Basis eines ANARX-Modells mit I-Regler-Kaskade

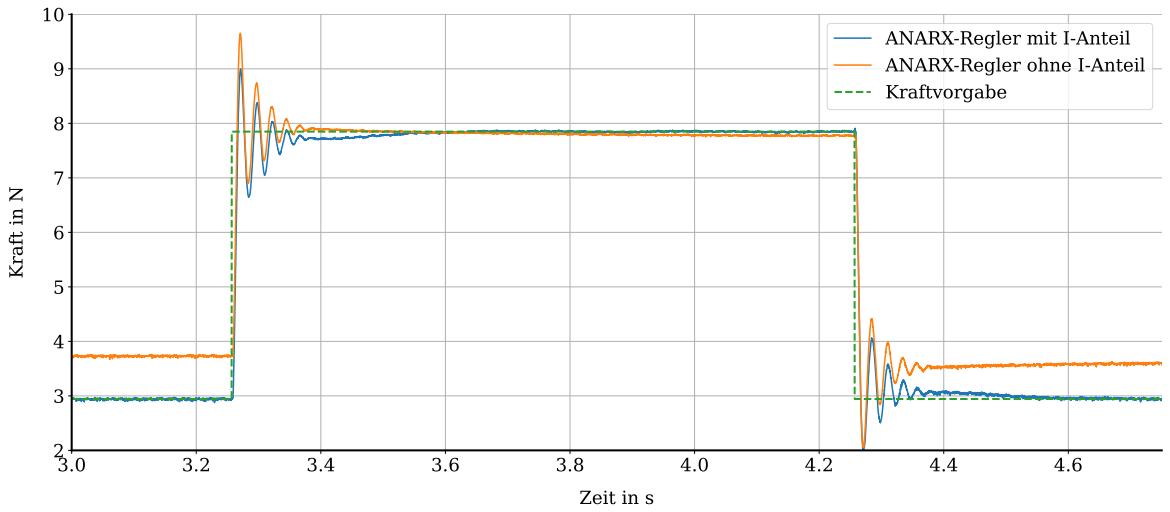


Abbildung 5.17: Sprungantwort des ANARX-Reglers mit und ohne integrierenden An teil

dargestellt.

In Abbildung 5.17 sind die Sprungantworten des geregelten Systems mit und ohne integrierende Kaskade dargestellt.

Auf eine ausführliche Analyse der Performance des vorgestellten Regelansatzes wird an dieser Stelle verzichtet. Dies wird in Abschnitt 5.3 nachgeholt.

## 5.3 Vergleich

Nachdem bereits verschiedene Ansätze vorgestellt wurden, mit denen die Kraft an der Kraftmesseinheit geregelt werden kann, sollen diese im folgenden Abschnitt nun verglichen werden. Neben dem Regler auf Basis von ANARX (Abschnitt 5.2.4) und dem RL Regler auf Basis von DDPG 5.1 wird in den Vergleich noch ein naiver Regelansatz

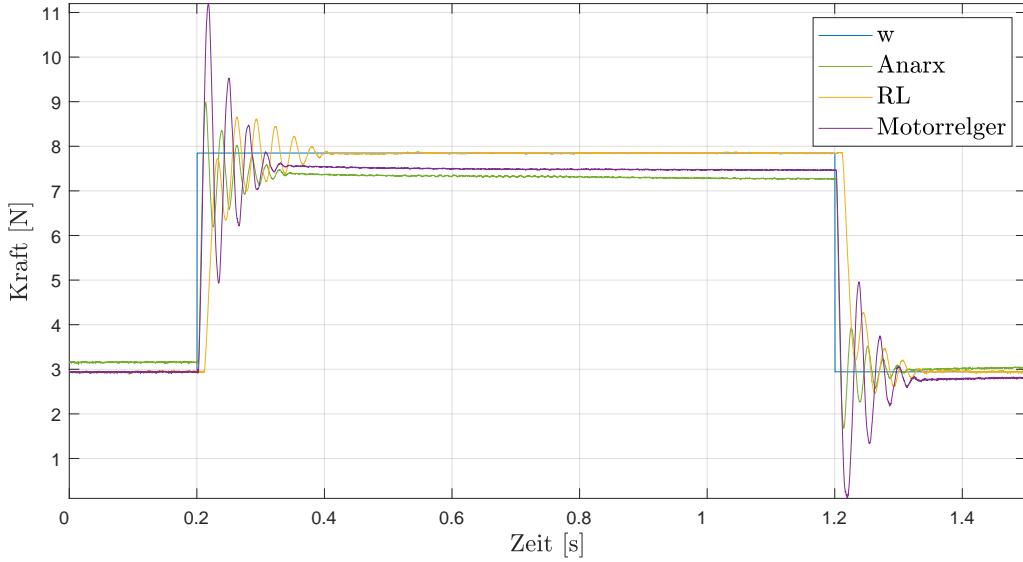


Abbildung 5.18: Vergleich der Regler an einem Sprung

aufgenommen, der letztlich der im Motor integrierten Krafteregelung entspricht. Dabei ist zu beachten, dass die Regler mit verschiedenen Abtastraten arbeiten. Der naive Regler arbeitet mit vollen 4 kHz, während der ANARX-Regler durch die Frequenz des Modells auf 400 Hz beschränkt ist. Der RL-Regler wird nur mit einer Frequenz von 100 Hz geupdatet. Alle drei Ansätze fahren zum Vergleich auf dem realen System die gleiche Trajektorie, die eine Rechtecksfunktion und einen Sinus enthält.

### 5.3.1 Naiver Ansatz

Für den naiven Ansatz wird die Trajektorie mit einer Skalierung direkt als Krafteingang an den Motor gegeben. Die Skalierung ist dabei so gewählt, dass das System im Ausgangszustand die Zielkraft am DMS erreicht. Es handelt sich also praktisch um einen einfachen Vorfaktor für den im Motor integrierten PID-Stromregler.

### 5.3.2 Sprünge

In Abbildung 5.18 ist die Antwort aller geregelten Systeme auf einen negativen und positiven Sprung der Führungsgröße zu sehen. Dabei ist  $w$  die Kraftvorgabe. Man kann sofort erkennen, dass ANARX und Motorregler den vorgegebenen Endwert nicht erreichen, d.h. sie sind nicht stationär genau. Der RL-Regler erreicht nach einer Einschwingphase den Endwert exakt. Was die Einschwingzeit an den jeweiligen Endwerten

anbetrifft liegen ANARX und Motorregler mit 150 ms etwa gleichauf, der RL-Regler benötigt mit 200 ms länger. Beim RL-Regler variiert diese Zeit allerdings je nach Sprung und Training - der dargestellte Sprung stellt für ihn eher ein „Best-Case“-Szenario dar. Der größte Unterschied zwischen den Regelansätzen findet sich in der Schwingungskompensation: Der ANARX-Regler hält bei gleicher Einstellzeit die Amplitude der Schwingung deutlich niedriger als der naive Regler. Der RL-Regler kann die Schwingung oft sehr gut kontrollieren und ein Überschwingen vollständig vermeiden. In manchen Situationen verursacht er allerdings auch ein deutliches Überschwingen (vgl. auch Abb. 5.6). Außerdem ist auch die langsamere Abtastzeit des RL-Reglers gut an der Totzeit, nach der Änderung von  $w$ , zu erkennen.

### 5.3.3 Sinus

In Abbildung 5.19 ist die Antwort aller geregelten Systeme auf eine sinusförmige Schwingung zu sehen. Dabei ist  $w$  die Kraftvorgabe. Wie schon beim Sprung, erreicht nur der RL-Regler die Endwerte tatsächlich. In diesem Beispiel kommt der naive Regler den Endwerten etwas näher als der ANARX-Regler. In den aufsteigenden Flanken haben alle Ansätze einen etwa identischen Schleppfehler. In den absteigenden Flanken sind ANARX- und Motorregler sehr genau, wohingegen der RL-Regler einen größeren Schleppfehler aufweist. Darüber hinaus weist die Trajektorie des RL-Reglers während der kompletten Schwingung eine Reihe von Artefakten auf. Diese könnten darauf zurückgeführt werden, dass die Stellgröße des RL-Reglers durch die geringere Abtastzeit nicht so glatt wie bei den anderen Ansätzen ist.

### 5.3.4 Fazit

In Tabelle 5.3 sind die Ergebnisse dieses Abschnitts nochmal zusammengefasst. Stationäre Genauigkeit erreichte in der Grundkonfiguration nur der RL-Regler, dafür stellen die andern Ansätze ihren jeweiligen Endwert etwas schneller ein. Schwingungen können sowohl ANARX- als auch RL-Regler besser kompensieren als der naive Ansatz. Einen gewissen Schleppfehler weisen alle Ansätze auf, beim RL-Ansatz ist dieser etwas stärker ausgeprägt. Ursache dafür könnte allerdings sein, dass der RL-Regler mit der niedrigsten Frequenz läuft. Für den ANARX-Regler hat sich in unseren Experimenten gezeigt, dass durch hinzufügen eines einfachen I-Reglers stationäre Genauigkeit erreicht werden kann, wobei die Einstellzeit dann etwa auf das Niveau des RL-Reglers fällt (vgl. Abbildung 5.17). Diese Erweiterung könnte dem naiven Regelansatz sicherlich auch zu stationärer Genauigkeit verhelfen. Die Performance des RL-Reglers auf der

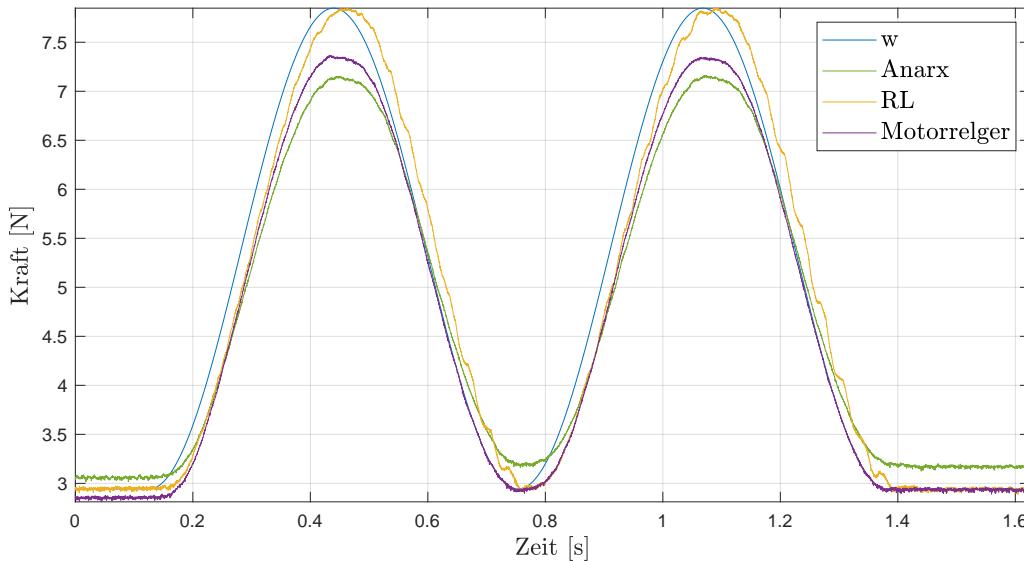


Abbildung 5.19: Vergleich der Regler an einer Schwingung

Sinusschwingung könnte mit Training an schwingenden Signalen definitiv auch noch verbessert werden. Um die eigentliche Idee der Regler nicht zu sehr zu verzerren wurde in diesem Vergleich jedoch auf diese Erweiterungen verzichtet.

Ein weiterer wichtiger Vergleich zwischen den Methoden ist der Aufwand der zur Implementierung und Anwendung nötig ist. In dieser Kategorie gewinnt der naive Regelansatz offensichtlich mit Abstand. Für die Anwendung des ANARX-Reglers ist zunächst das Trainieren eines neuronalen Netzes notwendig, was anschließend zur Laufzeit ausgewertet werden muss. Die Auswertung stellt auf Zielsystemen mit guter ONNX-Unterstützung<sup>4</sup> kein Problem dar. Wird ONNX nicht oder - wie in unserem Falle<sup>5</sup> - nur eingeschränkt unterstützt, ist die Überführung der Netzauswertung auf das Zielsystem aufwändig. Kann das Netz zur Laufzeit ausgewertet werden, ist die Implementierung des Reglers hingegen trivial (vgl. 5.2.4).

Den RL-Regler kann man in der Anwendung definitiv als den Aufwändigsten der drei Ansätze bezeichnen. Je nach System verursacht das benötigte Training am realen System nämlich erheblichen Aufwand oder sogar Kosten. Es muss dabei sichergestellt werden, dass, wie in Kapitel 5.1.6 beschrieben, der Lernalgorithmus das System während der Exploration nicht beschädigt.

<sup>4</sup><https://onnx.ai/>

<sup>5</sup>Aus einer Email vom Beckhoff-Support zur Neural Network Inference Engine: „Hallo Herr Höpfner, es gibt momentan keine Liste der Unterstützten ONNX Operationen. Der Concat Befehl wird momentan nicht unterstützt.“

Tabelle 5.3: Vergleich der untersuchten Regelansätze

	<b>RL-Regler</b>	<b>ANARX-Regler</b>	<b>Naiver Regler</b>
<b>Einschwingzeit</b>	200 ms	150 ms	150 ms
<b>Stationäre Genauigkeit</b>	ja	nein	nein
<b>Schwingungskompensation</b>	++	++	+
<b>Schleppfehler</b>	--	-	-
<b>Aufwand</b>	hoch	mittel/hoch	gering

Ob sich der hohe Aufwand zur Implementierung lohnt, hängt letztlich vom Anwendungsfall ab. Dieser Vergleich sollte dafür eine solide Entscheidungsgrundlage bieten.

## 6 Ausblick

Das Erlernen eines RL-Reglers mittels DDPG war erfolgreich und die Ergebnisse gut. Zur Fortführung wäre es denkbar andere Lernalgorithmen zu verwenden, wie z.B. *Twin Delayed DDPG (TD3)*. Dieser Algorithmus ist eine Weiterentwicklung von DDPG und soll dem teils instabilen Lernvorgang entgegenwirken. Während des Lernvorgangs wurde des Öfteren festgestellt, dass ein Regler der bereits gut funktionierte plötzlich wieder stark zu schwingen begann. Eventuell könnte eine Veränderung des Algorithmus hier eine Verbesserung bewirken. Des Weiteren ist denkbar, das Training mit einer geringen maximalen Stellgröße zu beginnen und diese Grenze während des Trainings anzuheben. Hierdurch könnte die Gefahr, dass während der Explorationen der mechanische Aufbau beschädigt wird verringert werden. Die Idee ist, dass der RL-Regler zuerst lernen soll, wie das System gedämpft werden kann. Erst wenn dies erfolgt ist, wird die maximale Stellgröße schrittweise erhöht, bis die gewünschte Größe erreicht ist. Eine andere Idee die Lerngeschwindigkeit und Zuverlässigkeit zu erhöhen, ist es das Netz erst in Simulation vorzutrainieren. Wenn dies funktioniert kann die Reglerauslegung auch stark davon profitieren, dass die in der Simulation wesentlich schneller die Trainingsdurchgänge aufgenommen werden können. Der Nachteil an dieser Vorgehensweise ist jedoch, dass eine gute Simulation benötigt wird. Jedoch ist die Idee hinter RL, dass kein genaues Modell entworfen werden muss, was einer guten Simulation widerspricht. Hier muss also abgewägt werden, wie viel Zeit durch eine Simulation für das Training eingespart werden kann.

Eine Verbesserung ist außerdem dadurch denkbar, dass die Zykluszeit des RL-Reglers verkürzt wird. Eine Steigerung auf eine Frequenz von 200 Hz oder sogar 500 Hz ist eventuell durch ein Hardwareupgrade des Entwickler-PCs möglich. Für schnellere Abtastzeiten ist jedoch die Auswertung des Netzes auf der SPS schon während des Trainingsvorgangs nötig. Beckhoff bietet mit der *Neural Network Inference Engine*<sup>1</sup> die Möglichkeit das genutzte Netz zur Laufzeit zu verändern. Somit ist es nach jeder Episode möglich das Actor-Netzwerk weiterzutrainieren und das geupdate Netzwerk auf die

---

<sup>1</sup><https://www.beckhoff.com/de-de/produkte/automation/twincat/tfxxxx-twincat-3-functions/tf3xxx-tc3-measurement/tf3810.html>

SPS zu überspielen. Durch diese Vorgehensweise kann das Netz einerseits schneller ausgewertet werden und andererseits entfällt die Verzögerung durch die Übertragung. Als Weiterentwicklung des ganzen Aufbaus ist denkbar, dass ein zweiter Kraftsensor an dem Punkt, an dem das Seil mit dem Rahmen verspannt wird, am Seil angebracht wird. Das Ziel der Regelung ist es dann, die Kraft am Ende des Seils mithilfe des Messsignals des DMS zu regeln. Hierdurch müsste der Regler in seinem Netz das ganze System samt Umlenkrollen und möglicher Seillängung durch den Zug am Seil lernen. Dieser Schritt ist nötig, damit der Kraftregler tatsächlich innerhalb des Seilroboters verwendet werden kann.

Auch für die Systemidentifikation mit neuronalen Netzen wäre ein zweiter Kraftsensor ein interessanter Ansatz. So könnte mithilfe eines A(NARX)-Netzes aus der Kraft am DMS die tatsächlich am Endeffektor wirkende Kraft bestimmt werden. Insgesamt eignete sich die NN-(A)NARX Methode, bei entsprechender Wahl der Modellparameter, ausgesprochen gut, um das betrachtete System zu modellieren.

Der Ansatz ein nicht-deskriptives Modell so zu wählen, dass die gewünschte Größe (hier der optimale Systemeingang  $\mathbf{u}$ ) leicht bestimmt werden kann hat sich verblüffend effektiv gezeigt. Eine interessante Erweiterung dazu wäre die Berechnung eines optimalen Eingangs  $\mathbf{u}$  im Sinne einer Mehrschritt-Vorraussage. Die Performance des Reglers könnte dadurch vermutlich weiter verbessert werden. Denkbar wäre zum Beispiel der Einsatz eines (A)NARX-Modells innerhalb eines *Model-Predictive-Control*-Ansatzes (MPC)[33]. Neben den bereits genannten gibt es eine lange Reihe von „klassischen“ Regelansätzen, die auf Modellen in Zustandsraumform basieren. Ein interessanter Anknüpfungspunkt wäre die Überprüfung, welche dieser Ansätze auch mit Modellen aus der ANARX-Klasse funktionieren könnten. So könnte sich eine spannende Schnittstelle zwischen datenbasierten Methoden und klassischer Regelungstechnik ergeben.

# Literatur

- [1] Adel Ameri, Amir Molaei und Mohammad A. Khosravi. „Nonlinear Observer-Based Tension Distribution for Cable-Driven Parallel Robots“. In: *Cable-Driven Parallel Robots*. Hrsg. von Marc Gouttefarde, Tobias Bruckmann und Andreas Pott. Bd. 104. Mechanisms and Machine Science. Cham: Springer International Publishing, 2021, S. 105–116. ISBN: 978-3-030-75788-5. DOI: 10.1007/978-3-030-75789-2{\textunderscore}9 (S. 3).
- [2] Vahid Azimi, Mohammad Bagher Menhaj und Ahmad Fakharian. „Adaptive control of a wind turbine based on neural networks“. In: *2013 13th Iranian Conference on Fuzzy Systems (IFSC)*. 2013, S. 1–6. DOI: 10.1109/IFSC.2013.6675650 (S. 18).
- [3] Weiwei Bai et al. „Adaptive Reinforcement Learning Neural Network Control for Uncertain Nonlinear System With Input Saturation“. In: *IEEE transactions on cybernetics* 50.8 (2020), S. 3433–3443. DOI: 10.1109/TCYB.2019.2921057 (S. 10).
- [4] Jonas Bieber et al. „Motor Current Based Force Control of Simple Cable-Driven Parallel Robots“. In: *Cable-Driven Parallel Robots*. Hrsg. von Marc Gouttefarde, Tobias Bruckmann und Andreas Pott. Bd. 104. Mechanisms and Machine Science. Cham: Springer International Publishing, 2021, S. 271–283. ISBN: 978-3-030-75788-5. DOI: 10.1007/978-3-030-75789-2{\textunderscore}22 (S. 3).
- [5] S. A. Billings. *Nonlinear system identification: NARMAX methods in the time, frequency, and spatio-temporal domains*. Chichester, West Sussex, United Kingdom: John Wiley & Sons, Inc, 2013. ISBN: 9781118535530 9781118535547 9781118535554 (S. 18).
- [6] P. H. Borgstrom et al. „Rapid Computation of Optimally Safe Tension Distributions for Parallel Cable-Driven Robots“. In: *IEEE Transactions on Robotics* 25.6 (2009), S. 1271–1281. ISSN: 1552-3098. DOI: 10.1109/TR.2009.2032957 (S. 3).

- [7] Ishan Chawla et al. „Neural Network-Based Inverse Kineto-Static Analysis of Cable-Driven Parallel Robot Considering Cable Mass and Elasticity“. In: *Cable-Driven Parallel Robots*. Hrsg. von Marc Gouttefarde, Tobias Bruckmann und Andreas Pott. Bd. 104. Mechanisms and Machine Science. Cham: Springer International Publishing, 2021, S. 50–62. ISBN: 978-3-030-75788-5. DOI: 10.1007/978-3-030-75789-2{\textunderscore}5 (S. 4).
- [8] Yan Duan et al. *Benchmarking Deep Reinforcement Learning for Continuous Control* (S. 10).
- [9] A. Dzielinski. „Neural networks based NARX models in nonlinear adaptive control“. In: *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*. Bd. 3. 1999, 2098–2103 vol.3. DOI: 10.1109/IJCNN.1999.832710 (S. 18).
- [10] Shixiang Gu et al. „Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates“. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 29.05.2017 - 03.06.2017, S. 3389–3396. ISBN: 978-1-5090-4633-1. DOI: 10.1109/ICRA.2017.7989385 (S. 10).
- [11] Jiang Guo. „Backpropagation through time“. In: *Unpubl. ms., Harbin Institute of Technology* 40 (2013), S. 1–6 (S. 26).
- [12] Mohamed Handaoui et al. *ReLeaSER: A Reinforcement Learning Strategy for Optimizing Utilization Of Ephemeral Cloud Resources* (S. 13).
- [13] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980 (S. 24).
- [14] William Koch et al. *Reinforcement Learning for UAV Attitude Control* (S. 10).
- [15] U. Kotta und N. Sadegh. „Two Approaches for State Space Realization of NARMA Models: Bridging the Gap“. In: *Mathematical and Computer Modelling of Dynamical Systems* 8.1 (März 2002), S. 21–32. ISSN: 1387-3954. DOI: 10.1076/mcmd.8.1.21.8340 (S. 20).
- [16] W. Kraus et al. „Hybrid Position-Force Control of a Cable-Driven Parallel Robot with Experimental Evaluation“. In: *Mechanical Sciences* 6.2 (2015), S. 119–125. DOI: 10.5194/ms-6-119-2015 (S. 3).
- [17] Werner Kraus. „Force control of cable-driven parallel robots“. Dissertation (S. 1, 3).

- [18] Werner Kraus, Michael Kessler und Andreas Pott. „Pulley friction compensation for winch-integrated cable force measurement and verification on a cable-driven parallel robot“. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 26.05.2015 - 30.05.2015, S. 1627–1632. ISBN: 978-1-4799-6923-4. DOI: [10.1109/ICRA.2015.7139406](https://doi.org/10.1109/ICRA.2015.7139406) (S. 3).
- [19] Werner Kraus et al. „System identification and cable force control for a cable-driven parallel robot with industrial servo drives“. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 31.05.2014 - 07.06.2014, S. 5921–5926. ISBN: 978-1-4799-3685-4. DOI: [10.1109/ICRA.2014.6907731](https://doi.org/10.1109/ICRA.2014.6907731) (S. 3).
- [20] Johann Lamaury und Marc Gouttefarde. „Control of a large redundantly actuated cable-suspended parallel robot“. In: *2013 IEEE International Conference on Robotics and Automation*. IEEE, 6.05.2013 - 10.05.2013, S. 4659–4664. ISBN: 978-1-4673-5643-5. DOI: [10.1109/ICRA.2013.6631240](https://doi.org/10.1109/ICRA.2013.6631240) (S. 3).
- [21] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning* (S. 13).
- [22] A. Rupam Mahmood et al. *Benchmarking Reinforcement Learning Algorithms on Real-World Robots* (S. 10).
- [23] Martin J.-D. Otis et al. „Cable Tension Control And Analysis Of Reel Transparency For 6-Dof Haptic Foot Platform On A Cable-Driven Locomotion Interface“. In: (2009). DOI: [10.5281/zenodo.1079462](https://doi.org/10.5281/zenodo.1079462) (S. 3).
- [24] Utkarsh A. Mishra und Stéphane Caro. „Unsupervised Neural Network Based Forward Kinematics for Cable-Driven Parallel Robots with Elastic Cables“. In: *Cable-Driven Parallel Robots*. Hrsg. von Marc Gouttefarde, Tobias Bruckmann und Andreas Pott. Bd. 104. Mechanisms and Machine Science. Cham: Springer International Publishing, 2021, S. 63–76. ISBN: 978-3-030-75788-5. DOI: [10.1007/978-3-030-75789-2\\_6](https://doi.org/10.1007/978-3-030-75789-2_6) (S. 4).
- [25] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning* (S. 10).
- [26] Katharina Müller, Christopher Reichert und Tobias Bruckmann. „Analysis of a Real-Time Capable Cable Force Computation Method“. In: *Cable-Driven Parallel Robots*. Hrsg. von Andreas Pott und Tobias Bruckmann. Bd. 32. Mechanisms and Machine Science. Cham: Springer International Publishing, 2015, S. 227–238. ISBN: 978-3-319-09488-5. DOI: [10.1007/978-3-319-09489-2\\_16](https://doi.org/10.1007/978-3-319-09489-2_16) (S. 3).
- [27] OpenAI. *Deep Deterministic Policy Gradient* (S. 13).

- [28] Martijn van Otterlo und Marco Wiering. „Reinforcement Learning and Markov Decision Processes“. In: *Reinforcement Learning: State-of-the-Art*. Hrsg. von Marco Wiering und Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3\_1 (S. 11).
- [29] Adam Paszke et al. „Automatic differentiation in pytorch“. In: (2017) (S. 25, 27).
- [30] E. Petlenkov. „NN-ANARX structure based dynamic output feedback linearization for control of nonlinear MIMO systems“. In: *2007 Mediterranean Conference on Control & Automation*. 2007, S. 1–6. DOI: 10.1109/MED.2007.4433965 (S. 20, 22–24).
- [31] Eduard Petlenkov und Juri Belikov. „NN-ANARX Structure for Control of Nonlinear SISO and MIMO Systems: Neural Networks Based Approach“. In: *CCC2007*. Bd. 4. 2007, S. 138–145 (S. 22).
- [32] Eduard Petlenkov, Sven Nomm und Ulle Kotta. „Neural Networks Based ANARX Structure for Identification and Model Based Control“. In: *2006 9th International Conference on Control, Automation, Robotics and Vision*. 2006, S. 1–5. DOI: 10.1109/ICARCV.2006.345417 (S. 20).
- [33] J.B. Rawlings. „Tutorial overview of model predictive control“. In: *IEEE Control Systems Magazine* 20.3 (2000), S. 38–52. DOI: 10.1109/37.845037 (S. 62).
- [34] G. E. Uhlenbeck und L. S. Ornstein. „On the Theory of the Brownian Motion“. In: *Physical Review* 36.5 (1930), S. 823–841. ISSN: 0031-899X. DOI: 10.1103/PhysRev.36.823 (S. 46).
- [35] K. Vassiljeva, E. Petlenkov und J. Belikov. „State-space control of nonlinear systems identified by ANARX and Neural Network based SANARX models“. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 2010, S. 1–8. DOI: 10.1109/IJCNN.2010.5596581 (S. 22).
- [36] Ning Wang, Ying Gao und Xuefeng Zhang. „Data-Driven Performance-Prescribed Reinforcement Learning Control of an Unmanned Surface Vehicle“. In: *IEEE transactions on neural networks and learning systems* 32.12 (2021), S. 5456–5467. DOI: 10.1109/TNNLS.2021.3056444 (S. 10).
- [37] Guoxing Wen et al. „Optimized Adaptive Nonlinear Tracking Control Using Actor-Critic Reinforcement Learning Strategy“. In: *IEEE Transactions on Industrial Informatics* 15.9 (2019), S. 4969–4977. ISSN: 1551-3203. DOI: 10.1109/TII.2019.2894282 (S. 10).

- [38] David Winter. „Mehrgrößenregelung des parallelen Seilroboters MoCaRo“. 2019 (S. 6).

# **Erklärung**

Die vorliegende Arbeit habe ich selbstständig ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer oder anderer Prüfungen noch nicht vorgelegt worden.

Augsburg, den 25. 07. 2022

Julius Brandl und Valentin Höpfner