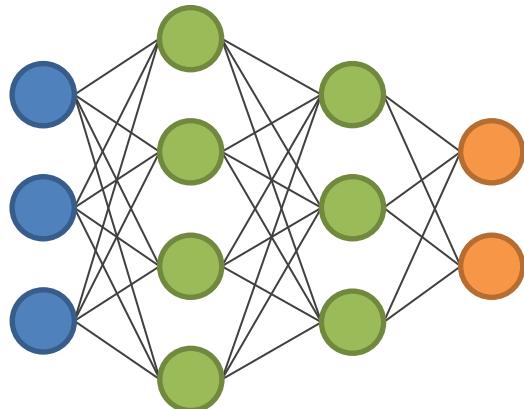
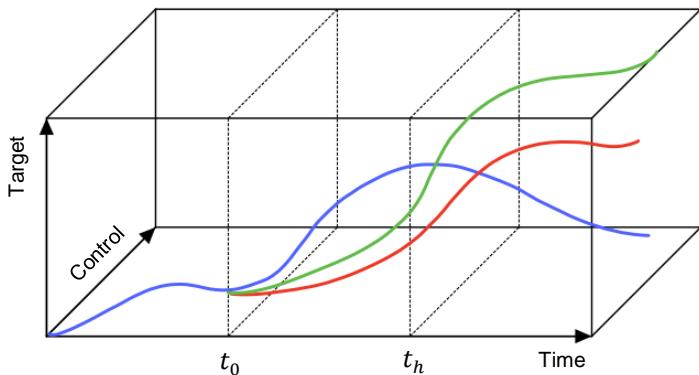




# IMPLEMENTIERUNG EINER ADAPTIVEN AUTONOMEN PROZESSREGELUNG MITTELS PRÄDIKTIVER KI-MODELLIERUNG



ANGEFERTIGT VON:  
JONAS BRINKMANN, 5005717  
BRAUNSCHWEIG, NOVEMBER 2022  
BETREUER: M. SC. CHRISTOPH THON  
ERSTPRÜFER: PROF. DR.-ING. CARSTEN SCHILDE  
ZWEITPRÜFER: PROF. DR.-ING. ARNO KWADÉ

NUMERISCH

BACHELORARBEIT

**Error! Use the Home tab to apply Überschrift 1 to the text that you want to appear here.**

II

**Eidesstattliche Erklärung**

Hiermit erkläre ich, Jonas Brinkmann, geboren am 10.03.2000, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und nur die angegebenen Hilfsmittel und Quellen verwendet habe. Alle wörtlichen oder sinngemäßigen Zitate, die veröffentlichten oder nicht veröffentlichten Schriften entnommen sind, sind als solche gekennzeichnet.

Braunschweig,

---

Jonas Brinkmann

## Inhaltsverzeichnis

Inhaltsverzeichnis .....	IV
Abkürzungsverzeichnis.....	VII
Symbolverzeichnis.....	VIII
Griechische Symbole .....	VIII
Lateinische Symbole .....	VIII
1 Einleitung und Motivation .....	11
2 Theoretische Grundlagen .....	12
2.1 Prozessregelung.....	12
2.1.1 Prozess .....	12
2.1.2 Regelkreis .....	13
2.1.3 Übertragungsverhalten .....	14
2.1.4 Übertragungsglieder .....	16
2.2 Künstliche Intelligenz.....	20
2.2.1 Maschinelles Lernen.....	21
2.2.2 Lernmethoden.....	22
2.2.3 Künstliche neuronale Netze .....	25
2.3 Fehlerberechnung bei der Vorhersage .....	30
2.3.1 Berechnung bei der Klassifikation.....	30
2.3.2 Berechnung bei der Regression.....	32
2.4 Prädiktive Modellierung über ein gleitendes Zeitfenster .....	34

---

2.4.1	Methoden für die Vorhersage über mehrere Zeitschritte.....	36
2.5	Model Predictive Control (MPC) .....	37
2.5.1	Kostenfunktion und Nebenbedingungen .....	38
2.5.2	Optimale Regelstrategie.....	40
3	Implementierung.....	42
3.1	Umformung einer Zeitreihe in überwachtes Lernen.....	44
3.2	Regel-KI .....	48
3.3	Berechnung der optimalen Regelstrategie .....	51
3.4	Virtuelles System.....	52
3.5	Datenauswertung und Visualisierung .....	55
4	Ergebnisse und Diskussion .....	57
4.1	Problem mit der ReLU Aktivierungsfunktion .....	57
4.2	Regelung eines statischen Systems .....	60
4.2.1	Einfluss der Fenstergröße auf den Regelungserfolg.....	60
4.2.2	Einfluss der Prozessparameter auf den Regelungserfolg .....	69
4.2.3	Einfluss von Störungen auf den Regelungserfolg .....	70
4.2.4	Auswirkung der Nebenbedingung $\Delta u_{\max}$ auf den Regelungserfolg .....	76
4.2.5	Einordnung der Einflussgrößen auf den Regelungserfolg.....	77
4.3	Regelung eines dynamischen Systems .....	79
4.3.1	Änderung der Führungsgröße.....	79
4.3.2	Dynamische Zustandsgrößen $k_1$ und $k_2$ .....	81

---

4.3.3 Änderung der Führungsgröße und dynamische Zustandsgrößen.....	85
4.4 Übertragung auf einen realen Prozess .....	90
5 Zusammenfassung und Ausblick.....	92
6 Literaturverzeichnis .....	94
7 Abbildungsverzeichnis.....	98
8 Tabellenverzeichnis.....	101
9 Anhang .....	103
9.1 Python Skript aus “main.py”.....	103
9.2 Python Skript der Regel-KI aus „control_ai.py“ .....	105
9.3 Python Skript aus “utils.py” .....	109
9.4 Python Skript der Simulation aus “simulation.py“ .....	111
9.5 Python Skript für die Visualisierung der Ergebnisse aus „export.py“ .....	114

## Abkürzungsverzeichnis

Abkürzung	Erläuterung
AI	Artificial Intelligence
ML	Machine Learning / Maschinelles Lernen
DL	Deep Learning
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
FNN	Feedforward Neural Network
HMM	Hidden Markov Model
HHMM	Hierarchical Hidden Markov Model
LSTM	Long Short-Term Memory
KI	Künstliche Intelligenz
KNN	Künstliches Neuronales Netz
SVM	Support Vector Machines
ReLU	Rectified Linear Unit
RP	Richtig Positiv
FP	Falsch Positiv
FN	Falsch Negativ
RN	Richtig Negativ
MSE	Mean Square Error
RMSE	Root Mean Square Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MdAPE	Median Absolute Percentage Error
MPC	Model Predictive Control
HyREN	Hybrid Regression Evolutionary Network

## Symbolverzeichnis

### Griechische Symbole

Abkürzung	Einheit	Erläuterung
$\sigma(t)$	-	Sprungfunktion
$\mu_0$	-	Sprunghöhe
$\delta(t)$	-	Dirac-Impuls
$\vartheta$	-	Gewicht für vorhergesagten Fehler
$\alpha$	-	Basis der exponentiellen Gewichtsfunktion
$\lambda$	-	Gewicht für Steuerinkremente
$\varepsilon$	%	Wahrscheinlichkeit für eine zufällige Handlung
$\varepsilon_{dec}$	-	Multiplikator zum Reduzieren von $\varepsilon$
$\sigma_r$	-	Standardabweichung des zufälligen Rauschens

### Lateinische Symbole

Abkürzung	Einheit	Erläuterung
$y(t)$	-	Regelgröße
$e(t)$	-	Regelabweichung
$z(t)$	-	Störgröße
$u(t)$	-	Stellgröße
$w(t)$	-	Führungsgröße
$k_s$	-	Statische Verstärkung eines Systems
$T$	s	Zeitkonstante
$d$	-	Dämpfungsfaktor
$x_{k,l}$	-	Eingabewerte eines Neurons
$y_{k,l}$	-	Ausgabewert eines Neurons
$b_{k,l}$	-	Bias eines Neurons
$w_{k,n,l}$	-	Gewicht eines Eingabewertes des Neurons

$l$	-	Schicht eines KNN
$f_l$	-	Aktivierungsfunktion der Schicht $l$ eines KNN
$e_t$	-	Absolute Abweichung einer Vorhersage
$y_t$	-	Tatsächlicher Wert zum Zeitpunkt $t$
$\hat{y}_t$	-	Vorhergesagter Wert für den Zeitpunkt $t$
$h$	-	Prognosehorizont
$p_t$	%	Prozentualer Fehler einer Vorhersage zum Zeitpunkt $t$
$n$	-	Anzahl vergangener Zeitpunkte, die für die Vorhersage genutzt werden
$m$	-	Regelhorizont
$\hat{e}(t)$	-	Vorhergesagte Regelabweichung für Zeitpunkt $t$
$J(h, m)$	-	Gesamtkosten einer Regelstrategie
$u_{min}$	-	Kleinstmöglicher Wert der Stellgröße
$u_{max}$	-	Größtmöglicher Wert der Stellgröße
$\Delta u_{min}$	-	Minimale Änderungsrate der Stellgröße
$\Delta u_{max}$	-	Maximale Änderungsrate der Stellgröße
$q$	-	Anzahl Zeitschritte notwendig für Umwandlung in Zeitfenster
$t_{act}$	ms	Regelintervall
$t_{train}$	ms	Trainingsintervall
$t_{window}$	ms	Größe des Zeitfensters
$Q_{train}$	-	Anzahl an neuen Daten pro Trainingsiteration
$U$	-	Menge aller möglichen diskreten Werte für die Stellgröße $u$
$M$	-	Matrix die alle möglichen Regelstrategien enthält
$y_s$	-	Wert bei dem die Steigung von $y$ beim virtuellen System minimal ist
$k_1$	-	Interne Zustandsgröße des virtuellen Systems
$k_2$	-	Interne Zustandsgröße des virtuellen Systems

**Error! Use the Home tab to apply Überschrift 1 to the text that you want to appear here.**

X

$z_r$	-	Zufälliges Rauschen des Systems
$z_i$	-	Größe eines diskreten Störimpulses
$z_d$	-	Dynamisches Störverhalten des Systems

## 1 Einleitung und Motivation

In der heutigen industriellen Produktion ist es von entscheidender Bedeutung, Prozesse effizient und zuverlässig zu steuern, um eine hohe Produktivität und Qualität zu gewährleisten. Die Prozessregelung spielt dabei eine wichtige Rolle, da sie dazu beiträgt, die Stabilität des Prozesses zu gewährleisten und den Einfluss von Störungen zu minimieren [1]. Insbesondere in der Verfahrenstechnik gibt es viele komplexe Prozesse, die sich nur schwer regeln lassen. Manuell implementierte, statische Regelungskonzepte sind oft kostenintensiv und unflexibel bei systematischen Änderungen. Ein ideales Regelungssystem sollte in der Lage sein, sich selbstständig an die Anforderungen des Prozesses anzupassen, den Prozess ohne Vorkenntnisse der Systemeigenschaften möglichst optimal zu regeln und ohne großen Aufwand auf verschiedene Systeme übertragbar zu sein. Die Verfügbarkeit höherer Rechenleistung und der Fortschritt im Bereich des maschinellen Lernens haben zu einem stark gestiegenen Interesse an der Automatisierung des Reglerentwurfs und der Adaption auf Basis historischer Prozessdaten geführt [2].

In dieser Arbeit wird eine Regel-KI entwickelt, die das Regelungsverfahren Model Predictive Control (MPC) verwendet, um in Echtzeit die Auswirkungen verschiedener Regelungsstrategien auf einen Prozess über einen bestimmten Zeithorizont vorherzusagen und die optimale Strategie zu identifizieren [3]. Als Modell wird ein künstliches neuronales Netz verwendet, das durch kontinuierliches Training mit einem gleitenden Zeitfenster in der Lage ist, die Vorhersagegenauigkeit und damit den Regelungserfolg über die Zeit zu verbessern und sich an dynamische Änderungen der Systemeigenschaften anzupassen. Abschließend wird die implementierte Regel-KI an einem virtuellen Proxy-System in verschiedenen Szenarien getestet und diskutiert, ob die Anwendung der Regel-KI an einem realen Prozess möglich ist.

## 2 Theoretische Grundlagen

### 2.1 Prozessregelung

Die Aufgabe der Prozessregelung ist es, die Stabilität eines dynamischen Prozesses zu gewährleisten und den Einfluss von Störungen zu minimieren [1]. Dies wird erreicht, indem der Prozess von außen so beeinflusst wird, dass er nach einer gewünschten Vorgabe abläuft [4].

#### 2.1.1 Prozess

Ein Prozess im verfahrenstechnischen Kontext ist ein dynamisches System, das sich durch bestimmte Eingangs- und Störgrößen verändert und durch interne Zustandsgrößen wie Temperatur, Druck oder Konzentration beschrieben wird. Die internen Zustandsgrößen zeigen die zeitliche Veränderung des Systems auf. Die Reaktion des Systems auf die Eingangs- und Störgrößen wird durch die Ausgangsgrößen bestimmt. Die Wirkung einer Eingangsgröße auf eine bestimmte Ausgangsgröße ist in der Regel zeitverzögert und kann nichtlinear sein. Dies muss beim Entwurf eines Reglers berücksichtigt werden. Ein dynamisches System wird auch als Regelstrecke bezeichnet. Die Darstellung eines Prozesses im Blockschaltbild ist in Abbildung 1 dargestellt. [1, 4]

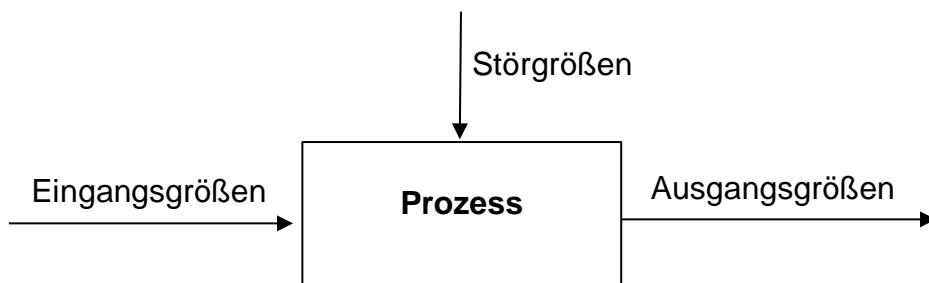


Abbildung 1: Blockdiagramm eines Prozesses

Ein Prozess kann zum Beispiel eine Wasserheizung sein, bei der die internen Zustandsgrößen die Temperatur und der Durchfluss des Wassers sind. Die Eingangsgrößen sind die Stellung des Temperaturreglers und die Umgebungstemperatur, die die Durchflussmenge und damit die Temperatur der

Heizung beeinflussen. Die Temperaturänderung beim Verstellen des Temperaturreglers ist dabei zeitverzögert. Eine Störgröße könnte die Schwankung der Raumtemperatur sein, zum Beispiel durch das Öffnen eines Fensters. Eine Ausgangsgröße könnte in diesem Fall die gemessene Temperatur der Heizung sein. [4]

### 2.1.2 Regelkreis

Das Ziel eines Regelkreises ist es, eine Regelgröße  $y(t)$  eines Prozesses oder einer Regelstrecke, möglichst genau an eine vorgegebene Führungsgröße  $w(t)$  anzunähern. Dazu muss die Regelgröße entweder direkt gemessen oder aus anderen Messgrößen berechnet werden.

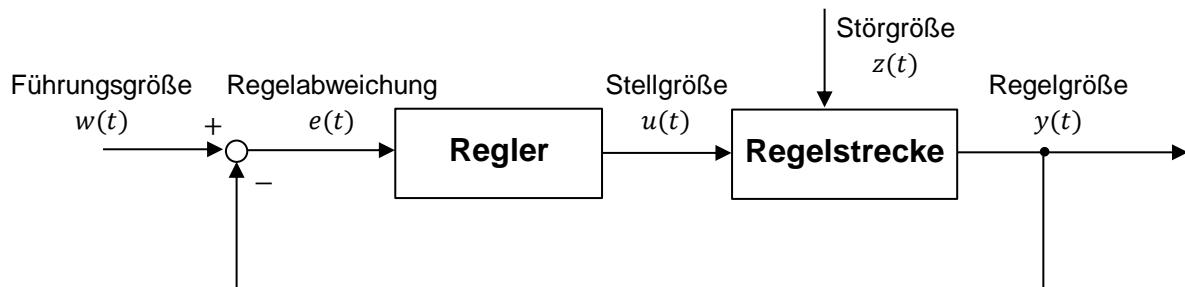


Abbildung 2: Struktur eines Regelkreises. Erstellt auf Basis von [4]

Abbildung 2 zeigt den Aufbau eines Regelkreises. In seiner vereinfachten Form besteht er aus einem Regler und der zu regelnden Strecke, die durch die Stellgröße  $u(t)$  und die Störgröße  $z(t)$  beeinflusst wird. Die Regelgröße wird negativ rückgekoppelt und mit der Führungsgröße verglichen. Die Differenz zwischen Regelgröße und Führungsgröße ist die Regelabweichung  $e(t)$ , die durch einen geeigneten Regler minimiert werden soll. [4]

$$e(t) = w(t) - y(t) \quad (1)$$

In Abhängigkeit von der Regelabweichung bestimmt der Regler dann die Eingangsgröße beziehungsweise die Stellgröße, die wiederum den Prozesszustand ändert. Die Reaktion einer Regelgröße auf die Änderung der Stellgröße wird durch das Übertragungsverhalten eines Systems beschrieben. [4]

$$u(t) = k(e(t)) \quad (2)$$

Ein geeignetes Regelgesetz  $k(e(t))$  für den gegebenen Prozess zu finden, ist eine der Hauptaufgaben eines Regelungstechnikers [4].

Ein Beispiel für einen Regelkreis ist das Autofahren. In diesem Fall ist der Mensch der Regler, das Auto das zu regelnde System und die Geschwindigkeit die Regelgröße, die gemessen und über den Tacho angezeigt wird. Die Stellgröße ist hier der Winkel des Gaspedals, der die Geschwindigkeit des Autos beeinflusst. Das Ziel des Fahrers ist es, eine bestimmte Geschwindigkeit zu erreichen oder zu halten. Fährt das Auto zu langsam, muss der Fahrer das Gaspedal weiter durchtreten, um die Geschwindigkeit zu erhöhen. Fährt es zu schnell, muss der Fahrer das Gaspedal zurücknehmen, um die Geschwindigkeit zu verringern. Auf diese Weise wird die gewünschte Geschwindigkeit erreicht und gehalten.

### 2.1.3 Übertragungsverhalten

Das Übertragungsverhalten eines Systems beschreibt die Reaktion der Ausgangsgröße auf eine Änderung der Eingangsgröße und kann zum Beispiel mithilfe der Sprung- und Impulsantwort untersucht werden. Die Analyse der Sprung- und Impulsantwort kann dazu beitragen, das dynamische Verhalten eines Systems vorherzusagen, die Stabilität des Systems zu beurteilen und die Reaktion des Systems auf Störungen zu verstehen. Das Übertragungsverhalten kann durch sogenannte Übertragungsglieder modelliert werden. [4]

**Sprungantwort:** Die Sprungantwort beschreibt die zeitliche Reaktion des Systems auf eine sprungförmige Änderung der Stellgröße. Sie kann zur Entwicklung und Optimierung von Regelstrategien und damit zur Verbesserung des Systemverhaltens verwendet werden. Die Stellgröße wird durch die Sprungfunktion mit der Sprunghöhe  $u_0$  dargestellt. [4]

$$\sigma(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases} \quad (3)$$

$$u(t) = u_0 \sigma(t) \quad (4)$$

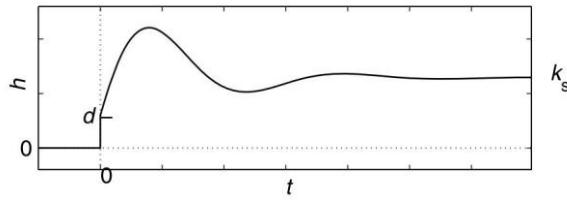


Abbildung 3: Sprungantwort eines Systems zweiter Ordnung [4]

Die Sprungantwort einer Sprungfunktion mit der Sprunghöhe  $u_0 = 1$  wird als Übergangsfunktion  $h(t)$  bezeichnet. Abbildung 3 zeigt ein Beispiel einer Sprungantwort oder Übergangsfunktion eines Systems zweiter Ordnung. Es ist zu erkennen, dass die Übergangsfunktion für  $t \rightarrow \infty$  gegen einen bestimmten Grenzwert  $k_s$  konvergiert, der die statische Verstärkung des Systems darstellt. Die statische Verstärkung gibt an, wie stark das System auf eine Änderung des Eingangssignals reagiert. Der Wert von  $d$  gibt an, inwieweit die Ausgangsgröße der Eingangsgröße unmittelbar und ohne Verzögerung folgt. Systeme, für die  $d \neq 0$  ist, werden als sprungfähige Systeme bezeichnet. [4]

**Impulsantwort:** Die Impulsantwort beschreibt die zeitliche Reaktion des Systems auf eine impulsartige Änderung der Eingangsgröße. Sie wird häufig verwendet, um zu verstehen, wie das System auf Störungen reagiert. Das System wird durch einen kurzen Impuls angeregt, der durch den Rechteckimpuls  $r_\varepsilon(t)$  dargestellt werden kann. [4]

$$r_\varepsilon(t) = \begin{cases} \frac{1}{\varepsilon}, & 0 \leq t \leq \varepsilon \\ 0, & \text{sonst} \end{cases} \quad (5)$$

Für jedes  $\varepsilon$  ist die Fläche des Rechteckimpulses gleich eins. Ein System wird dabei mit einem Dirac-Impuls  $\delta(t)$  angeregt, der sich aus dem Rechteckimpuls für  $\varepsilon \rightarrow 0$  ergibt. [4]

$$\delta(t) = \lim_{\varepsilon \rightarrow 0} r_\varepsilon \quad (6)$$

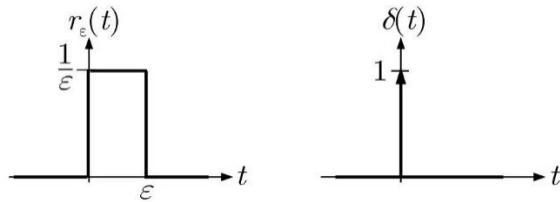


Abbildung 4: Darstellung des Rechteckimpuls und des Dirac-Impuls [4]

Der Dirac-Impuls ist die Ableitung der Sprungfunktion  $\sigma(t)$  und ist unendlich groß und unendlich kurz. Er wird graphisch durch einen Pfeil der Länge eins dargestellt. In der Realität ist der Dirac-Impuls nicht realisierbar, ein ähnliches Systemverhalten kann aber mit dem Rechteckimpuls für ein kleines  $\varepsilon$  erreicht werden. [4]

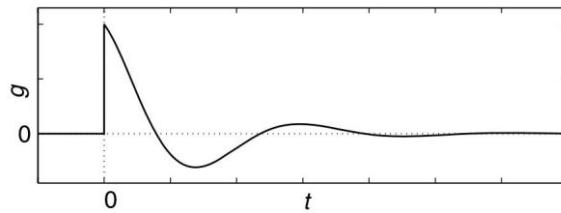


Abbildung 5: Impulsantwort eines Systems zweiter Ordnung [4]

Die Impulsantwort eines durch den Dirac-Impuls angeregten Systems wird auch als Gewichtsfunktion  $g(t)$  bezeichnet. Abbildung 5 zeigt ein Beispiel für eine Impulsantwort oder Gewichtsfunktion. Ein stabiles System kehrt nach der Anregung mit einem Impuls mit einer Zeitverzögerung in seinen Ausgangszustand zurück.

#### 2.1.4 Übertragungsglieder

Übertragungsglieder dienen zur Modellierung der Reaktion von Ausgangsgrößen eines dynamischen Systems auf Änderungen von Eingangsgrößen. Sie werden in Strukturbildern als Blöcke dargestellt und können nach dem qualitativen Verlauf ihrer Sprungantwort in Proportional-, Integrier-, Differenzier- und Totzeitglieder unterteilt werden. Die Art des Übertragungsgliedes bestimmt das Verhalten des modellierten Systems bei Änderung der Eingangsgröße. [4, 5]

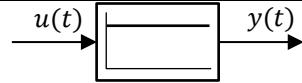
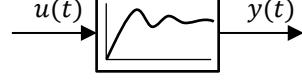
**Proportionalglieder (P-Glieder):** Proportionalglieder sind dynamische Übertragungsglieder, die bei konstanter Eingangsgröße  $u(t) = \bar{u}$  im stationären Zustand eine dem Wert der Eingangsgröße proportionale Ausgangsgröße haben. [4]

$$y_s(t) \sim \bar{u} \quad (7)$$

Proportionalglieder können in verzögerte und unverzögerte Glieder unterteilt werden. Ein P-Glied ohne Verzögerung ist ein statisches Übertragungsglied, bei dem das Ausgangssignal zum Zeitpunkt  $t$   $k_s$  mal so groß ist wie das Eingangssignal. Ein P-Glied mit Verzögerung wird als  $PT_n$ -Glied bezeichnet, wobei  $n$  die Ordnung des Systems angibt. Somit ist das  $PT_1$ -Glied ein Verzögerungsglied erster Ordnung und das  $PT_2$ -Glied ein Verzögerungsglied zweiter Ordnung. [4]

Im Gegensatz zum verzögerungsfreien P-Glied ist bei  $PT_n$ -Gliedern die Ausgangsgröße erst für  $t \rightarrow \infty$   $k_s$ -mal so groß wie die Eingangsgröße. Die Ausgangsgröße nimmt nicht sofort, sondern erst nach einer Verzögerung den Endwert an.  $PT_n$ -Glieder können auch durch Reihenschaltung von  $n$   $PT_1$ -Gliedern realisiert werden. Je mehr Glieder in Reihe geschaltet sind, desto länger wird das Eingangssignal verzögert und desto kleiner wird die Amplitude der Impulsantwort. [4]

Tabelle 1: Funktionalbeziehungen und Blocksymbole von Proportional Gliedern [4, 5]

Übertragungsglied	Funktionalbeziehung	Blocksymbol
P-Glied	$y(t) = k_s u(t)$	
$PT_1$ -Glied	$T\dot{y}(t) + y(t) = k_s u(t)$	
$PT_2$ -Glied	$T^2\ddot{y}(t) + 2dT\dot{y}(t) + y(t) = k_s u(t)$	

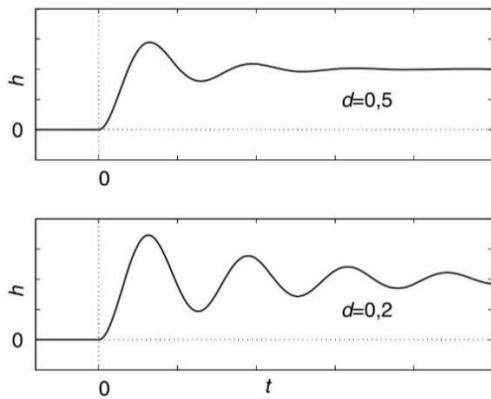


Abbildung 6: Sprungantwort von schwingfähigen PT<sub>2</sub>-Gliedern mit kleiner Dämpfung [4]

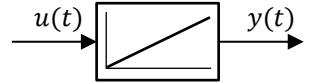
Bei Verzögerungsgliedern ist  $T$  eine Zeitkonstante, die die Verzögerungszeit beeinflusst, und  $d$  ist ein Dämpfungsfaktor. Die Variation von  $T$  und  $d$  kann zu sehr unterschiedlichen Sprungantworten führen. Der Einfluss des Dämpfungsfaktors auf die Übergangsfunktion eines PT<sub>2</sub>-Gliedes ist in Abbildung 6 dargestellt. Liegt der Dämpfungsfaktor im Bereich  $0 < d < 1$ , so tritt ein Überschwingen der Sprungantwort auf. Je kleiner der Dämpfungsfaktor ist, desto größer ist das Überschwingen. [4]

**Integrierglieder (I-Glieder):** Bei Integriergliedern wird die Ausgangsgröße durch Integration der Eingangsgröße bestimmt. Bei konstanter Eingangsgröße  $u(t) = \bar{u}$  nimmt die Ausgangsgröße die Form einer Rampenfunktion an. Die Ausgangsgröße nimmt nur dann einen konstanten Wert an, wenn die Eingangsgröße gleich null ist. Bei stationärem Systemverhalten ist die Ausgangsgröße proportional zur integrierten Eingangsgröße. [4]

$$y_s(t) \sim \int_0^t \bar{u} dt = \bar{u}t \quad (8)$$

Tabelle 2: Funktionalbeziehung und Blocksymbol vom I-Glied [4, 5]

Übertragungsglied	Funktionalbeziehung	Blocksymbol

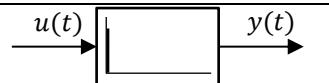
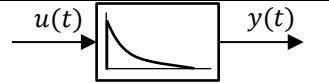
I-Glied	$y(t) = \frac{1}{T_I} \int_0^t u(\tau) d\tau + y(0)$	
---------	--	---

**Differenzierglieder (D-Glieder):** Bei Differenziergliedern wird die Ausgangsgröße durch die Änderung der Eingangsgröße bestimmt. Bei konstanter Eingangsgröße konvergiert die Ausgangsgröße demnach gegen null. Bei stationärem Systemverhalten ist die Ausgangsgröße proportional zur abgeleiteten Eingangsgröße. [4]

$$y_s(t) \sim \frac{du}{dt} \quad (9)$$

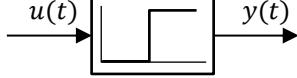
Die Sprungantwort des verzögerungsfreien D-Gliedes enthält einen Dirac-Impuls und ist daher nur theoretisch möglich. In realen Systemen tritt ein D-Verhalten immer nur mit Verzögerung auf. Das verzögerte D-Glied wird als DT<sub>n</sub>-Glied bezeichnet, wobei n die Anzahl der Verzögerungsglieder beschreibt. Ein DT<sub>1</sub>-Glied wird durch ein PT<sub>1</sub>-Glied zusätzlich gedämpft. [4]

Tabelle 3: Funktionalbeziehungen und Blocksymbole von Differenziergliedern [4, 5]

Übertragungsglied	Funktionalbeziehung	Blocksymbol
D-Glied	$y(t) = T_D \dot{u}$	
DT <sub>1</sub> -Glied	$T \dot{y} + y(t) = T_D \dot{u}$	

**Totzeitglied (T<sub>t</sub>-Glied):** Ein Totzeitglied verschiebt das Eingangssignal auf der Zeitachse um  $T_t$ . Sie werden in der Regel mit anderen Übertragungsgliedern kombiniert. [4]

Tabelle 4: Funktionalbeziehung und Blocksymbol vom Totzeitglied [4, 5]

Übertragungsglied	Funktionalbeziehung	Blocksymbol
$T_t$ -Glied	$y(t) = u(t - T_t)$	

## 2.2 Künstliche Intelligenz

Der Begriff Künstliche Intelligenz (KI) wurde erstmals 1956 auf der Konferenz „Dartmouth Summer Research Project on Artificial Intelligence“ definiert [6]. Diese Konferenz markierte den Beginn der KI-Forschung und brachte führende Wissenschaftler zusammen, um das Potenzial „intelligenter Maschinen“ zu erforschen [6]. Eine frühe praktische Anwendung der KI war der „Logic Theorist“ von Newell und Simon aus dem Jahr 1956 [7]. Dabei handelte es sich um ein Programm, das in der Lage war, einige Theoreme aus Whitehead und Russells „Principia Mathematica“ selbstständig zu beweisen [7]. Seitdem hat sich das Gebiet der KI stetig weiterentwickelt, mit Phasen der Euphorie und Phasen der Enttäuschung. Diese Entwicklung ist in Abbildung 7 dargestellt.

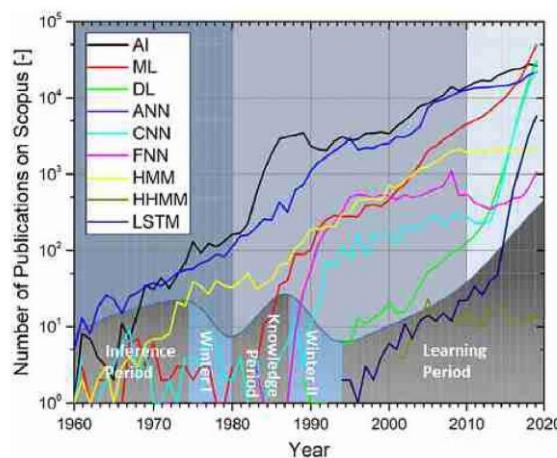


Abbildung 7: Anzahl der Veröffentlichungen von verschiedenen ML-Methoden auf Scopus und qualitative Darstellung der jeweiligen KI-Epochen über die Zeit [8]

Eine der am weitesten verbreiteten Methoden der KI ist das maschinelle Lernen und das untergeordnete Deep Learning (Tiefes Lernen), das künstliche neuronale Netze als Modell verwendet [8]. In den folgenden Kapiteln werden das maschinelle Lernen und die untergeordneten Lernverfahren, die zum Training eines Modells verwendet werden können, erläutert. Der Aufbau von künstlichen neuronalen Netzen wird in Abschnitt 2.2.3 näher erläutert.

### 2.2.1 Maschinelles Lernen

Nach Arthur Samuel ist maschinelles Lernen das Gebiet, das Computern die Fähigkeit verleiht, selbstständig zu lernen und Probleme zu lösen, ohne explizit dafür programmiert zu werden. Bekannt wurde Arthur Samuel 1959 durch ein Programm, das in der Lage war, das Spiel „Dame“ selbstständig zu erlernen und nach kurzer Trainingszeit den Entwickler in diesem Spiel zu übertreffen. [9]

Ziel des maschinellen Lernens ist es, möglichst effizient aus vorhandenen Daten zu lernen und neue Daten zu generalisieren. Ein Modell, das effektiv generalisieren kann, ist in der Lage, korrekte Vorhersagen für unbekannte Daten zu treffen, die es während des Lernprozesses nicht gesehen hat. Maschinelles Lernen wird dann eingesetzt, wenn aus den vorhandenen Daten, zum Beispiel durch einen Experten, keine relevanten Informationen und Zusammenhänge abgeleitet werden können. Insbesondere bei komplexen, nichtlinearen Problemen ist sein Einsatz von großem Vorteil. Im Bereich des maschinellen Lernens gibt es viele verschiedene Algorithmen, deren Auswahl von der jeweiligen Problemstellung abhängt. Es gibt keinen allgemeingültigen Algorithmus, daher muss im Vorfeld geprüft werden, welcher für das vorliegende Problem geeignet ist. Für ein komplexes Modell sind oft große Datenmengen und hohe Rechenleistung erforderlich, die aufgrund des exponentiellen Wachstums der Datenmengen sowie der Rechen- und Speicherleistung in den letzten zehn Jahren zunehmend zur Verfügung stehen. Das Interesse am maschinellen Lernen ist daher in den letzten Jahren stark gestiegen. [8, 10, 11]

## 2.2.2 Lernmethoden

Maschinelles Lernen lässt sich in drei wesentliche Lernmethoden unterteilen: das überwachte Lernen (Supervised Learning), das unüberwachte Lernen (Unsupervised Learning) und das verstärkende Lernen (Reinforcement Learning) [10]. Die verschiedenen Lernmethoden mit den jeweiligen Algorithmen sind in Abbildung 8 dargestellt und werden in diesem Abschnitt erläutert.

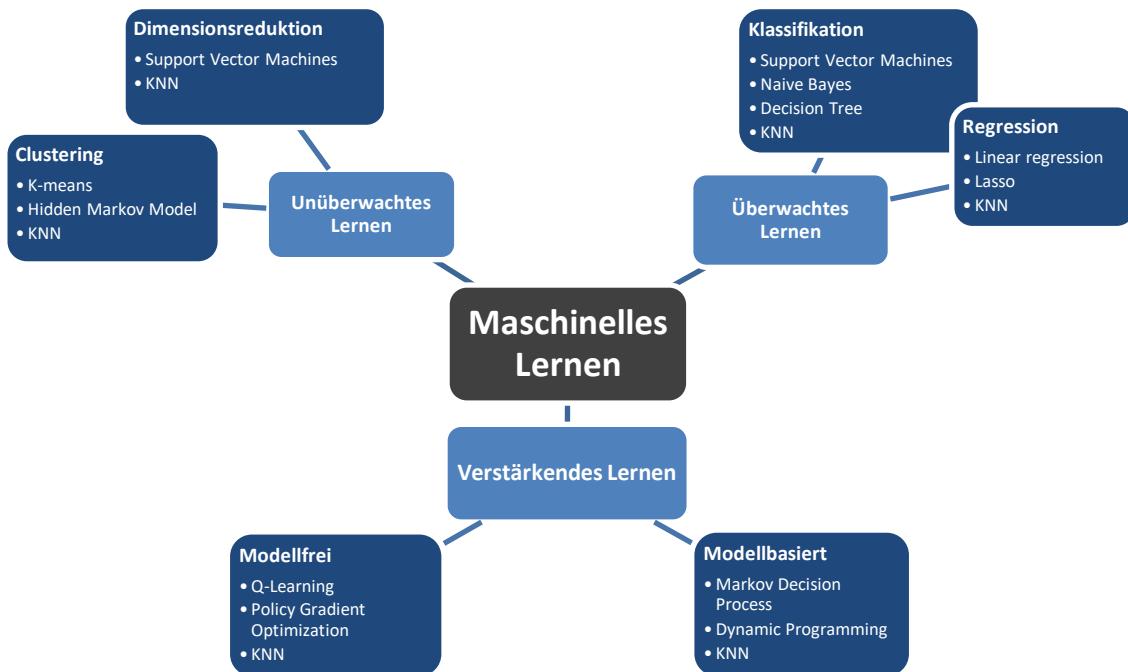


Abbildung 8: Klassen des maschinellen Lernens mit den jeweiligen Algorithmen. Erstellt auf Basis von [8, 12, 13].

**Überwachtes Lernen:** Das Ziel des überwachten Lernens ist es, eine Funktion zu erlernen, die bestimmte Eingaben auf zugehörige Ausgaben abbildet. Dazu werden gelabelte Datensätze benötigt, bei denen die zugehörigen Ausgaben bereits bekannt sind. Vor dem Training werden die Daten in einen Trainingsdatensatz und einen Testdatensatz aufgeteilt, um später die Genauigkeit der erlernten Funktion zu

überprüfen. Dieser Lernmethode werden unter anderem die Klassifikation und die Regression zugeordnet. Bei der Klassifikation handelt es sich um ein Problem, bei dem die Ausgabe diskret ist, es also nur eine Lösung aus einem bekannten Lösungsraum gibt. Sie wird zum Beispiel bei der Bilderkennung eingesetzt. Die Regression wird verwendet, wenn die Ausgabe kontinuierlich und numerisch sein soll, zum Beispiel bei der Wettervorhersage. Überwachtes Lernen wird für Vorhersagen auf der Grundlage historischer Daten verwendet und wird von etwa 70% aller KI-Programme eingesetzt.

[8, 10, 11, 13]

Ein Beispiel für die Anwendung von überwachtem Lernen ist der MNIST-Datensatz, der handgeschriebene Zahlen von null bis neun enthält, die durch ein  $28 \times 28$  Pixel großes Graustufenbild repräsentiert werden. Jedes Bild ist mit der entsprechenden Ziffer beschriftet. Es handelt sich also um ein Klassifikationsproblem, bei dem eine Funktion erlernt werden soll, die den  $28 \times 28$  Pixeln (784 Inputs) mit unterschiedlichen Graustufen eine Ziffer zwischen null und neun zuordnet. [14]

**Unüberwachtes Lernen:** Im Gegensatz zum überwachten Lernen benötigt das unüberwachte Lernen keine beschrifteten Daten, was den zusätzlichen Aufwand für Experten reduziert. Bei dieser Lernmethode muss der Algorithmus die Daten selbstständig erkunden, um inhärente Zusammenhänge zu erkennen. [8, 10]

Unüberwachtes Lernen wird hauptsächlich für „Clustering“ und „Dimensionsreduktion“ verwendet. Beim Clustering werden Daten mit ähnlichen Strukturen und Eigenschaften gruppiert. Dimensionsreduktion hingegen versucht die Dimensionen der Eingabedaten zu reduzieren, indem Eingaben, die wenig oder keinen Einfluss auf die Ausgabe haben, herausgefiltert werden. [8, 10, 11]

**Verstärkendes Lernen:** Bei verstärkendem Lernen (Reinforcement Learning) führt ein lernender Agent Aktionen aus und erhält Rückmeldung aus der Umgebung, wie sich diese Aktionen auf die Zielerreichung auswirken. [15]

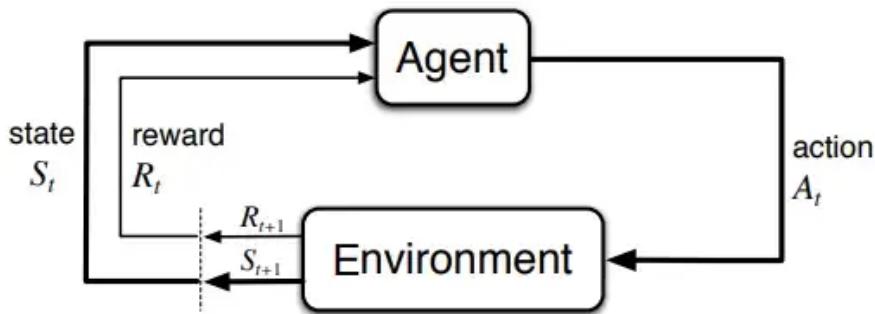


Abbildung 9: Ablauf vom Reinforcement Learning [16]

Der Agent wird mithilfe von Belohnungen und Bestrafungen programmiert, ein bestimmtes Ziel zu erreichen, ohne dabei explizit vorzuschreiben, wie das Ziel erreicht werden soll. Dazu muss der Algorithmus durch „Trial-and-Error“ die optimalen Aktionen für die gegebene Situationen selbstständig erlernen. Dabei entsteht ein Konflikt zwischen „Exploration“ (Erkundung) und „Exploitation“ (Ausnutzung). Um eine hohe Belohnung zu erhalten, sollte der Agent die erfolgreichen Strategien ausnutzen. Um erfolgreiche Strategien zu entwickeln und zu verbessern, muss der Agent jedoch neue Aktionen ausprobieren, wobei auch das Risiko von unerwünschten Ergebnissen besteht. Eine weitere Herausforderung beim Reinforcement Learning besteht darin, dass sich in den meisten komplexen Systemen Handlungen nicht nur auf die nächste Belohnung, sondern auch auf zukünftige Zustände und damit auf zukünftige Belohnungen auswirken. Eine Handlung, die auf den ersten Blick nicht unmittelbar zielführend erscheint, kann in der Zukunft notwendig sein, um das Ziel zu erreichen, was im Nachhinein nur schwer festzustellen ist. Reinforcement Learning basiert auf einem kontinuierlichen Fluss neuer Trainingsdaten und wird daher auch als Online-Learning bezeichnet. [8, 15, 17]

Es gibt zwei verschiedene Ansätze, um ein Reinforcement Learning Problem zu lösen. Der erste Ansatz besteht darin, die Verhaltensweisen auszuwählen und zu entwickeln, die in der Umgebung am effektivsten sind. Dieser Ansatz wird beispielsweise von genetischen Algorithmen verwendet, die ähnlich wie die darwinsche Evolutionstheorie

in der Biologie funktionieren. Die zweite Möglichkeit besteht darin, statistische Techniken und dynamische Programmierung zu verwenden, um den Nutzen verschiedener Aktionen in der Umwelt zu jedem Zeitpunkt zu bewerten und unter den möglichen Strategien diejenige auszuführen, die wahrscheinlich die beste zukünftige Belohnung bringt. [17, 18]

### 2.2.3 Künstliche neuronale Netze

Das am häufigsten verwendete Verfahren im maschinellen Lernen ist das künstliche neuronale Netz (KNN). Die Funktionsweise eines KNN ist lose an das Verständnis des menschlichen Gehirns aus den 1960er Jahren angelehnt [19]. Der folgende Abschnitt erläutert die Funktionsweise von KNNs und geht dabei auf verschiedene Aktivierungsfunktionen sowie den Backpropagation-Algorithmus (Fehlerrückführung) ein.

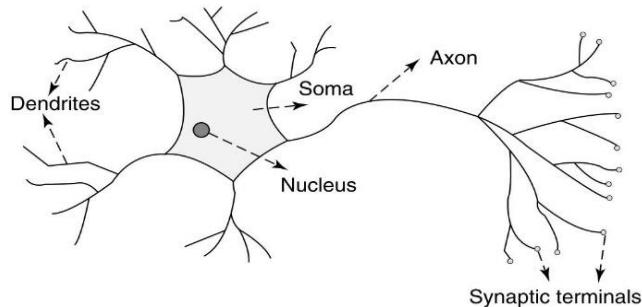
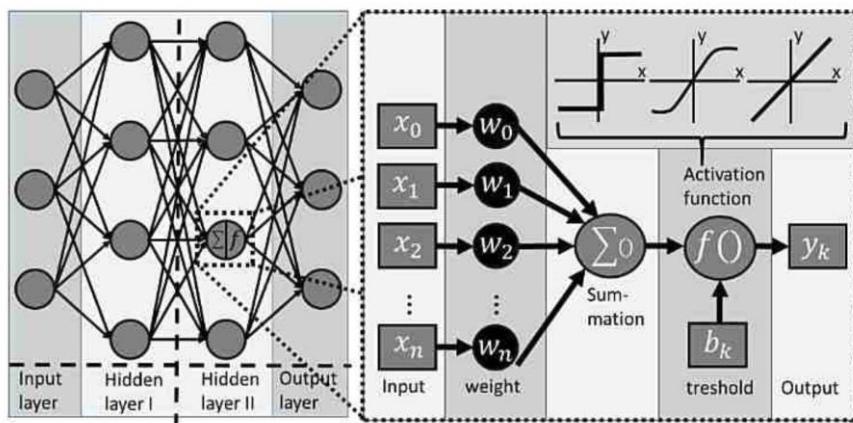


Abbildung 10: Neuron eines Säugetiers [20]

Das menschliche Gehirn besteht aus über 80 Milliarden miteinander verbundenen Neuronen, wobei jedes Neuron eine Zelle ist, die durch biochemische Reaktionen Informationen empfangen, verarbeiten und senden kann. Jedes Neuron verfügt über ein Axon, das die Zelle durch Axon-Terminalen mit anderen Neuronen verbindet. Die Übertragung von Signalen zwischen Neuronen erfolgt durch einen komplexen chemischen Prozess. [20]



**Abbildung 11:** Links: Darstellung eines künstlichen neuronalen Netzes. Rechts: Darstellung eines einzelnen Neurons mit den gewichteten Eingaben und der Aktivierungsfunktion. [18]

Künstliche neuronale Netze bestehen aus Neuronen, die in Schichten angeordnet und miteinander verbunden sind, wobei jede Verbindung eine andere Gewichtung hat. Die erste Schicht des Netzes wird als Eingabeschicht und die letzte als Ausgabeschicht bezeichnet. Jedes Eingabe-Neuron repräsentiert einen Eingabeparameter und jedes Ausgabe-Neuron einen Ausgabeparameter. Eingabeparameter können numerische oder binäre Werte sein, zum Beispiel Prozessparameter oder die Farbe eines Pixels in einem Bild. Schichten, die zwischen Eingabeschicht und Ausgabeschicht liegen, werden als verborgene Schichten bezeichnet. Die Schwelle, ab der ein Neuron aktiv wird, wird durch eine sogenannte Aktivierungsfunktion bestimmt. Sie bestimmt anhand der summierten und gewichteten Eingaben des Neurons und einem konstanten „Bias“, ob ein Neuron aktiv ist und wie groß dessen Ausgabe ist. Die Ausgabe wird dann an die Neuronen der nächsten Schicht weitergegeben. [18, 21]

Als Eingaben  $x_{k,l}$  bekommt ein einzelnes Neuron  $k$  aus der Schicht  $l$  die Ausgaben  $y_{n,l-1}$  aller Neuronen  $n$  aus der vorherigen Schicht  $l - 1$  [22] (siehe Abbildung 11 rechts).

$$\mathbf{x}_{k,l} = \mathbf{y}_{l-1} \quad (10)$$

Die Ausgabe  $y_{k,l}$  des Neurons  $k$  in der Schicht  $l$  des KNN wird dann mit der Aktivierungsfunktion  $f_l$  berechnet [22]

$$y_{k,l} = f_l \left( \sum_n w_{k,n,l} y_{n,l-1} + b_{k,l} \right) \quad (11)$$

wobei die Summe über die gewichteten Eingaben des Neurons  $k$  in der Schicht  $l$  gebildet wird.  $y_{n,l-1}$  ist die Ausgabe des Neurons  $n$  aus der vorherigen Schicht  $l - 1$  und  $w_{k,n,l}$  das Gewicht dieser Verbindung.  $b_{k,l}$  ist der Bias. In Vektor Schreibweise:

$$\mathbf{y}_l = f_l(\mathbf{w}_l \mathbf{y}_{l-1} + \mathbf{b}_l) \quad (12)$$

Je nach Aktivierungsfunktion können Linearitäten, Nichtlinearitäten oder andere Eigenschaften in das Modell eingeführt werden [18]. Sie sind daher notwendig, um vor allem komplexe, nichtlineare Zusammenhänge in den Daten zu erkennen. Die Wahl der Aktivierungsfunktion hängt von der Problemstellung ab und hat großen Einfluss auf die Vorhersagegenauigkeit des neuronalen Netzes. Die am häufigsten verwendeten Aktivierungsfunktionen sind nichtlinear, da sie unempfindlicher gegenüber fehlerhaften Daten sind als lineare Aktivierungsfunktionen. Bekannte Beispiele für nichtlineare Aktivierungsfunktionen sind Sigmoid, Tanh und ReLU. [21] Die Funktionen sind in Abbildung 12 dargestellt.

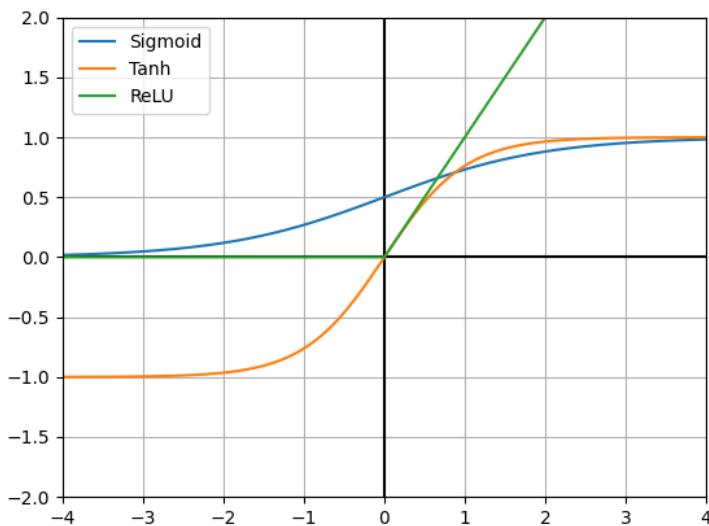


Abbildung 12: Darstellung der Sigmoid, Tanh und ReLU Aktivierungsfunktion

**Sigmoid-Funktion:** Die Sigmoidfunktion skaliert Werte auf den Bereich zwischen null und eins. Sie ist stetig und differenzierbar, was die Verwendung von „Backpropagation“ ermöglicht. Die Sigmoidfunktion wird häufig als Aktivierungsfunktion verwendet, um die Ausgaben von Neuronen auf diesen Bereich zu beschränken. [21]

$$f_{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (13)$$

**Tanh-Funktion:** Die Tangens-Hyperbolicus-Funktion ähnelt der Sigmoid-Funktion, ist aber im Vergleich zu dieser punktsymmetrisch zum Ursprung. Die Werte der Tanh-Funktion liegen zwischen minus eins und eins, was negative Vorzeichen bei den Ausgaben ermöglicht. Wie die Sigmoidfunktion ist auch die Tanh-Funktion stetig und differenzierbar, wodurch „Backpropagation“ möglich ist. Sie wird häufig als Aktivierungsfunktion verwendet, wenn die Ausgaben der Neuronen auch negativ sein sollen. [21]

$$f_{Tanh}(x) = \frac{2}{1 + e^{-2x}} - 1 = 2f_{Sigmoid}(2x) - 1 = \tanh(x) \quad (14)$$

**ReLU-Funktion:** Die Rectified-Linear-Unit-Funktion ist eine der am häufigsten verwendeten Aktivierungsfunktionen, da sie einen biologischen Bezug zu einem realen Neuron hat und eine der effizientesten Aktivierungsfunktionen ist [23]. Sie funktioniert so, dass bei negativen Werten das Neuron inaktiv bleibt und nur bei positiven Werten aktiv wird. Diese Funktion ist effizienter als alle anderen Aktivierungsfunktionen, da nicht alle Neuronen gleichzeitig aktiv sind [21]. Bei der Verwendung von ReLU kann es jedoch vorkommen, dass Neuronen „sterben“, da der Gradient für negative Werte null ist und somit die Gewichte für negative Werte nicht durch „Backpropagation“ aktualisiert werden, was den Lernprozess des künstlichen neuronalen Netzes erschwert [23]. Um dieses Problem zu umgehen, kann zum Beispiel die Leaky-ReLU Funktion verwendet werden, die für negative Werte statt null eine kleine lineare Komponente und damit auch einen Gradienten hat, der für negative Werte nicht null ist. Das Problem der „toten“ Neuronen tritt hier nicht auf, dafür sind jedoch wieder alle Neuronen aktiv.

Sowohl ReLU als auch Leaky ReLU sind für  $x \neq 0$  stetig und differenzierbar. Da die Ableitung für  $x = 0$  nicht definiert ist, wird sie entweder auf eins oder auf null gesetzt, Backpropagation ist auch hier möglich. [21, 23]

$$f_{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} = \max(0, x) \quad (15)$$

$$f_{Leaky\ ReLU}(x) = \begin{cases} x, & x \geq 0 \\ ax, & x < 0 \end{cases} = \max(ax, x) \quad (16)$$

**Backpropagation (Fehlerrückführung):** Damit ein KNN lernen und seine Vorhersagegenauigkeit verbessern kann, müssen die Gewichte der einzelnen Verbindungen zwischen den Neuronen möglichst optimal angepasst werden. Dazu wird der Backpropagation Algorithmus verwendet, der eine Kosten- oder Fehlerfunktion minimiert, indem die Gewichte des KNN in einer bestimmten Weise verändert werden. Der Fehler bezieht sich dabei auf die Abweichung der Vorhersage des KNN von dem tatsächlichen Wert. [22]

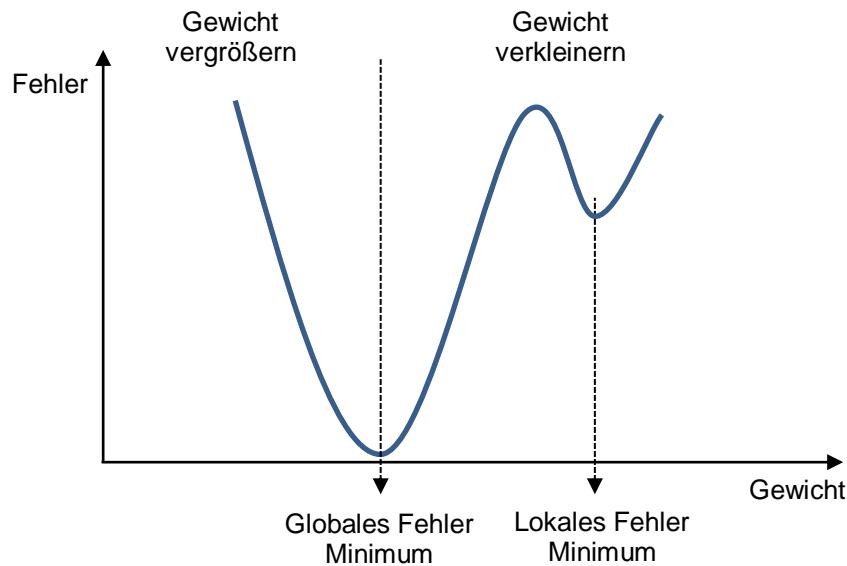


Abbildung 13: Einfluss der Gewichtung auf den Fehler. Erstellt auf Basis von [24]

Zur Anpassung der Gewichte wird die partielle Ableitung der Fehlerfunktion  $C$  in Bezug auf die Gewichtung  $w$  des neuronalen Netzes berechnet ( $\frac{\partial C}{\partial w}$ ). Sie gibt an, wie schnell

sich der Fehler ändert, wenn die Gewichte geändert werden. Abbildung 13 zeigt ein Beispiel für eine Funktion des Fehlers in Abhängigkeit von der Gewichtung. Das Ziel des Backpropagation-Algorithmus ist es, die Gewichte in Richtung des absteigenden Gradienten beziehungsweise in Richtung des Fehlerminimums anzupassen, wobei die Größe der jeweiligen Gewichtsänderung proportional zum Einfluss des Gewichts auf den Fehler ist. Die Anpassung der Gewichte erfolgt iterativ über die Trainingsdaten. Dabei kann es vorkommen, dass das Gewicht in Richtung eines lokalen Minimums angepasst wird, wodurch das globale Optimum nie erreicht wird. [22]

## 2.3 Fehlerberechnung bei der Vorhersage

Zur Beurteilung der Genauigkeit der Vorhersage eines Modells, ist ein Vergleich zwischen dem vorhergesagten Wert und dem tatsächlichen Wert notwendig. In diesem Kapitel wird die Fehlerberechnung bei der Klassifikation und bei der Regression erläutert.

### 2.3.1 Berechnung bei der Klassifikation

Bei der binären Klassifikation (positiv oder negativ) gibt es vier verschiedene Resultate einer Vorhersage [25].

**Richtig positiv (RP):** Das Modell hat „positiv“ vorhergesagt und der tatsächliche Wert ist positiv.

**Richtig negativ (RN):** Das Modell hat „negativ“ vorhergesagt und der tatsächliche Wert ist negativ.

**Falsch positiv (FP):** Das Modell hat „positiv“ vorhergesagt und der tatsächliche Wert ist jedoch negativ.

**Falsch negativ (FN):** Das Modell hat „negativ“ vorhergesagt und der tatsächliche Wert ist jedoch positiv.

Tabelle 5: Mögliche Resultate einer binären Klassifizierung

Richtig positiv (RP) ✓	Falsch positiv (FP) ✗
Tatsächlich: Positiv	Tatsächlich: Negativ
Falsch negativ (FN) ✗	Richtig negativ (RN) ✓
Tatsächlich: Positiv	Tatsächlich: Negativ

Die Genauigkeit A der Vorhersage wird mit (17) berechnet [26].

$$A = \frac{\text{Anzahl richtige Vorhersagen}}{\text{Gesamtzahl der Vorhersagen}} = \frac{RP + RN}{RP + RN + FP + FN} \quad (17)$$

Wenn mit einem unausgeglichenen Datensatz gearbeitet wird, kann es vorkommen, dass die Gesamtgenauigkeit zwar hoch ist, das Modell aber nicht unbedingt in der Lage ist, zwischen verschiedenen Fällen zu unterscheiden. Zum Beispiel soll aus 100 Bildern erkannt werden, auf welchen ein Apfel zu sehen ist. Auf 10 Bildern ist ein Apfel zu sehen, auf den restlichen 90 nicht. Wenn jedes Bild als „kein Apfel“ klassifiziert würde, wäre die Genauigkeit 90%, obwohl keine Unterscheidung getroffen wird. Daher ist die Genauigkeit in den meisten Fällen nicht ausreichend, um eine Aussage über die Effektivität des Modells zu treffen. [26]

Um dieses Problem zu umgehen wird die Präzision und Trefferquote benutzt. Die Präzision  $P$  wird definiert als Anteil der positiven Vorhersagen die tatsächlich richtig waren [27].

$$P = \frac{RP}{RP + FP} \quad (18)$$

Die Trefferquote (Recall)  $R$  wird definiert als Anteil der tatsächlich positiven Ergebnisse die richtig identifiziert wurden [27].

$$R = \frac{RP}{RP + FN} \quad (19)$$

Um eine Aussage über die Effizienz eines Modells treffen zu können, müssen sowohl die Genauigkeit P als auch die Trefferquote R berücksichtigt werden. Beide stehen im Konflikt miteinander. Eine Verbesserung der Präzision führt in der Regel zu einer Verringerung der Trefferquote und umgekehrt. Ziel ist es, Präzision und Trefferquote möglichst im Gleichgewicht zu halten. [27]

### 2.3.2 Berechnung bei der Regression

Für Regressionsprobleme stehen verschiedene Methoden zur Fehlerberechnung zur Verfügung. Welche Methode letztendlich verwendet wird, hängt von der Problemstellung ab und ist ein weiterer Hyperparameter, der für das vorliegende Problem getestet werden muss [28].

**Skalenabhängige Fehler:** Die üblicherweise verwendeten Methoden zur Fehlerberechnung sind skalenabhängig. Sie sind nützlich für den Vergleich verschiedener Methoden, die auf denselben Datensatz angewendet werden, aber nicht für den Vergleich verschiedener Datensätze, die unterschiedlich skaliert sind. Bekannte Beispiele für skalenabhängige Fehlerberechnungen sind Mean Square Error (MSE), Root Mean Square Error (RMSE) und Mean Absolute Error (MAE). [28, 29]

Der absolute Vorhersagefehler  $e_t$  zum Zeitpunkt  $t$  wird definiert als

$$e_t = y_t - \hat{y}_t \quad (20)$$

wobei  $y_t$  der tatsächliche Wert zum Zeitpunkt  $t$  ist und  $\hat{y}_t$  der vorhergesagte Wert des Modells für den Zeitpunkt  $t$ . Über einen bestimmten Horizont  $h$  lässt sich die Gesamtabweichung mit den verschiedenen Methoden bestimmen. [29]

$$MSE = \frac{1}{h} \sum_{i=1}^h e_i^2 \quad (21)$$

$$RMSE = \sqrt{MSE} \quad (22)$$

$$MAE = \frac{1}{h} \sum_{i=1}^h |e_i| \quad (23)$$

Historisch sind RMSE und MSE auf Grund ihrer Relevanz bei der statistischen Modellierung beliebte Methoden den Fehler zu berechnen, treffen aber auf Kritik bei der Bewertung von zeitlichen Vorhersagen [28].

**Prozentuale Fehler:** Der Vorteil von prozentualen Fehlern ist ihre Skalenunabhängigkeit, weshalb sie häufig zur Bewertung von Datensätzen mit unterschiedlichen Skalen verwendet werden. Häufig verwendete Methoden sind Mean Absolute Percentage Error (MAPE) und Median Absolute Percentage Error (MdAPE). [28]

$$p_t = 100 \cdot \frac{e_t}{y_t} \quad (24)$$

$$MAPE = \frac{1}{h} \cdot \sum_{i=1}^h |p_i| \quad (25)$$

$$MdAPE = median(|p_i|) \quad (26)$$

Sie sind jedoch ungeeignet für Daten mit kleinen Werten, insbesondere wenn der Datensatz  $y_t = 0$  enthält, da in diesem Fall der prozentuale Fehler undefiniert ist. Für Werte nahe null gibt es große Schwankungen in der Fehlerberechnung. Daher kann MAPE in diesen Fällen deutlich größer sein als MdAPE, da der Median im Allgemeinen resistenter gegen Ausreißer ist als der Mittelwert. Ein weiterer Nachteil ist, dass beide Methoden positive Fehler stärker gewichten als negative. Dies führt zu einer Asymmetrie in der Berechnung. [28]

## 2.4 Prädiktive Modellierung über ein gleitendes Zeitfenster

Die prädiktive Modellierung ist eine Form des maschinellen Lernens, deren Ziel es ist, den zukünftigen Zustand eines Systems anhand bekannter historischer Daten vorherzusagen [30]. Üblicherweise werden dazu statistische Modelle verwendet, die jedoch Schwierigkeiten bei der Modellierung hochdimensionaler, nichtlinearer Systeme haben [31]. Zur Modellierung und Vorhersage komplexer nichtlinearer Systeme eignen sich KNNs, die die inhärenten Systemeigenschaften nur mit Hilfe historischer Daten erlernen können. Für die Modellierung von Zeitfolgen wird in dieser Arbeit ein gleitendes Zeitfenster benutzt. Eine Zeitfolge ist dabei eine Sequenz von historischen Messungen eines Systemzustands in gleichen Zeitabständen [30].

**Gleitendes Zeitfenster:** Die Methode des gleitenden Zeitfensters wandelt sequentielle Daten in Form einer Zeitfolge in ein klassisches Problem des überwachten Lernens um. Damit ist es möglich, ein künstliches neuronales Netz mit Hilfe des Backpropagation-Algorithmus zu trainieren. Für jedes Zeitfenster werden die  $n$  letzten Zustände verwendet, um einen oder mehrere zukünftige Zustände vorherzusagen. Verschiedene Methoden zur Vorhersage über mehrere Zeitschritte werden in Kapitel 2.4.1 vorgestellt. [32]

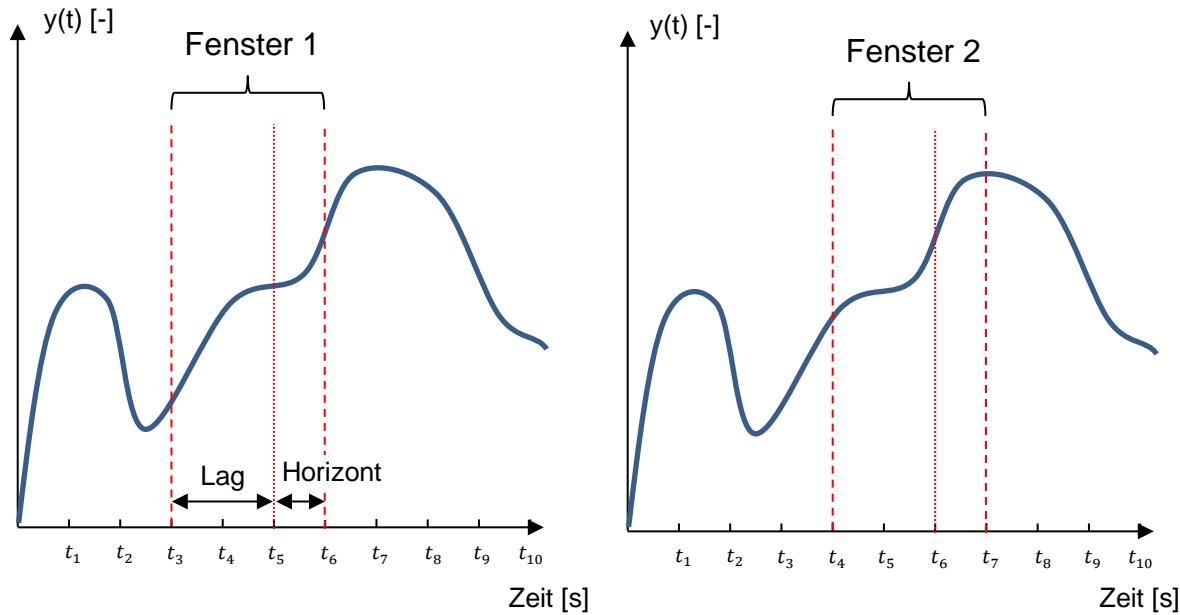


Abbildung 14: Beispiel von einem gleitenden Zeitfenster

Abbildung 14 zeigt ein Beispiel für ein gleitendes Zeitfenster. Die Verzögerung (Lag)  $d$  bezeichnet den Zeitraum der Vergangenheit, der für die Vorhersage berücksichtigt wird [30]. Der Horizont  $h$  gibt an, wie viele zukünftige Zustände vorhergesagt werden sollen und ist in diesem Beispiel gleich eins [30]. Das erste Zeitfenster liegt zwischen  $t_3$  und  $t_6$ , wobei hier mit den Werten für  $t_3$ ,  $t_4$  und  $t_5$  der Wert zum Zeitpunkt  $t_6$  vorhergesagt werden soll. Für die folgenden Schritte wird das Zeitfenster jeweils um eins nach rechts verschoben und dieser Vorgang wiederholt.

Für den Lernprozess müssen alle Werte zu den jeweiligen Zeitpunkten bekannt sein, damit das überwachte Lernen angewendet werden kann. Jedes Zeitfenster stellt dann einen Eintrag im Trainingsdatensatz dar. Bei kontinuierlichen Systemen ist es möglich, das Modell mit den neuen Daten weiter zu trainieren und so die Genauigkeit des Modells über die Zeit zu verbessern. Zur Vorhersage unbekannter zukünftiger Zustände kann dann das trainierte Modell mit den  $n$  letzten Zuständen als Eingabe, verwendet werden.

#### 2.4.1 Methoden für die Vorhersage über mehrere Zeitschritte

Eine Vorhersage über mehrere Zeitschritte hat die Aufgabe, die nächsten  $h$  Werte ( $y_{t+1}, \dots, y_{t+h}$ ) mithilfe der historischen Zeitreihe der letzten  $n$  Werte ( $y_{t-n+1}, \dots, y_t$ ) vorherzusagen, wobei  $h > 1$  der Prognosehorizont ist. Das Problem hierbei ist, dass jeder Wert eine zeitliche Abhängigkeit von den vorhergehenden Werten hat. Beispielsweise hängt die Vorhersage des Wertes  $y_{t+2}$  auch vom Wert  $y_{t+1}$  ab, der jedoch nicht bekannt ist, sondern auch vorhergesagt werden muss. In diesem Kapitel werden drei verschiedene Methoden zur Lösung dieses Problems vorgestellt. Dabei ist  $f$  das Modell, das die vergangenen Werte auf die zukünftigen Werte abbildet und  $e$  die Abweichung vom Modell inklusive Störungen. [33]

**Rekursive Methode:** Bei der rekursiven Methode wird ein Modell  $f$  trainiert, das nur eine Vorhersage über den nächsten Zeitschritt treffen kann [33].

$$y_{t+1} = f(y_{t-n+1}, \dots, y_t) + e_{t+1} \quad (27)$$

Die Vorhersage der nächsten  $h$  Werte erfolgt dann rekursiv, wobei der vorhergesagte Wert jeweils mit als Eingabeparameter für die Vorhersage der folgenden Werte verwendet wird. Dies wird solange wiederholt, bis der gesamte Horizont  $h$  vorhergesagt ist. Ein Nachteil dieser Methode ist die Sensitivität gegenüber Prognosefehlern, da diese für die folgenden Vorhersagen weitergegeben werden und somit über  $h$  immer größer werden. [33]

**Direkte Methode:** Im Gegensatz zur rekursiven Methode werden bei der direkten Methode  $h$  unterschiedliche Modelle  $f_H$  trainiert, die jeweils einen zukünftigen Wert vorhersagen [33].

$$y_{t+H} = f_H(y_{t-n+1}, \dots, y_t) + e_{t+H} \quad (28)$$

Bei dieser Methode werden die vorhergesagten Werte nicht zur Vorhersage neuer Werte verwendet, so dass es zu keiner Fehlerakkumulation kommt. Ein Nachteil dieser Methode ist jedoch, dass die komplexen Beziehungen zwischen den vorhergesagten Werten ( $\hat{y}_{t+1}, \dots, \hat{y}_{t+h}$ ) nicht modelliert werden, weil diese Werte von jeweils

verschiedenen Modellen vorhergesagt werden. Außerdem ist diese Methode deutlich ineffizienter und rechenintensiver als andere Methoden, da  $h$  verschiedene Modelle trainiert werden müssen. [33]

**MIMO (Multiple Input Multiple Output) Methode:** Die MIMO Methode ermöglicht, im Gegensatz zur rekursiven und direkten Methode, die Ausgabe von mehreren Werten auf einmal, wodurch bei längerfristigen Vorhersagen auch komplexe Zusammenhänge zwischen den zukünftigen Werten modelliert werden können [33]. Es wird ein einziges Modell trainiert, das  $h$  Ausgaben hat.

$$(y_{t+1}, \dots, y_{t+h}) = f(y_{t-n}, \dots, y_t) + e \quad (29)$$

mit der Vektorfunktion  $f : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^h$  und dem Rauschvektor  $e \in \mathbb{R}^h$ . Ein Nachteil dieser Methode ist, dass der Horizont  $h$  im Nachhinein nicht mehr geändert werden kann ohne ein komplett neues Modell zu trainieren. [33]

## 2.5 Model Predictive Control (MPC)

MPC ist ein modernes Regelungsverfahren, das die prädiktive Modellierung nutzt, um ein System oder einen Prozess möglichst optimal zu regeln. Das Ziel ist es, die Reaktion eines Systems auf verschiedene Stellgrößen  $u$  über einen bestimmten Zeithorizont in Echtzeit vorherzusagen und die beste Regelstrategie unter Minimierung einer Kostenfunktion und bestimmter Nebenbedingungen zu finden. Eine Regelstrategie ist dabei eine Kombination von Stellgrößen  $u$  zu diskreten Zeitpunkten über einen bestimmten Regelhorizont  $m$  ( $u_t, u_{t+1}, \dots, u_{t+m}$ ). MPC ermöglicht die autonome Regelung komplexer Prozesse ohne Eingriff von Experten über einen längeren Zeitraum und ist flexibel, was die Anwendung auf verschiedene Systeme mit unterschiedlichen Eigenschaften erlaubt. [3, 34]

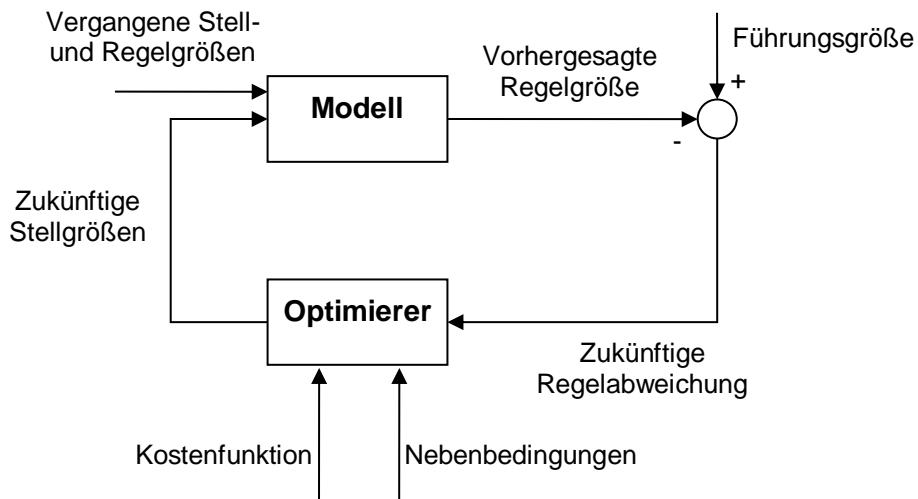


Abbildung 15: Grundlegende Struktur von MPC. Erstellt auf Basis von [3].

MPC benötigt ein geeignetes Prozessmodell, um auf der Basis vergangener Werte für Stell- und Regelgrößen und möglicher Regelstrategien das zukünftige Verhalten vom Prozess vorhersagen zu können. Die Effektivität der Regelung hängt demnach stark von der Güte dieses Modells ab. Mit der zukünftigen Regelabweichung (30) berechnet der Optimierer unter Berücksichtigung der Kostenfunktion und bestimmter Nebenbedingungen die optimale Regelstrategie. [3]

### 2.5.1 Kostenfunktion und Nebenbedingungen

Die möglichen Regelstrategien werden mit einer Kostenfunktion und wenn nötig mit zusätzlichen Nebenbedingungen bewertet. Das allgemeine Ziel besteht darin, dass die Regelstrategie ausgewählt werden soll, bei der sich die vorhergesagten Werte  $\hat{y}$  optimal an die Führungsgröße  $w$  annähern. Zur Berechnung der Gesamtkosten  $J$  werden die zukünftigen Regelabweichungen  $\hat{e}(t)$  zu jedem Zeitpunkt über den Prognosehorizont  $h$  benötigt (siehe Formel (1)). [3]

$$\hat{e}(t) = w(t) - \hat{y}(t) \quad (30)$$

$$J(h, m) = \sum_{j=1}^h \vartheta(j)[\hat{e}(t+j)]^2 + \sum_{j=1}^m \lambda(j)[\Delta u(t+j-1)]^2 \quad (31)$$

Der erste Term der Kostenfunktion  $J(h, m)$  ist eine gewichtete quadratische Summe der vorhergesagten Regelabweichungen mit dem jeweiligen Gewicht  $\vartheta(j)$ . Das Gewicht kann sich über den Prognosehorizont ändern, so dass Werte zu verschiedenen Zeitpunkten einen unterschiedlichen Einfluss auf die Kosten haben. Dafür kann zum Beispiel eine Exponentialfunktion verwendet werden. [3]

$$\vartheta(j) = \alpha^{h-j} \text{ mit } j \in [1, \dots, h] \quad (32)$$

Ist  $0 < \alpha < 1$ , so werden vorhergesagte Fehler die weiter in der Zukunft liegen höher gewichtet als die ersten, was zu einer sanfteren Regelung mit geringerem Aufwand führt. Bei  $\alpha > 1$  werden die ersten Fehler höher gewichtet, was zu einer strikteren Regelung führt. [3]

Der zweite Term von  $J(h, m)$  bezieht sich auf den Regelaufwand der bei einer Änderung der Stellgröße entsteht, dies könnte zum Beispiel Energiekosten umfassen, die möglichst minimal gehalten werden sollen. Sollte der Regelaufwand für die Problemstellung nicht bedeutsam sein, kann der zweite Term vernachlässigt werden. [3]

Neben der Kostenfunktion werden in der Praxis auch bestimmte Nebenbedingungen berücksichtigt. Der Aktor hat zum Beispiel nur einen begrenzten Aktionsraum und eine maximale Anstiegsgeschwindigkeit, die bei den verschiedenen Regelstrategien berücksichtigt werden müssen. Ein Ventil ist zum Beispiel durch die Stellung vollständig geöffnet oder geschlossen, dessen Öffnungs- und Schließgeschwindigkeit begrenzt. Nebenbedingungen können auch durch Sicherheits- oder Umweltauflagen entstehen, bei denen Prozessparameter wie Temperatur oder Druck beschränkt werden müssen. In den meisten Fällen wird eine untere und obere Grenze von  $u$  (33), eine Begrenzung der Anstiegsrate von  $u$  (34) und eine Beschränkung von  $y$  (35) berücksichtigt. [3]

$$u_{min} \leq u(t) \leq u_{max} \quad (33)$$

$$\Delta u_{min} \leq u(t) - u(t-1) \leq \Delta u_{max} \quad (34)$$

$$y_{min} \leq y(t) \leq y_{max} \quad (35)$$

## 2.5.2 Optimale Regelstrategie

Die optimale Regelstrategie ist die Strategie mit den geringsten Gesamtkosten, die auch alle Nebenbedingungen erfüllt. Um diese zu identifizieren, müssen zunächst alle möglichen Regelstrategien gebildet werden. Dazu werden alle möglichen Kombinationen der Stellgrößen  $u$  zu den diskreten Zeitpunkten über einen Regelhorizont  $m$  ( $u_t, u_{t+1}, \dots, u_{t+m}$ ) gebildet. Für jede Regelstrategie werden dann mit Hilfe des Modells die zukünftigen Werte der Regelgröße über den Prognosehorizont vorhergesagt. Mit diesen Vorhersagen werden dann die zugehörigen Kosten bestimmt und aus allen Strategien diejenige ausgewählt, die die geringsten Kosten verursacht und zusätzliche Nebenbedingungen erfüllt. Von der optimalen Regelstrategie zum Zeitpunkt  $t$  wird dann immer nur das erste Stellsignal angelegt. Die Berechnung der optimalen Regelstrategie wird für jeden Zeitschritt wiederholt. [34]

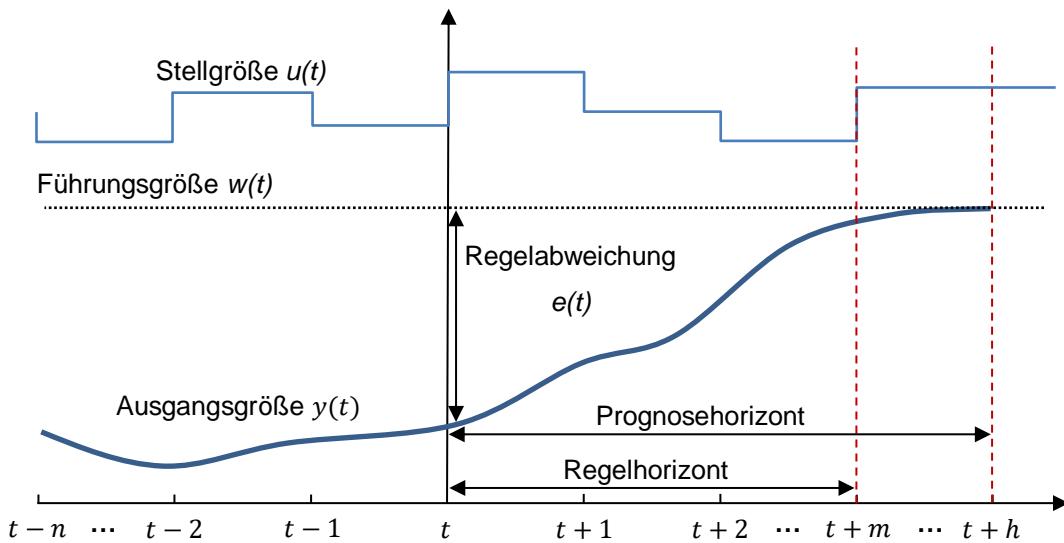


Abbildung 16: Vorhersage einer Regelstrategie über ein gleitendes Zeitfenster bei MPC

Abbildung 16 zeigt ein Beispiel für die Vorhersage einer Regelstrategie. Es wird ein gleitendes Zeitfenster verwendet, nur dass hier zusätzlich zum Systemzustand die Stellgröße zur Modellierung herangezogen wird. Neben dem eigentlichen Prognosehorizont  $h$  wird ein Regelhorizont  $m$  eingeführt, über dem alle möglichen Regelstrategien gebildet werden. Die Bewertung der Strategien erfolgt dann über den eigentlichen Prognosehorizont. Da das System auf eine Stellgröße zeitverzögert reagiert, sollte der Prognosehorizont größer als der Regelhorizont sein, um die zeitlichen Abhängigkeiten zu berücksichtigen. Außerdem kann der Regelhorizont kleiner gewählt werden, um den Rechenaufwand zu reduzieren.

### 3 Implementierung

In diesem Kapitel wird beschrieben, wie die automatische Prozessregelung mit Hilfe von KI und prädiktiver Modellierung in Python implementiert wird. Zusätzlich wird ein virtuelles Stellvertretersystem implementiert, um die Effizienz der KI-Regelung zu testen.

Das Ziel der Regel-KI ist es, die Regelgröße des Systems mithilfe von MPC an einen vorgegebenen Referenzwert anzunähern. Als prädiktives Modell wird in dieser Arbeit ein KNN verwendet, das durch ein gleitendes Zeitfenster den Zusammenhang zwischen vergangenen Stell- und Regelgrößen lernen soll und Vorhersagen über zukünftige Systemzustände treffen kann. Im Verlauf des virtuellen Prozesses wird das KNN kontinuierlich mit neuen Daten trainiert, um die Vorhersagegenauigkeit und damit die Effektivität der Regelung über die Zeit zu verbessern. Dazu wurde in der ersten Version das am iPAT entwickelte KI-Framework „HyREN“ (Hybrid Regression Evolutionary Network) eingesetzt, das einen genetischen Algorithmus zum Training eines KNN verwendet [18]. Diese Methode benötigt jedoch mehr Rechenleistung als das klassische überwachte Lernen mit dem Backpropagation-Algorithmus, wodurch der gesamte Trainingsprozess deutlich länger dauert. Bei der Simulation eines virtuellen Systems ist eine längere Trainingsdauer nicht unbedingt von Nachteil, da der Prozess in diesem Fall nicht weiterläuft, bis das Training der Regel-KI abgeschlossen ist. Bei der Übertragung der Regel-KI auf einen realen Prozess, muss jedoch berücksichtigt werden, dass der Prozess während des Trainings weiterläuft. Es ist von entscheidender Bedeutung, die Trainingszeit der Regel-KI so kurz wie möglich zu halten, um eine Regelung in Echtzeit zu ermöglichen. Es wurde festgestellt, dass mit dem KI-Framework HyREN eine Trainingsiteration mehrere Minuten dauert. Im Vergleich dazu benötigt das überwachte Lernen mit dem Backpropagation Algorithmus weniger als eine Sekunde, abhängig von der Wahl der Hyperparameter wie „batch\_size“ und „epochs“ und der verfügbaren Rechenleistung. Aus diesem Grund wird HyREN in dieser Arbeit nicht verwendet.

Das Hauptprogramm, das die Schnittstelle zwischen der Regel-KI und dem virtuellen System bildet, wird in der Datei „main.py“ ausgeführt. In dieser Datei werden sowohl die Regel-KI als auch das virtuelle System konfiguriert. Nach jedem Zeitschritt wird der aktuelle Systemzustand von „main.py“ an die Regel-KI übergeben. In regelmäßigen Zeitabständen berechnet die Regel-KI mittels MPC die optimale Regelstrategie und übergibt diese an „main.py“. Dort wird die erste Stellgröße der optimalen Regelstrategie auf das virtuelle System angewendet. Das virtuelle System berechnet dann den nächsten Systemzustand in Abhängigkeit von der angelegten Stellgröße und der Störgröße und gibt diesen an „main.py“ zurück. Die Regel-KI wird zudem in regelmäßigen Abständen mit den neuen Daten trainiert. Der beschriebene Zyklus wird solange wiederholt, bis eine vorgegebene Schrittgrenze erreicht ist. Der Ablauf des Algorithmus ist in Abbildung 17 dargestellt.

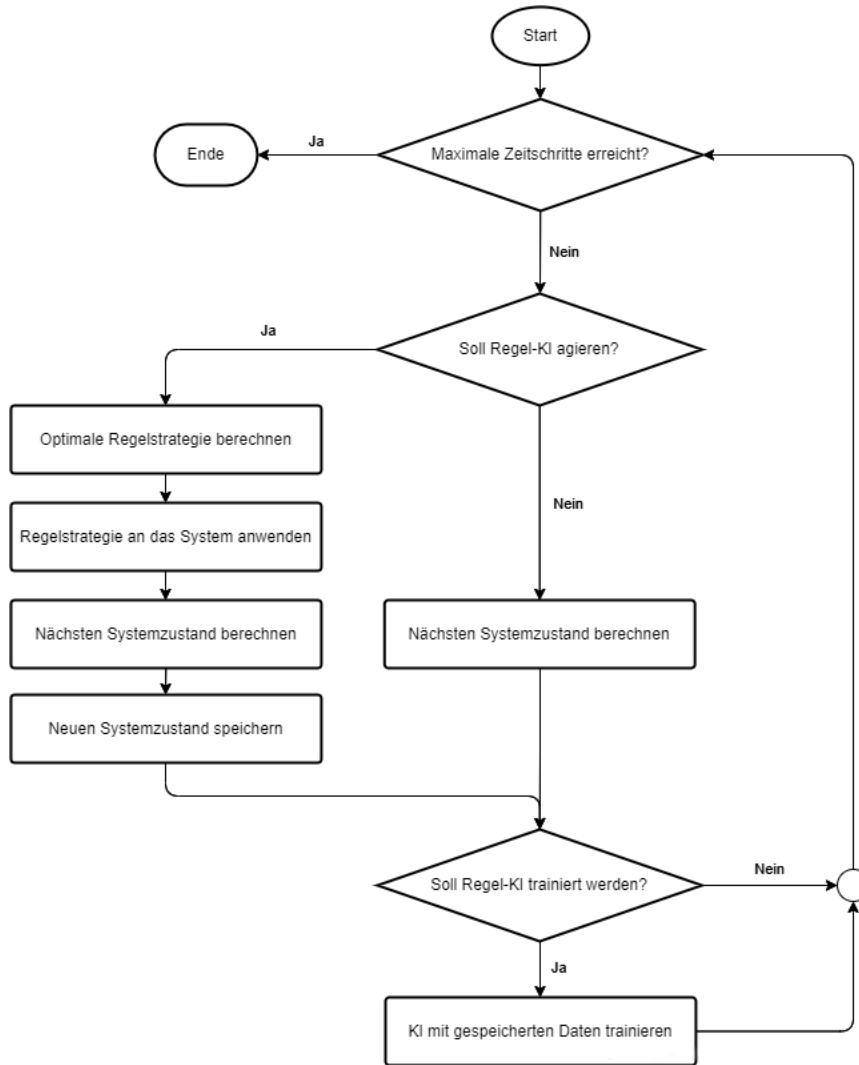


Abbildung 17: Ablauf des Hauptprogramms

### 3.1 Umformung einer Zeitreihe in überwachtes Lernen

Damit die Regel-KI trainiert werden kann, muss die bekannte Zeitreihe zunächst in klassisches überwachtes Lernen überführt werden. Dazu wird die Methode des gleitenden Zeitfensters verwendet, bei der die Werte vorhergehender Zeitschritte zur Vorhersage der Werte nachfolgender Zeitschritte verwendet werden. Ein Beispiel für eine univariate Zeitreihe, beziehungsweise eine Zeitreihe, in der nur eine Variable betrachtet wird, ist in Tabelle 6 dargestellt. [35]

Tabelle 6: Beispiel einer univariaten Zeitreihe

Zeit	Wert
1	64
2	21
3	45
4	89

Tabelle 7: Beispiel einer Umwandlung einer univariaten Zeitreihe in überwachtes Lernen

X	y
?	64
64	21
21	45
45	89
89	?

Die Umwandlung in überwachtes Lernen ist in Tabelle 7 dargestellt. X sind die Werte, die als Eingaben für das KNN verwendet werden und y die Werte, die als Ausgaben für das KNN verwendet werden. Die erste und die letzte Zeile können nicht verwendet werden, da jeweils ein Wert fehlt. Bei der Umwandlung wird die Spalte mit den Werten kopiert und um eine Zeile verschoben. [35]

In Python kann diese Umwandlung auf einen „DataFrame“ mit der Funktion „.shift(x)“ angewendet werden, die alle Spalten des „DataFrames“ um x Zeilen verschiebt [36]. Für eine Vorhersage über mehrere Zeitschritte muss die Zeitreihe für jeden Schritt kopiert und entsprechend verschoben werden. Die Umformung in Trainingsdaten erfolgt in diesem Programm mit der Funktion „convert\_input\_data\_training()“ aus „utils.py“. Die Funktion erlaubt es, die Anzahl der vergangenen Werte  $n$ , den

Regelhorizont  $m$  und den Prognosehorizont  $h$  zu variieren und so unterschiedliche Zeitfenster zu erzeugen. Für die Umformung in Trainingsdaten werden für ein Zeitfenster  $q$  Zeitschritte benötigt, mit

$$q = n + h + 1 \quad (36)$$

**Tabelle 8: Multivariate Zeitreihe mit Stell- und Regelgröße**

Zeit	Stellgröße	Regelgröße
0	$u_0$	$y_1$
1	$u_1$	$y_1$
...		
$t - n$	$u_{t-n}$	$y_{t-n}$
...		
$t - 1$	$u_{t-1}$	$y_{t-1}$
$t$	$u_t$	$y_t$
$t + 1$	$u_{t+1}$	$y_{t+1}$
...		
$t + m$	$u_{t+m}$	$y_{t+m}$
...		
$t + h$	$u_{t+h}$	$y_{t+h}$
...		
$i$	$u_i$	$y_i$

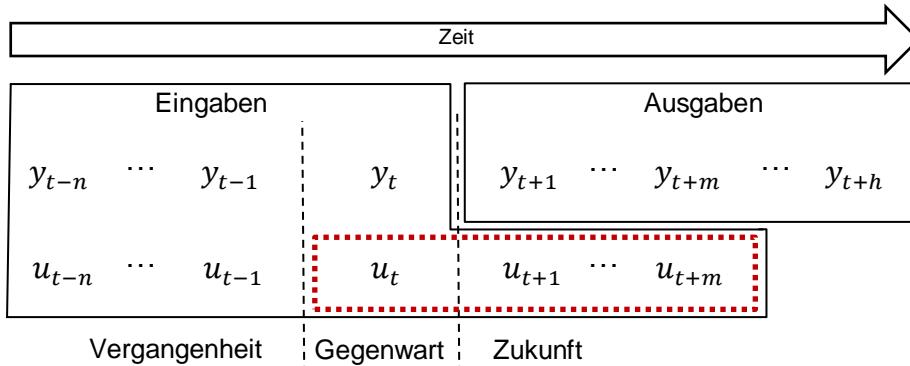


Abbildung 18: Struktur eines Zeitfensters zum Zeitpunkt  $t$  mit Stell- und Regelgröße

Für eine Vorhersage der zukünftigen Regelgröße ( $\hat{y}_{t+1}, \hat{y}_{t+2}, \dots, \hat{y}_{t+h}$ ) werden sowohl die Regelgröße als auch die Stellgröße der vergangenen Zeitpunkte benötigt. Es liegt also eine multivariate Zeitreihe vor. Tabelle 8 zeigt die aufgezeichnete Zeitreihe des Systems, bei der für jeden Zeitschritt die Stellgröße und die Regelgröße aufgezeichnet wurde. Dabei ist  $i$  der letzte aufgezeichnete Zeitpunkt. Damit mindestens ein gültiges Zeitfenster zum Training existiert, muss  $i \geq q$  sein, da sonst nicht genug Werte für ein Zeitfenster vorhanden sind. Für einen beliebigen Zeitpunkt  $t \in [n, i - h]$  mit  $i \geq q$  kann ein Zeitfenster gebildet werden, das  $n$  vergangene Zeitpunkte benutzt, um  $h$  zukünftige Zeitpunkte vorherzusagen, wobei alle Werte bekannt sind. Die Umwandlung der Zeitreihe in Tabelle 8 in ein Zeitfenster zum Zeitpunkt  $t$ , mit den jeweiligen Eingaben und Ausgaben für das KNN, ist in Abbildung 18 dargestellt. Als Eingaben werden die  $n$  letzten Werte der Stell- und Regelgröße, die aktuelle Stell- und Regelgröße zum Zeitpunkt  $t$  und die nächsten  $m$  Stellgrößen verwendet. Der rot gepunktete Kasten stellt eine Matrix aller möglichen Regelstrategien dar, die nur für die Vorhersage für  $t = i$  verwendet wird. Beim Erstellen von Trainingsdaten spielt sie keine Rolle, da die Werte  $(u_t, u_{t+1}, \dots, u_{t+m})$  in diesem Fall bekannt sind.

### 3.2 Regel-KI

Der Hauptbestandteil der Regel-KI befindet sich in der Klasse „Agent“ in der Datei „control\_ai.py“. In dieser Klasse ist das KNN gespeichert, das als prädiktives Modell verwendet und mit der statischen Funktion „build\_network()“ erstellt wird. Das KNN besteht aus einer Eingabeschicht, einer Ausgabeschicht und zwei verborgenen Schichten, deren Anzahl an Neuronen jeweils durch die Eingabeparameter der Funktion angepasst werden kann.

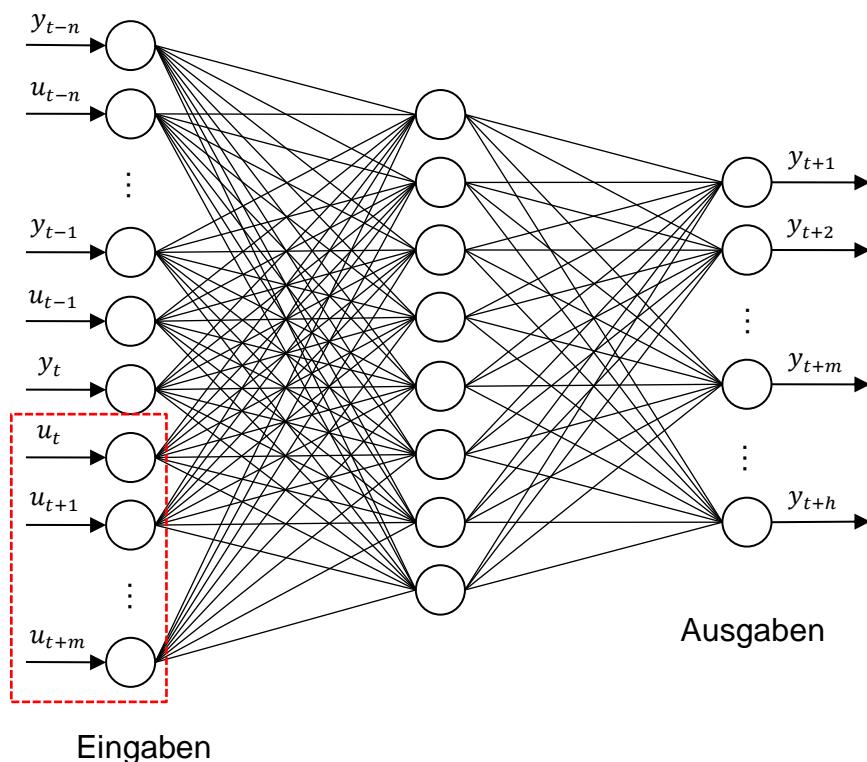
Für die Erstellung eines KNN wird das Modul „Keras“ verwendet. Keras ist eine leistungsfähige Bibliothek, die auf der von Google entwickelten maschinellen Lernplattform „TensorFlow“ basiert und die Erstellung komplexer Modelle mit minimalem Aufwand ermöglicht. Aufgrund der einfachen Handhabung ist Keras eine der beliebtesten Bibliotheken für Deep Learning. [37]

Für die Speicherung der Zeitreihe wird der Datentyp „deque“ aus dem Modul „collections“ verwendet. Dieser Datentyp ähnelt einer Liste, jedoch kann hierbei eine maximale Größe definiert werden kann [38]. Wird die maximale Größe nach dem Einfügen eines neuen Elements überschritten, wird das älteste Element automatisch gelöscht [38]. Diese Eigenschaft ist in diesem Zusammenhang von Vorteil, da bei einem kontinuierlichen Prozess die Datenmenge über einen längeren Zeitraum sehr hoch sein kann. Durch die Verwendung des Datentyps „deque“ wird vermieden, dass nach längerer Laufzeit der Regel-KI der Speicherplatz nicht mehr ausreicht.

In dieser Arbeit wird das in Kapitel 2.4.1 beschriebene MIMO-Verfahren verwendet, um Vorhersagen über mehrere Zeitschritte zu treffen, da dieses Verfahren bei einer hohen Anzahl von Vorhersagen effizient ist. Im Vergleich dazu wurde zunächst das rekursive Verfahren eingesetzt, das jedoch bei einer hohen Anzahl von Vorhersagen deutlich mehr Zeit benötigt, da jeder Zeitschritt jeder Regelstrategie einzeln vorhergesagt werden muss. Für eine Regelung in Echtzeit ist die rekursive Strategie daher nicht geeignet. Bei Verwendung des MIMO-Verfahrens hat das KNN so viele Ausgabe-Neuronen, wie Zeitschritte vorhergesagt werden müssen. Die Struktur der

Eingabe- und Ausgabeschicht eines KNN, das die nächsten  $h$  Zeitpunkte unter Verwendung des umgeformten Zeitfensters aus Abbildung 18 vorhersagt, ist in Abbildung 19 dargestellt. Die Anzahl der Eingabe-Neuronen wird mit (37) berechnet.

$$Q_{Input} = (n + 1) * 2 + m \quad (37)$$



**Abbildung 19: Struktur der Eingabe- und Ausgabeschicht der Regel-KI**

Der rote Kasten stellt auch in dieser Abbildung die Matrix der möglichen Regelstrategien dar. Für die Vorhersage sind alle Eingabewerte oberhalb des Kastens bekannt, da es sich um vergangene Werte für die Stell- und Regelgröße handelt. Wie in Abbildung 15 dargestellt, sagt das Modell mit den vergangenen Werten für Stell- und Regelgröße und den zukünftigen Stellgrößen aus jeder möglichen Regelstrategie den Verlauf der zukünftigen Regelgröße voraus.

Zu Beginn der Prozessregelung befindet sich die Regel-KI in einer Explorationsphase, da ihr keine Daten zum Trainieren zur Verfügung stehen. Ähnlich wie beim Reinforcement Learning besteht hier ein Konflikt zwischen Exploration und Exploitation (siehe Kapitel 2.2.2). In der Anfangsphase sollte die Regel-KI möglichst viel Erfahrung sammeln, um ein solides Verständnis des Prozesses zu erlangen. Im Laufe des Prozesses verbessert sich die Vorhersagegenauigkeit des verwendeten KNN mit zunehmender Datenmenge, so dass die Regel-KI zunehmend die bestmögliche Regelstrategie ausnutzen sollte.

Um Erkundung und Ausnutzung auszugleichen, wird der Epsilon-Greedy Algorithmus verwendet. Der Wert von Epsilon ist dabei eine Zahl zwischen null und eins, die die Wahrscheinlichkeit für die Erkundung durch eine zufällige Aktion beschreibt. Um zu bestimmen, ob die Regel-KI erkunden oder ausnutzen soll, wird eine Zufallszahl  $p$  zwischen null und eins generiert. Wenn  $p < \varepsilon$  ist, wird eine zufällige Aktion ausgeführt, andernfalls wird die berechnete optimale Regelstrategie ausgeführt. Ein Wert von  $\varepsilon = 1$  bedeutet, dass zu 100% eine zufällige Aktion ausgeführt wird, während  $\varepsilon = 0$  bedeutet, dass zu 100% die berechnete optimale Regelstrategie ausgeführt wird. Zu Beginn des Prozesses ist  $\varepsilon = 1$ , um das System zu erkunden. Im Laufe des Prozesses soll  $\varepsilon$  kontinuierlich reduziert werden, um die Wahrscheinlichkeit zufälliger Aktionen zu verringern. Dazu wird nach jeder Trainingsiteration des KNN der Wert für  $\varepsilon$  mit einer Konstanten  $\varepsilon_{dec} \in (0,1)$  multipliziert, was zu einer Reduktion von  $\varepsilon$  über die Zeit führt. Zusätzlich kann eine untere Schranke  $\varepsilon_{min}$  für  $\varepsilon$  festgelegt werden, so dass die Wahrscheinlichkeit für die Erkundung nie null wird. [39]

Die Regel-KI agiert in festen Zeitintervallen von  $t_{act}$  Millisekunden und wird in festen Zeitintervallen von  $t_{train}$  Millisekunden trainiert. In einem virtuellen System sollten  $t_{act}$  und  $t_{train}$  ein Vielfaches der Zeitschrittgröße  $\Delta t$  der Simulation sein, um ein vorhersehbares und konsistentes Verhalten der Regel-KI zu gewährleisten. Die Größe des Zeitfensters  $t_{window}$  kann mit der Gleichung (38) und die Menge neuer Daten pro Trainingsiteration  $Q_{train}$  mit der Gleichung (39) berechnet werden.

$$t_{window} = (n + h) * t_{act} \quad (38)$$

$$Q_{train} = \frac{t_{train} - t_{window}}{t_{act}} \quad (39)$$

### 3.3 Berechnung der optimalen Regelstrategie

Zur Bestimmung der optimalen Regelstrategie ist es zunächst erforderlich, den Verlauf der zukünftigen Regelgröße in Abhängigkeit aller möglichen Regelstrategien vorauszusagen. Dazu muss zuerst eine Matrix  $M$  erstellt werden, die alle möglichen Regelstrategien über einen bestimmten Regelhorizont enthält. Zur Berechnung der Matrix wird das kartesische Produkt verwendet, mit dem alle Kombinationen von zwei oder mehr Mengen berechnet werden können. Angenommen es liegen die Mengen  $A = \{a, b, c\}$  und  $B = \{1, 2\}$  vor, so ergibt das kartesische Produkt von A und B die Menge  $\{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$ . [40]

Mit der Menge  $U$  der möglichen diskreten Werte für die Stellgröße und dem Regelhorizont  $m$  lassen sich alle möglichen Regelstrategien  $M$  berechnen, mit

$$M = U^{m+1} \quad (40)$$

Es entsteht eine  $|U|^{m+1} \times (m + 1)$  Matrix. Für  $m = 2$  ergibt sich  $M = U \times U \times U$ . Es ist zu beachten, dass der Regelhorizont in dieser Arbeit nur zukünftige Stellgrößen  $(u_{t+1}, \dots, u_{t+m})$  berücksichtigt. Die aktuelle Stellgröße  $u_t$  wird immer betrachtet. Für den Regelhorizont  $m = 0$  wird demnach nur die aktuelle Stellgröße  $u_t$  zur Vorhersage verwendet. Die Anzahl der möglichen Regelstrategien nimmt mit steigendem  $m$  exponentiell zu. Enthält  $U$  beispielsweise zehn mögliche Werte für die Stellgröße, so ergeben sich bei  $m = 4$  100.000 verschiedene Regelstrategien.

Für die Auflistung aller möglichen Regelstrategien müssen die möglichen diskreten Werte für die Stellgröße beim Start in Form einer Liste an die Regel-KI übergeben werden. Anschließend werden mit der Funktion „compute\_all\_possible\_strategies()“ aus der Datei „utils.py“ alle möglichen Regelstrategien in einem „numpy.array“

aufgelistet. Die Funktion erhält als Parameter den Regelhorizont und eine Liste mit Werten für die Stellgröße.

Daten für die Vorhersage werden mit der Funktion „`_get_prediction_data()`“ in die richtige Form gebracht. Die Funktion erhält als Parameter den aktuellen Wert der Regelgröße, da dieser noch nicht im Speicher vorhanden ist. In dieser Funktion werden zunächst die Daten der  $n$  letzten Zeitpunkte und der aktuelle Wert der Regelgröße in ein „`numpy.array`“ umgewandelt  $(y_{t-n}, u_{t-n}, \dots, y_{t-1}, u_{t-1}, y_t)$ . Die Zeile der Liste wird dann so oft kopiert, wie es Regelstrategien gibt, damit das gerade erzeugte Array mit dem Array der Regelstrategien zusammengefügt werden kann, so dass jede Zeile die Eingabestruktur aus Abbildung 18 hat. Schließlich werden die vorhergesagten Regelgrößen  $(\hat{y}_{t+1}, \dots, \hat{y}_{t+h})$  mit der Kostenfunktion (31) bewertet. In dieser Arbeit, wird nur die Nebenbedingung (34) betrachtet, mit  $\Delta u_{min} = 0$ . In der Regel-KI kann ein „`du_max`“ definiert werden, um große Schwankungen der Stellgröße zu vermeiden. Regelstrategien, die dieses Kriterium nicht erfüllen, werden für die Vorhersage herausgefiltert. Dies hat den Vorteil, dass die Anzahl der Vorhersagen nicht zu groß wird. In den meisten Fällen wird die Regelstrategie mit den geringsten Kosten angewendet. Um zu vermeiden, dass die Regel-KI in ein lokales Minimum fällt, wird manchmal eine zufällige Regelstrategie aus den besten  $k$  ausgewählt.

### 3.4 Virtuelles System

In diesem Kapitel wird das virtuelle Stellvertretersystem implementiert, das dazu dient, die Wirksamkeit der Regel-KI zu testen. Da die Regel-KI in einer späteren Arbeit auf einen Bioreaktor zur pH-Regelung angewendet werden soll, wird zunächst ein Stellvertretersystem verwendet, das den pH-Wert in einem Bioreaktor grob modellieren soll. Dadurch kann die Übertragbarkeit der Regel-KI auf ein reales System vorab effizient getestet werden.

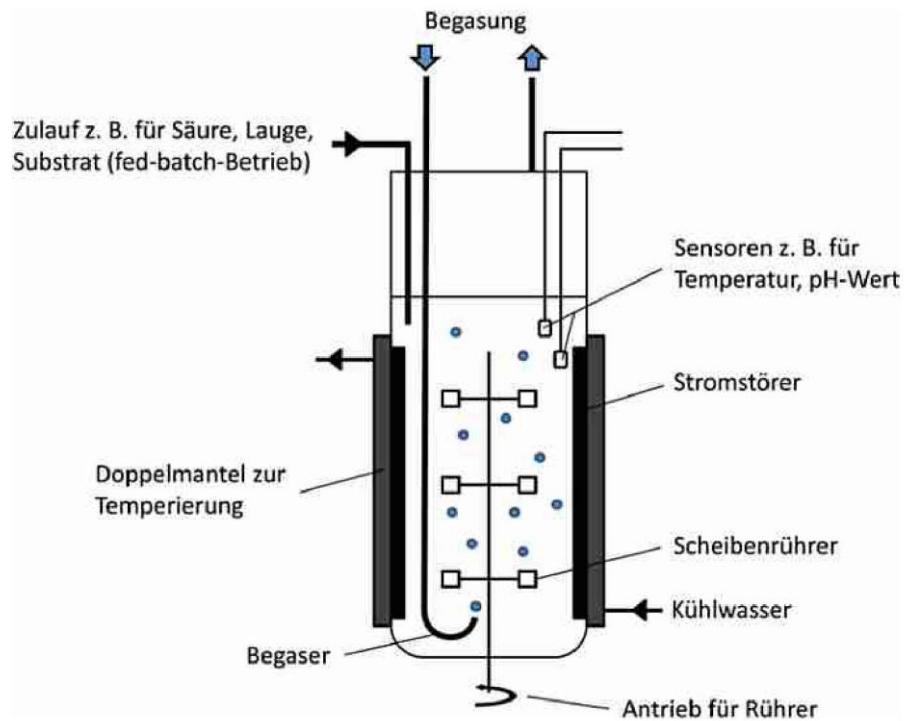


Abbildung 20: Schematische Darstellung eines Bioreaktors in Form eines Rührkessels [41]

Ziel einer zukünftigen Arbeit wird es sein, die Wirksamkeit der Regel-KI bei der pH-Wert-Regelung in einem Bioreaktor zu testen. In diesem Fall wird die Stellgröße durch die Ventilstellung repräsentiert, die die Zuflussrate einer Säure oder Lauge beeinflusst und somit den pH-Wert verändert. Die Regelgröße ist hier der pH-Wert, der über Sensoren im Kessel gemessen wird. In dieser Arbeit wird ein Proxysystem verwendet, das einen pH-Wert mit Gleichung (41) näherungsweise modellieren soll. Das Proxysystem ermöglicht eine schnelle Implementierung und Erprobung verschiedener Regelungsverfahren.

$$\dot{y}(t) + k_1(t)|y(t) - y_s| + k_2(t) = k_s u(t) + z(t) \quad (41)$$

Hier ist  $y(t)$  die Regelgröße des Systems, also der pH-Wert, der durch die Stellgröße  $u(t)$  beeinflusst wird.  $k_1(t)$  und  $k_2(t)$  sind interne Zustandsgrößen des Systems, die die Regelgröße zusätzlich beeinflussen und sich mit der Zeit ändern können.  $y_s$  beschreibt den stabilsten Wert des Systems, bei dem  $k_1$  keinen Einfluss hat und die

Steigung von  $y$  minimal ist.  $z(t)$  beschreibt die externen Störgrößen und teilt sich auf in ein zufälliges Rauschen  $z_r$ , einen möglichen Störimpuls  $z_i$  und ein kontinuierliches, dynamisches Störverhalten  $z_d$ .

$$z(t) = z_r(t) + z_i(t) + z_d(t) \quad (42)$$

Das zufällige Rauschen wird durch eine Zufallszahl mithilfe der „numpy“ Funktion „random.normal()“ erzeugt und soll unter anderem die Messunsicherheit modellieren. Die Zufallszahl ist dabei normalverteilt mit dem Erwartungswert  $\mu_r = 0$  und einer vorgegebenen Standardabweichung  $\sigma_r$ . Der Wert für die Standardabweichung kann der Simulation vor dem Start als Parameter übergeben werden. Der Zufallswert  $z_r(t)$  wird in jedem Zeitschritt der Simulation neu erzeugt. Der Störimpuls  $z_i(t)$  kann zu beliebigen Zeitpunkten erfolgen, um die Stabilität der Regelung zu testen. Die dynamische Störgröße  $z_d(t)$  kann zu Beginn der Simulation als Funktion der Zeit vorgegeben werden.

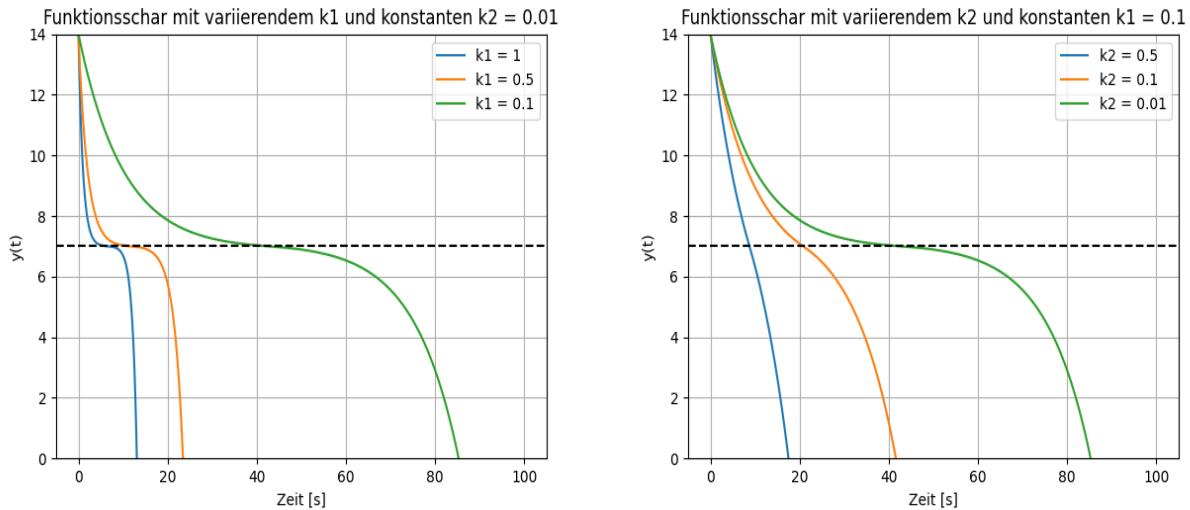


Abbildung 21: Einfluss von  $k_1$  und  $k_2$  auf das System mit  $u(t) = 0$  und  $z(t) = 0$

Das System wird durch  $y \in [0,14]$  begrenzt, um einen pH-Wert näherungsweise zu modellieren. Abbildung 21 zeigt den Einfluss von  $k_1$  und  $k_2$  auf das System mit  $y_s = 7$ . Es ist zu erkennen, dass  $k_1$  die Steigung in Abhängigkeit vom Abstand zum Wert  $y_s$

beeinflusst. Je größer  $k_1$  ist, desto schneller wird das System instabil, wenn es geringfügig von  $y_s$  abweicht. Der Wert von  $k_2$  beschreibt eine konstante Abnahme des Systems und beeinflusst wie stabil das System ist im Bereich von  $y_s$  ist. Die Werte von  $k_1$  und  $k_2$  enthalten alle physikalischen Eigenschaften, die den pH-Wert beeinflussen. Für  $k_1$  kann dies unter anderem die Reaktionskinetik und für  $k_2$  zum Beispiel die Diffusionsrate sein. Auch andere physikalische Eigenschaften wie Temperatur und Druck können  $k_1$  und  $k_2$  beeinflussen. Die Zusammenfassung in  $k_1$  und  $k_2$  dient der Vereinfachung des Proxysystems.

Das virtuelle System wird in der Datei „simulation.py“ simuliert. Diese Datei enthält die Klasse „Simulation“, in der das virtuelle System simuliert wird. Die Simulation kann mit der Klasse „SimConfig“ konfiguriert werden, in der die Systemparameter eingestellt werden können. Mit der Funktion „step()“ wird der Zustand des Systems zum nächsten Zeitschritt in Abhängigkeit der Stellgröße und des Störimpulses berechnet. Die Differentialgleichung (41) wird verwendet, um die Änderung  $\Delta y$  mit der Funktion „\_calculate\_change()“ zu berechnen, die dann auf die Regelgröße  $y$  addiert wird. Anschließend wird der alte Systemzustand in einem „dict“-Objekt gespeichert und in eine Liste eingefügt und die Zeit  $t$  um  $\Delta t$  erhöht. Am Ende wird noch sichergestellt, dass  $y$  innerhalb der vorgegebenen Grenzen liegt. Die Funktion gibt den gespeicherten Systemzustand und den neu berechneten Wert für  $y$  zurück.

### 3.5 Datenauswertung und Visualisierung

Um die Leistung der Regel-KI zu bewerten, werden für jede Regelstrategie die tatsächlichen Werte, die das System annehmen würde, mit der Differentialgleichung (41) berechnet und mit den vorhergesagten Werten verglichen. Die tatsächlichen Werte ( $y_{t+1}, \dots, y_{t+h}$ ) für jede Regelstrategie können mit der Funktion „calculate\_strategies()“ aus „simulation.py“ berechnet werden. Für die Berechnung der Differentialgleichung wird die Funktion „odeint()“ aus dem Modul „scipy.integrate“ verwendet. Es ist zu beachten, dass die Berechnung der tatsächlichen Werte nur zur

Überprüfung der Genauigkeit der Vorhersage dient und diese nicht zum Training der Regel-KI verwendet werden.

In der Klasse „Data“ aus der Datei „export.py“ wird der Fehler mit den Methoden MSE, MAPE und MAE berechnet (Abschnitt 2.3.2). Es wird der Gesamtfehler über alle Regelstrategien und Zeitpunkte berechnet. Zusätzlich wird der Gesamtfehler über alle Regelstrategien zu jedem einzelnen Zeitpunkt ( $t + 1, \dots, t + h$ ) berechnet. Die Ergebnisse werden mit den Funktionen von „export.py“ graphisch visualisiert und als „.png“-Dateien in einem Ordner gespeichert. Die Konfigurationseinstellungen jeder Simulation werden im Ordner „data“ des Hauptprojekts gespeichert. Dort werden auch die Graphen zur Visualisierung der Vorhersagegenauigkeit des KNN in regelmäßigen Abständen im Ordner „Graphs“ gespeichert. Am Ende der Simulation wird der Verlauf der Simulation und der zeitliche Verlauf des Fehlers über die Zeit im Ordner „summary“ gespeichert.

## 4 Ergebnisse und Diskussion

Dieses Kapitel befasst sich mit der Evaluierung der implementierten Regel-KI und den dabei aufgetretenen Problemen. Zunächst wird die Wirksamkeit der Regelung in einem statischen System untersucht, wobei der Einfluss der Größe des Zeitfensters, der Prozessparameter und der Störungen auf den Regelungserfolg untersucht wird. In allen Fällen liegt ein zufälliges, normalverteiltes Rauschen  $z_r$  vor, dessen Standardabweichung zu Beginn der Simulation festgelegt wird.

Anschließend wird der Regelungserfolg bei einem dynamischen System untersucht, indem sowohl die Führungsgröße als auch die Zustandsgrößen  $k_1$  und  $k_2$  über die Zeit variiert werden. Weiterhin wird die Genauigkeit des prädiktiven Modells untersucht, indem die vorhergesagten Werte mit den tatsächlichen Werten unter Berücksichtigung aller Regelstrategien verglichen werden.

Abschließend wird diskutiert, ob die Regel-KI auf einen realen Prozess übertragbar ist und für welche Prozesse diese Regelungsmethode ungeeignet ist.

### 4.1 Problem mit der ReLU Aktivierungsfunktion

Ein Problem, das bei der Auswertung auftrat, war, dass häufig ein oder mehrere Ausgabe-Neuronen des KNN mit der ReLU-Aktivierungsfunktion während der gesamten Laufzeit null blieben. Dies ist vermutlich auf das in Kapitel 2.2.3 im Abschnitt zur ReLU-Funktion erwähnte Problem der toten Neuronen zurückzuführen.

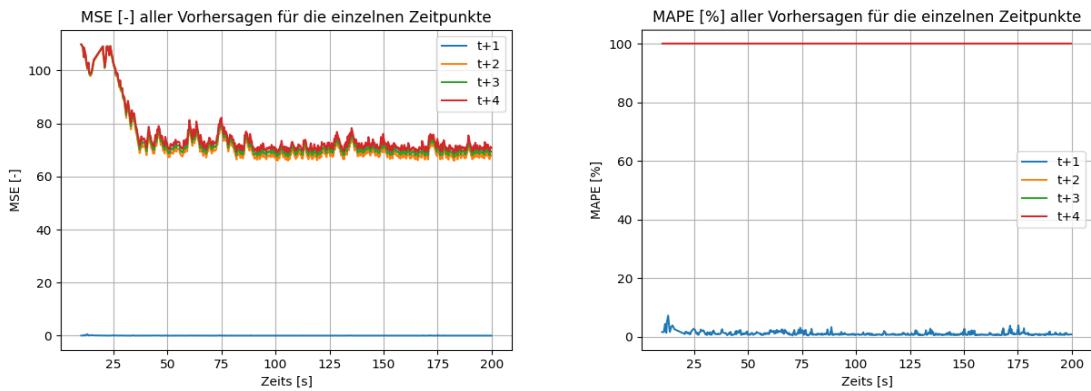
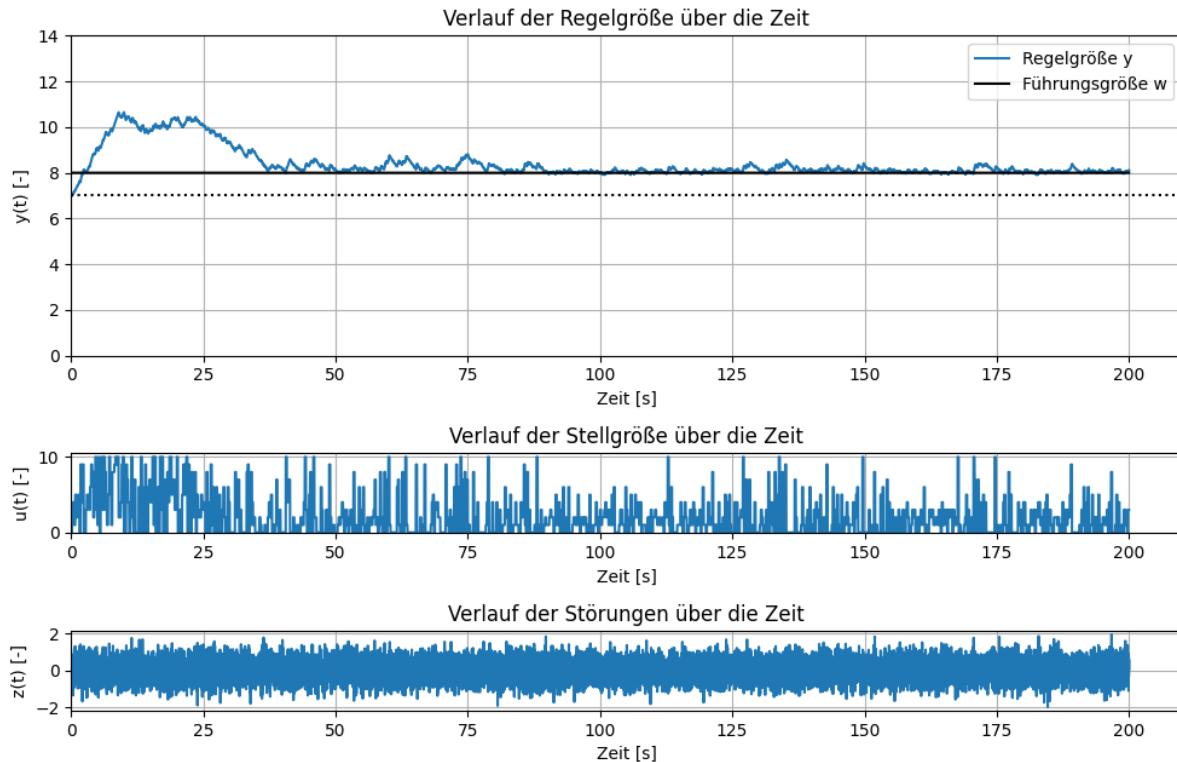


Abbildung 22: MSE und MAPE für die einzelnen Zeitpunkte, bei dem drei Ausgabe-Neuronen permanent null sind

Abbildung 22 zeigt den MSE und MAPE der einzelnen Zeitpunkte unter Verwendung der ReLU-Aktivierungsfunktion. Es ist zu erkennen, dass nur die Vorhersage für  $t + 1$  genau ist. Die restlichen drei Zeitpunkte haben einen MSE von etwa 70 und einen MAPE von 100%. Dies ist darauf zurückzuführen, dass die Ausgabe-Neuronen, die die Vorhersage für den zweiten, dritten und vierten Zeitschritt treffen, dauerhaft ausgeschaltet sind.



**Abbildung 23: Simulationsverlauf für eine Regel-KI bei der drei Ausgabe-Neuronen permanent null sind**

Trotz dieser fehlerhaften Vorhersage zeigt Abbildung 23 dass die Regelung erfolgreich ist. Dies ist darauf zurückzuführen, dass die Kosten für die fehlerhaft vorhergesagten Zeitschritte konstant bleiben, da sich weder die Ausgabe noch die Gewichtung der Zeitschritte ändert. Wenn die Kosten für einen Zeitschritt für alle Vorhersagen gleichbleiben, können sie praktisch vernachlässigt werden, da sie keinen Einfluss auf die Wahl der kostenminimalen Strategie haben. Eine erfolgreiche Regelung ist theoretisch möglich, solange mindestens ein Ausgabe-Neuron korrekte Vorhersagen treffen kann. Da dieses Problem jedoch vermieden werden soll, wird anstelle der ReLU-Funktion für jede Schicht des KNN die Leaky-ReLU-Aktivierungsfunktion mit einer Steigung von 0,1 im negativen Bereich verwendet. Dies behebt das Problem und führt zu besseren Vorhersagen.

## 4.2 Regelung eines statischen Systems

In diesem Kapitel wird die Effektivität der Regel-KI bei dem virtuellen System mit konstanten Zustandsgrößen  $k_1$  und  $k_2$  und einer konstanten Führungsgröße untersucht. Dazu wird der Einfluss der Fenstergröße, der Prozessparameter und der Störungen auf den Regelungserfolg untersucht. Die Simulationszeit wird auf 200 Sekunden festgelegt. Die Systemparameter werden wie folgt festgelegt:  $y_s = 7$ ,  $k_1 = 0,2$ ,  $k_2 = 0,05$ ,  $k_s = 0,15$  und  $U = \{0,1, \dots, 10\}$ . Das  $\Delta t$  der Simulation wird auf 10ms gesetzt. Die Gewichtung für die Kostenfunktion (32) wird mit  $\alpha = 0,7$  berechnet, spätere Werte werden folglich höher gewichtet als frühere Werte. Die Nebenbedingung (34) wird zunächst nicht berücksichtigt, wodurch die Stellgröße hohe Schwankungen aufweisen kann. Die Simulation beginnt bei  $y(t = 0) = 7$ . Für das Störverhalten wird zunächst nur das zufällige Rauschen mit  $\sigma_r = 0,5$  berücksichtigt. Der Einfluss von  $z_i$  und  $z_d$  wird in Kapitel 4.2.3 untersucht. Der Wert von  $\varepsilon_{dec}$  ist 0,7, was bedeutet, dass die Wahrscheinlichkeit einer zufälligen Aktion nach 10 Trainingsiterationen nur noch 3% beträgt. Dies sollte der Regel-KI genügend Zeit geben, das Systemverhalten zu lernen.

### 4.2.1 Einfluss der Fenstergröße auf den Regelungserfolg

Im folgenden Abschnitt wird untersucht, wie sich eine Variation von  $n$ ,  $m$ ,  $h$ ,  $t_{act}$  und  $t_{train}$  auf die Regelung und die Vorhersagegenauigkeit der Regel-KI auswirkt. Die Menge der Trainingsdaten pro Trainingsiteration hängt dabei von  $n$ ,  $h$ ,  $t_{act}$  und  $t_{train}$  ab. Je größer das Zeitfenster ist, desto weniger Trainingsdaten stehen zur Verfügung. Wird das Zeitfenster zu klein gewählt, kann es bei komplexen Systemen vorkommen, dass zeitliche Zusammenhänge nicht erkannt werden. Es muss ein Zeitfenster gewählt werden, das groß genug ist, um die zeitlichen Zusammenhänge abzubilden, ohne dass die Menge der Trainingsdaten zu klein wird.

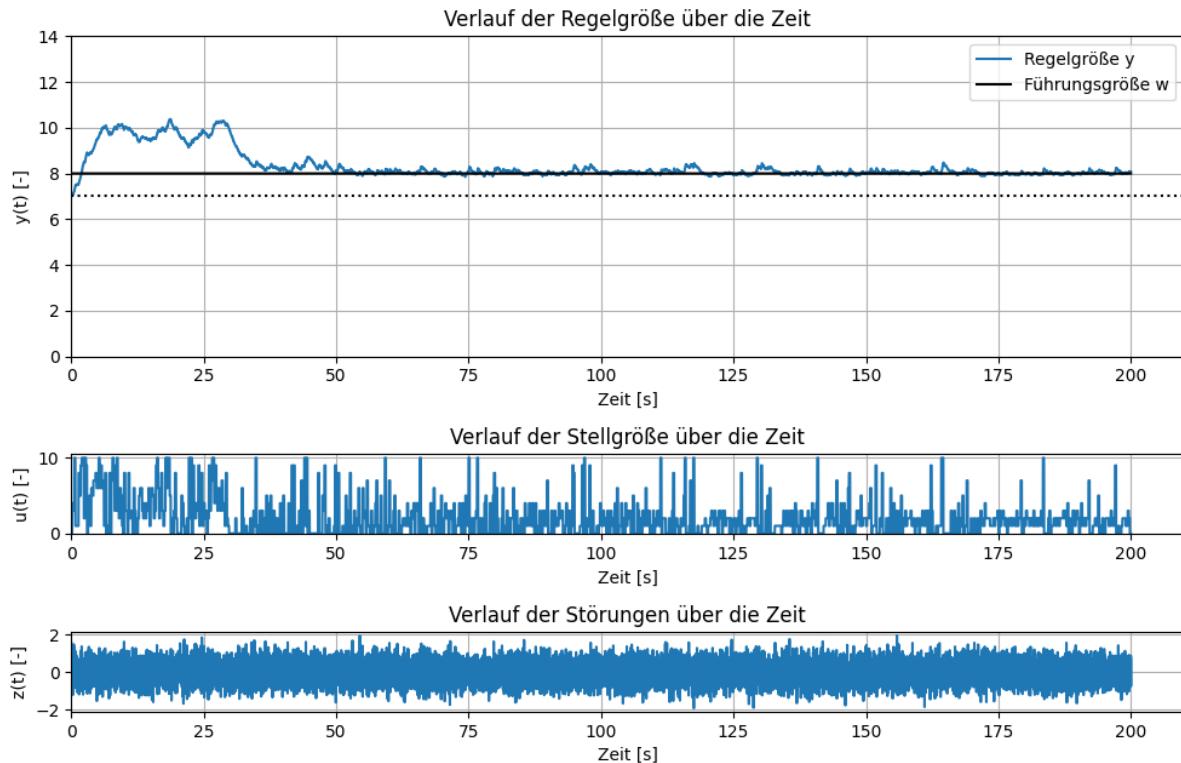


Abbildung 24: Simulationsverlauf für  $n = 4$ ,  $m = 2$ ,  $h = 4$ ,  $t_{act} = 200\text{ms}$ ,  $t_{train} = 10\text{s}$

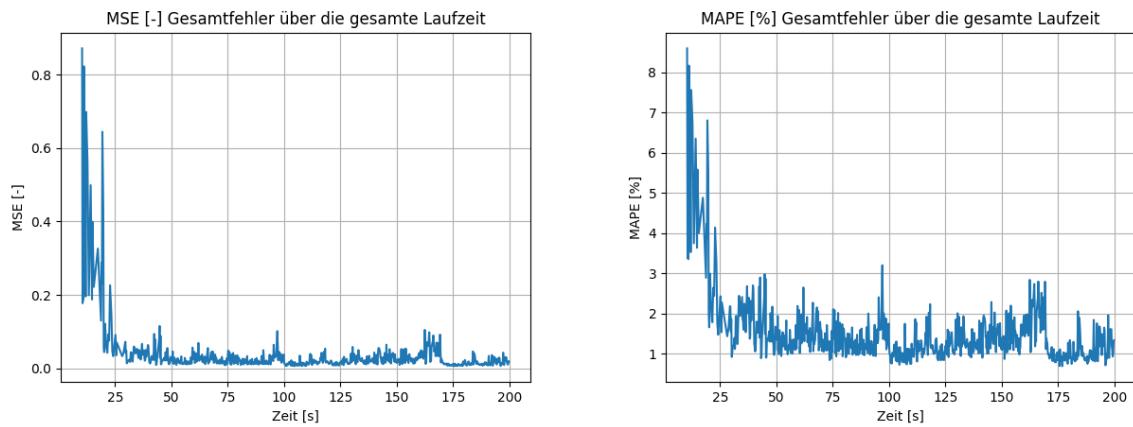


Abbildung 25: MSE und MAPE für  $n = 4$ ,  $m = 2$ ,  $h = 4$ ,  $t_{act} = 200\text{ms}$ ,  $t_{train} = 10\text{s}$

Zuerst wird die Regel-KI mit einem symmetrischen Zeitfenster mit  $n = 4$ ,  $m = 2$ ,  $h = 4$ ,  $t_{act} = 200\text{ms}$  und  $t_{train} = 10\text{s}$  untersucht. Die Wahl dieser Parameter führt zu einer Fenstergröße von 1,6 Sekunden (38) und 42 neuen Daten pro Trainingsiteration (39). Der Verlauf einer Simulation mit diesen Parametern ist in Abbildung 24 dargestellt. Es

ist zu erkennen, dass die Regel-KI die Führungsgröße nach etwa 50 Sekunden erreicht und über den gesamten Zeitraum hält. In Abbildung 25 ist zu erkennen, dass bereits nach wenigen Vorhersagen der Gesamtfehler (MSE und MAPE) unter 0,1 beziehungsweise 3% sinkt. Dies ist darauf zurückzuführen, dass das virtuelle System mit konstanten Zustandsgrößen nicht komplex ist und die Regel-KI daher die zeitlichen Zusammenhänge schnell lernt. Bei komplexeren Systemen würde die Regel-KI vermutlich länger brauchen, um die zeitlichen Zusammenhänge zu lernen. Der Grund, warum die Regel-KI die Führungsgröße erst nach 50 Sekunden erreicht, obwohl der Vorhersagefehler schon nach wenigen Sekunden gering ist, liegt am Epsilon-Greedy-Algorithmus, der in dem Bereich von 0 Sekunden bis 50 Sekunden häufig zufällige Aktionen auswählt. Mit  $\varepsilon_{dec} = 0,7$  beträgt die Wahrscheinlichkeit für eine zufällige Aktion bei  $t = 25s$   $0,7^2 = 49\%$ , da die KI bis zu diesem Zeitpunkt zweimal trainiert wurde. Nach 30 Sekunden wird die KI ein drittes Mal trainiert, wobei  $\varepsilon$  auf 34,3 % sinkt. Ab diesem Zeitpunkt ist zu erkennen, dass sich die Regel-KI langsam der Führungsgröße annähert. Weiterhin ist am Verlauf der Stellgröße zu erkennen, dass es nach dem Erreichen der Führungsgröße zu kleinen Ausschlägen kommt. Dies liegt in den meisten Fällen daran, dass in dieser Simulation mit einer Wahrscheinlichkeit von 10% eine der 10 besten Regelstrategien ausgewählt wird und dass die Nebenbedingung (34), die die maximale Anstiegsgeschwindigkeit der Stellgröße begrenzt, nicht berücksichtigt wird.

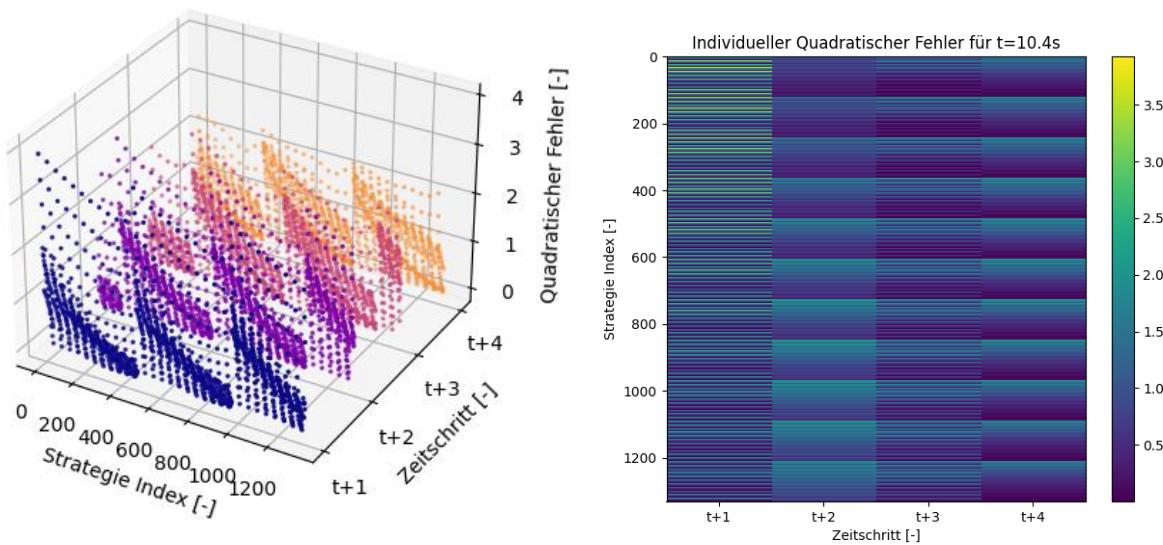


Abbildung 26: Quadratischer Fehler über alle Regelstrategien und Zeitschritte der ersten Vorhersage für  $t = 11,2\text{s}$

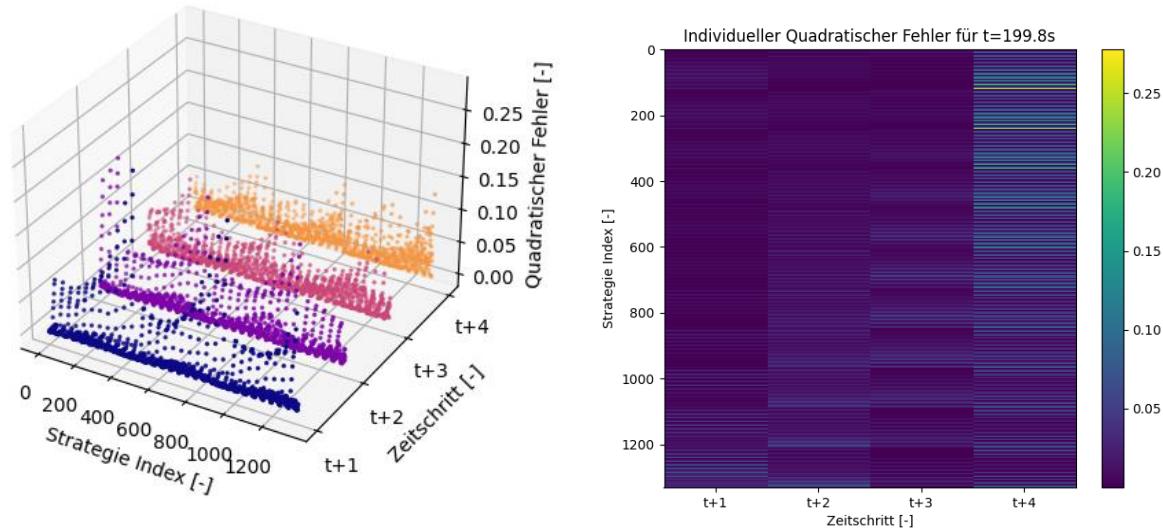


Abbildung 27: Quadratischer Fehler über alle Regelstrategien und Zeitschritte der letzten Vorhersage für  $t = 199,8\text{s}$

In Abbildung 26 und Abbildung 27 sind die quadratischen Fehler der Vorhersagen für jeden Zeitschritt und für jede Regelstrategie in einem 3D-Scatterplot (links) und einer Heatmap (rechts) dargestellt. Jeder Punkt im 3D-Scatterplot repräsentiert die Vorhersage einer Regelstrategie für einen bestimmten Zeitschritt. Die Heatmap kann als Vogelperspektive des 3D Scatterplots interpretiert werden, wobei die Größe des Fehlers durch einen Farbverlauf dargestellt wird, ähnlich wie auf einer Landkarte, die Berge darstellt.

Die erste der beiden Abbildungen bezieht sich auf die erste Vorhersage der Regel-KI. Es ist zu erkennen, dass die Vorhersage zu diesem Zeitpunkt ungenau ist, mit einem maximalen quadratischen Fehler von etwa 3,5. Die zweite der beiden Abbildungen zeigt den Fehler der letzten Vorhersage. Es ist zu erkennen, dass der Vorhersagefehler deutlich geringer ist als zu Beginn der Simulation. Der maximale quadratische Fehler beträgt am Ende nur noch etwa 0,25. Weiterhin ist zu erkennen, dass die Vorhersage im letzten Zeitschritt am ungenauesten ist, was auf die Differenz zwischen dem Regelhorizont und dem Prognosehorizont zurückzuführen ist. Dieser Effekt wird im Abschnitt „Einfluss von h und m“ näher erläutert.

**Einfluss des Trainingsintervalls ( $t_{train}$ ):** Die Variation von  $t_{train}$  ist hat Einfluss darauf, wie oft die Regel-KI trainiert wird und damit auch darauf, wie schnell  $\varepsilon$  abnimmt, da nach jeder Trainingsiteration der Wert von  $\varepsilon$  mit  $\varepsilon_{dec}$  multipliziert wird. Je häufiger die Regel-KI trainiert wird, desto schneller nimmt die Wahrscheinlichkeit ab, dass zufällige Aktionen ausgeführt werden.

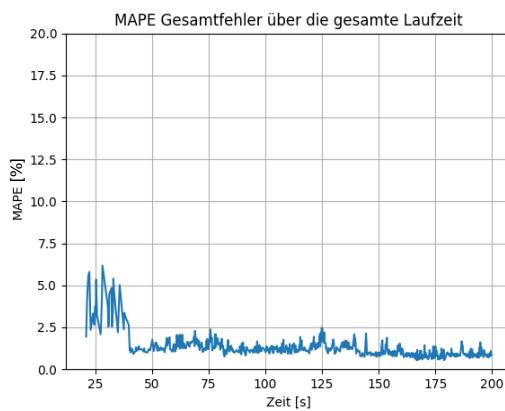
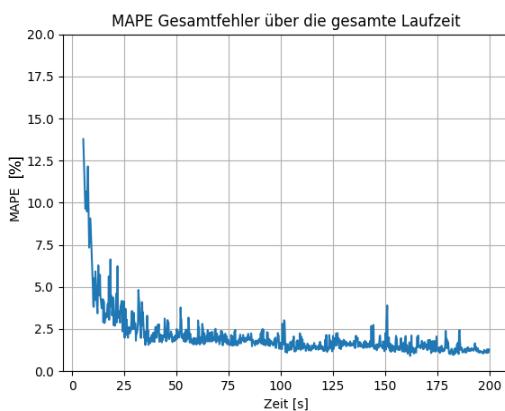


Abbildung 28: MAPE für  $t_{train} = 5\text{s}$  (links) und  $t_{train} = 20\text{s}$  (rechts)

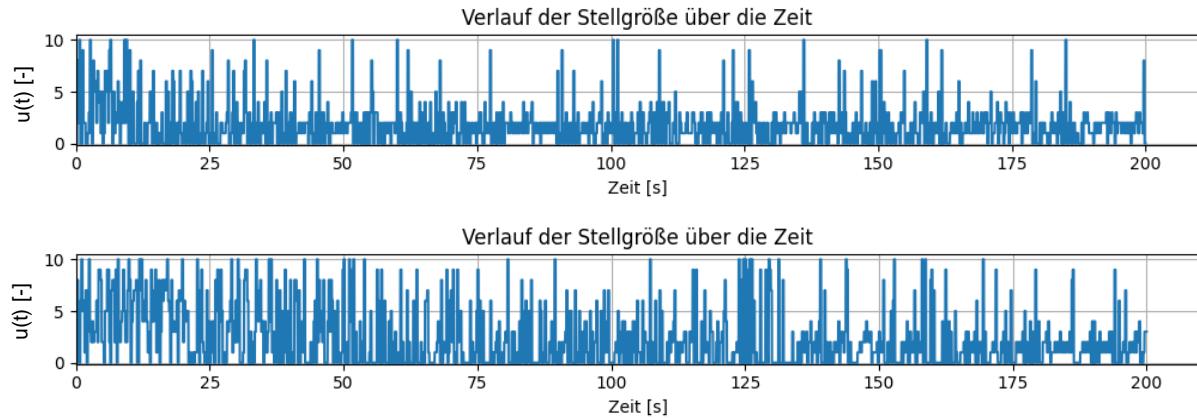


Abbildung 29: Verlauf der Stellgröße für  $t_{train} = 5\text{s}$  (oben) und  $t_{train} = 20\text{s}$  (unten)

Der Vergleich des Vorhersagefehlers bei  $t_{train} = 5\text{s}$  und  $t_{train} = 20\text{s}$  ist in Abbildung 28 dargestellt, in der jeweils der MAPE-Fehler der Vorhersage über die gesamte Laufzeit dargestellt ist. Die Zeitachse der Graphen beginnt jeweils nach 5s und 20s, da die Regel-KI vor dem ersten Trainingslauf keine Vorhersagen trifft, da  $\varepsilon$  bis zum ersten Training 100 % beträgt. Ein interessantes Ergebnis ist im rechten Graphen zu sehen, wo der Vorhersagefehler nach 40 Sekunden, also nach der zweiten Trainingsiteration, stark abfällt. Nach einer Laufzeit von 20 Sekunden ist der Verlauf der beiden Fehler ähnlich. Daraus folgt, dass  $t_{train}$  bei diesem System keinen großen Einfluss auf den Regelungserfolg hat. Der Einfluss von  $t_{train}$  auf die Stellgröße ist in Abbildung 29 dargestellt. Es ist zu erkennen, dass bei  $t_{train} = 5\text{s}$  die Schwankungen der Stellgröße geringer sind als bei  $t_{train} = 20\text{s}$ , da  $\varepsilon$  schnell abnimmt und somit weniger zufällige Aktionen ausgeführt werden.

**Einfluss des Regelintervalls ( $t_{act}$ ):** Der Wert von  $t_{act}$  bestimmt, in welchen Zeitabständen die Regel-KI eingreifen soll. Ein größerer Wert für  $t_{act}$  führt zu einem größeren Abstand zwischen den Zeitpunkten und damit zu einem größeren Zeitfenster. Dadurch kann die Stellgröße seltener geändert werden, was bei einer fehlerhaften Berechnung der optimalen Regelstrategie zu einer größeren Schwankung der Regelgröße führen kann. Außerdem werden weniger Trainingsdaten gespeichert. Es

kann auch zu größeren Vorhersagefehlern kommen, da für ein größeres  $t_{act}$ , Werte weiter in der Zukunft vorhergesagt werden müssen. Dafür werden bei größerem  $t_{act}$  längere zeitliche Zusammenhänge zwischen Stellgröße und Regelgröße erfasst.

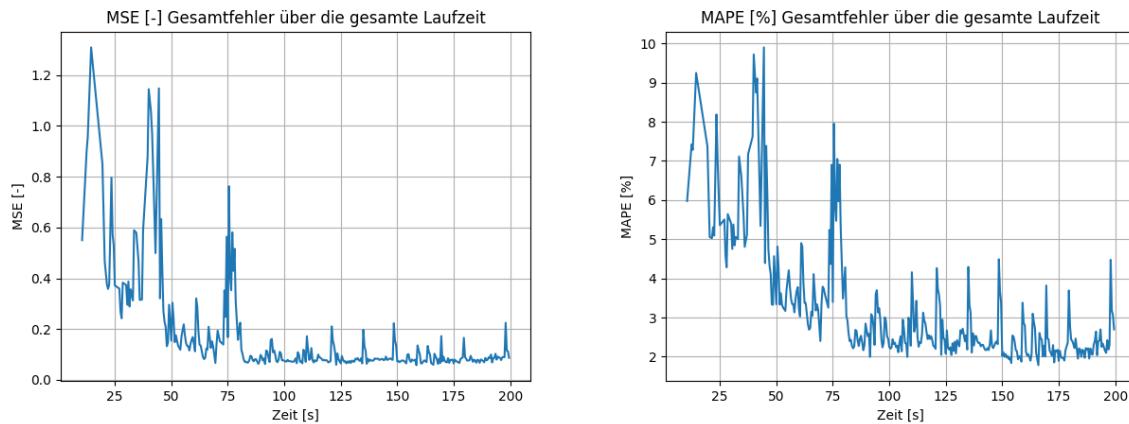


Abbildung 30: MSE und MAPE für  $t_{act} = 500\text{ms}$

Der Verlauf des Vorhersagefehlers für  $t_{act} = 500\text{ms}$  ist in Abbildung 30 dargestellt. Dieser ist im Vergleich zum Fehler für  $t_{act} = 200\text{ms}$  aus Abbildung 25 über die gesamte Laufzeit höher und weist größere Schwankungen auf. Dies liegt vermutlich daran, dass das Zeitfenster mit 4s mehr als doppelt so groß ist wie das Zeitfenster für  $t_{act} = 200\text{ms}$ , wodurch Werte weiter in der Zukunft vorhergesagt werden müssen, was zu größeren Fehlern führt. Dennoch ist die Regelung in beiden Fällen erfolgreich.

**Einfluss von  $h$  und  $m$ :** Je größer der Regel- und Prognosehorizont gewählt wird, desto vorausschauender kann die Regel-KI agieren, allerdings steigt damit auch der Rechenaufwand zur Bewertung der möglichen Regelstrategien. Die Wahl des Prognosehorizonts hängt auch von der Komplexität des Systems ab. Bei weniger komplexen Systemen ist es sinnvoll, einen kleineren Horizont zu wählen, um den Rechenaufwand zu reduzieren. Bei komplexeren Systemen mit stark verzögertem Übertragungsverhalten ist ein größerer Horizont von Vorteil.

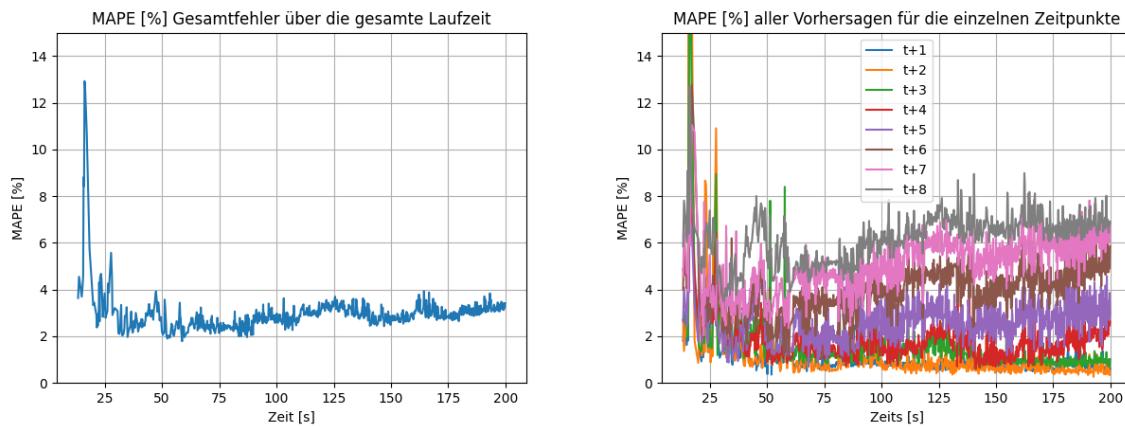


Abbildung 31: MAPE-Gesamtfehler und MAPE für jeden Zeitpunkt für  $h = 8$  und  $m = 2$

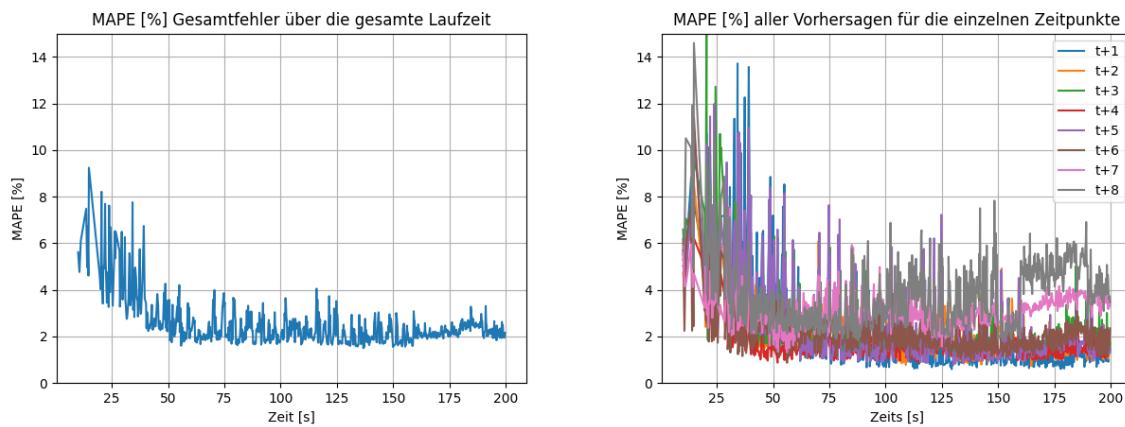


Abbildung 32: MAPE-Gesamtfehler und MAPE für jeden Zeitpunkt für  $h = 8$  und  $m = 4$

Abbildung 31 zeigt, dass Vorhersagen für Werte, die weiter in der Zukunft liegen, höhere Fehler aufweisen als Werte, die näher in der Zukunft liegen, was zu einem höheren Gesamtfehler führt. Dies hängt damit zusammen, dass  $m = 2$  bleibt, wodurch die Regel-KI mit den gleichen Eingaben mehr Werte vorhersagen muss. Hier fehlen Daten, um den zeitlichen Zusammenhang für die Werte der Zeitpunkte zwischen  $t + m$  und  $t + h$  zu lernen. Je größer die Differenz zwischen  $m$  und  $h$  ist, desto ungenauer wird die Vorhersage für weiter in der Zukunft liegende Zeitpunkte, wie der Vergleich zwischen Abbildung 31 und Abbildung 32 zeigt. Andererseits gilt, je größer  $m$  ist, desto größer ist die Anzahl der möglichen Regelstrategien und desto größer ist

der Rechenaufwand, um alle Strategien zu bewerten. Grundsätzlich ist die Wahl der Größe von  $m$  ein Kompromiss zwischen Regelleistung und Rechenaufwand. Es ist notwendig, ein Gleichgewicht zu finden, das den Anforderungen der jeweiligen Anwendung entspricht. Dabei sollte der Prognosehorizont so gewählt werden, dass die zeitlichen Zusammenhänge zwischen Stell- und Regelgröße möglichst optimal erfasst werden, ohne dass die Differenz zwischen dem Prognosehorizont und dem Regelhorizont zu groß wird.

**Einfluss von  $n$ :** Je größer  $n$  ist, desto mehr vergangene Werte werden für die Vorhersage verwendet und desto besser können zeitliche Zusammenhänge abgebildet werden. Es ist zu erwarten, dass ein größeres  $n$  zu einem geringeren Vorhersagefehler führt, da mehr Zeitpunkte für die Vorhersage berücksichtigt werden.

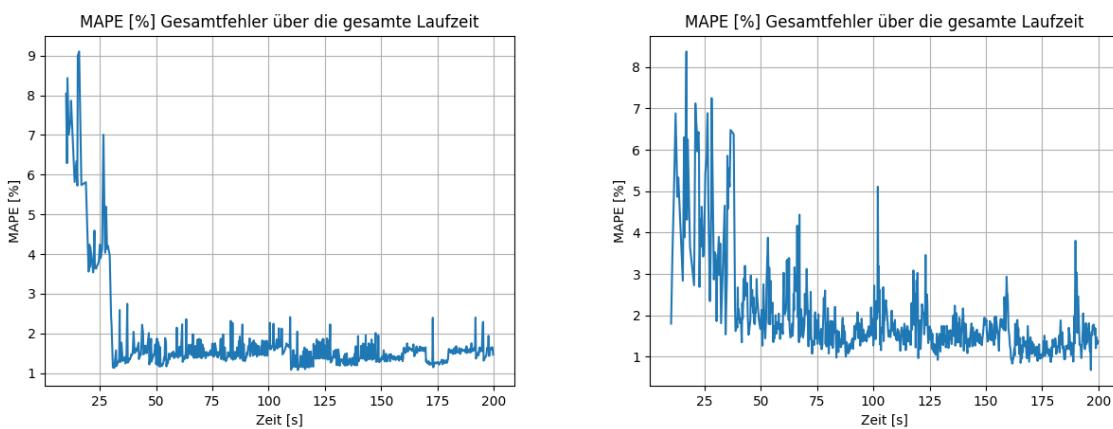


Abbildung 33: MAPE-Gesamtfehler für  $n = 1$  (links) und  $n = 8$  (rechts)

Abbildung 33 zeigt den Vergleich des MAPE-Gesamtfehlers für  $n = 1$  und  $n = 8$ . In diesem Fall ist die Vorhersage für  $n = 1$  genauer. Dies hängt vermutlich damit zusammen, dass in diesem Fall ein statisches System betrachtet wird, bei dem die internen Zustandsgrößen über die gesamte Laufzeit konstant bleiben und somit keine längerfristigen, dynamischen Zusammenhänge zwischen Stell- und Regelgröße existieren. Außerdem stehen bei einem kleineren Zeitfenster mehr Trainingsdaten zur Verfügung. Die Information über die Steigung wird für ein Zeitfenster mit  $n = 1$  nicht

erfasst, weshalb in den meisten Fällen  $n$  mindestens zwei sein sollte. In beiden Fällen ist die Regelung stabil.

Letztendlich ist die Wahl der Parameter für die Fenstergröße bei diesem virtuellen System nicht entscheidend für den Regelungserfolg, da die Führungsgröße in jedem Fall erreicht und gehalten wird. Die Parameter wirken sich lediglich auf die Stabilität der Regelung beziehungsweise auf die Größe der Schwankungen der Regelgröße und den Vorhersagefehler der Regel-KI aus.

#### 4.2.2 Einfluss der Prozessparameter auf den Regelungserfolg

Die Wahl der Menge  $U$  der möglichen Werte für die Stellgröße hängt von  $k_1$ ,  $k_2$  und  $k_s$  ab. Es sollte dabei in jedem Fall eine mögliche Stellgröße  $u \in U$  geben, die das System in Richtung der Führungsgröße bewegen kann. Zum Beispiel sollte der größte mögliche Wert der Stellgröße in der Lage sein, die Regelgröße zu ändern, wenn sie den minimalen Wert erreicht. Für  $y_s = 7$ ,  $k_1 = 0,2$ ,  $k_2 = 0,05$ ,  $z(t) = 0$  und  $y(t) = y_{min} = 0$  ist die Steigung mit (41)  $\dot{y} = -1,45$ . Damit das System den Minimalwert verlassen kann, muss in diesem Fall  $k_s u(t) > 1,45$  sein. Für  $k_s = 0,15$  muss es mindestens ein  $u \in U$  geben, das größer als 9,66 ist, um die Steigung in den positiven Bereich zu bringen. Ist diese Bedingung nicht erfüllt, kann die Regelgröße bei Erreichen von  $y = 0$  nicht mehr beeinflusst werden. Die allgemeine Bedingung für eine erfolgreiche Regelung in diesem System ist

$$k_s u_{max} > k_1 |y(t) - y_s| + k_2 - z(t) > k_s u_{min} \quad (43)$$

Ein größerer Wert für  $k_s$  oder größere Werte für die Stellgröße können zu größeren Schwankungen der Regelgröße führen, insbesondere wenn die Führungsgröße in der Nähe von  $y_s$  liegt, da dann die Steigung klein ist und ein großer Wert für  $k_s u(t)$  zu einer großen Auslenkung der Regelgröße führen kann.

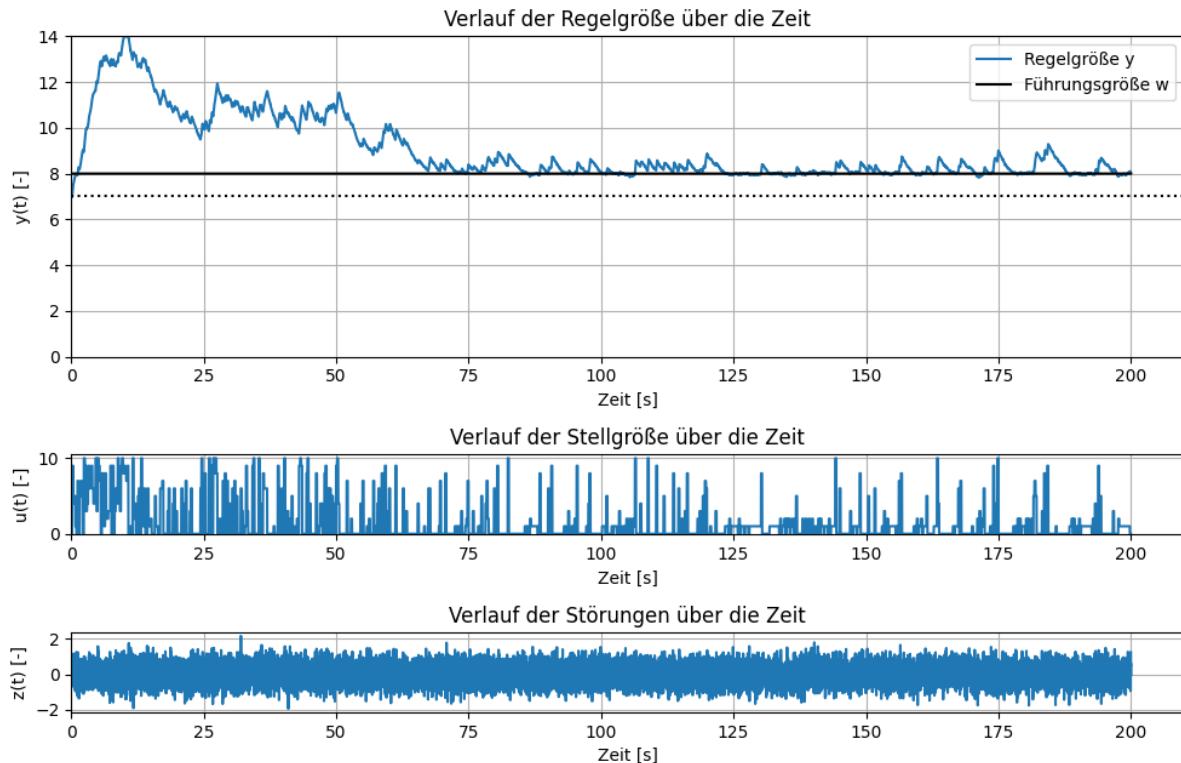
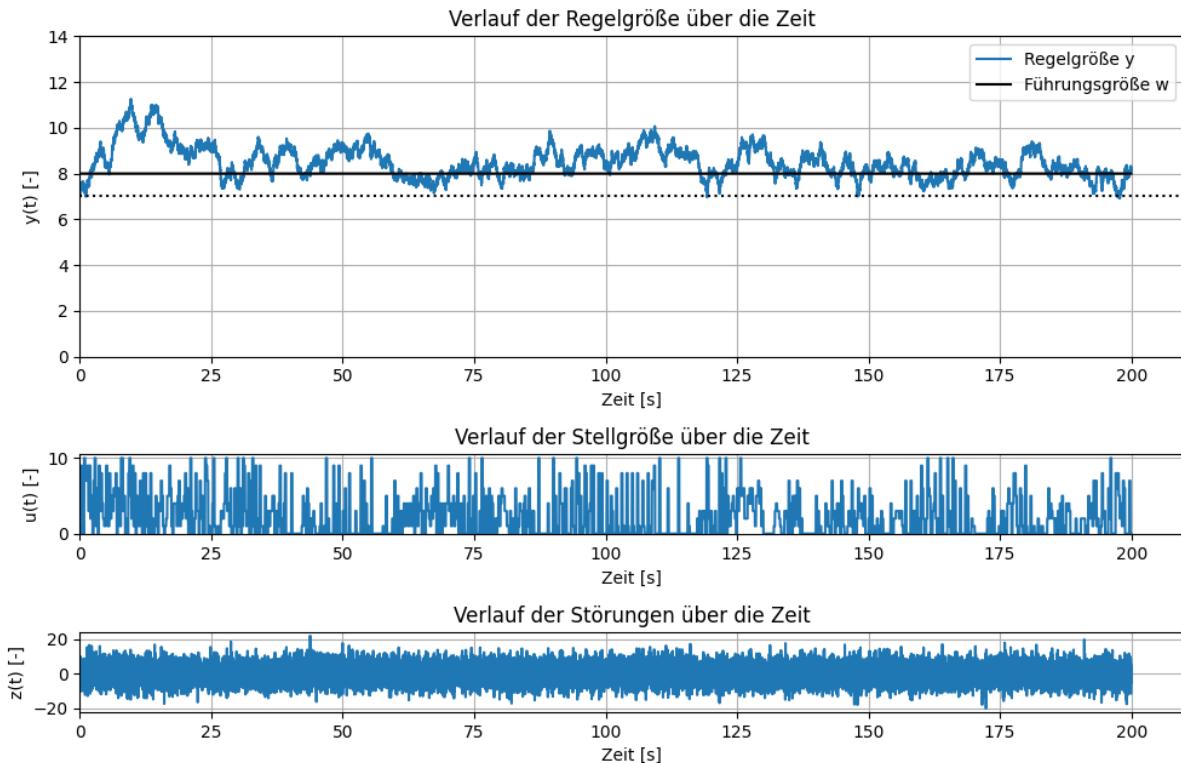


Abbildung 34: Simulationsverlauf für  $k_s = 0,3$

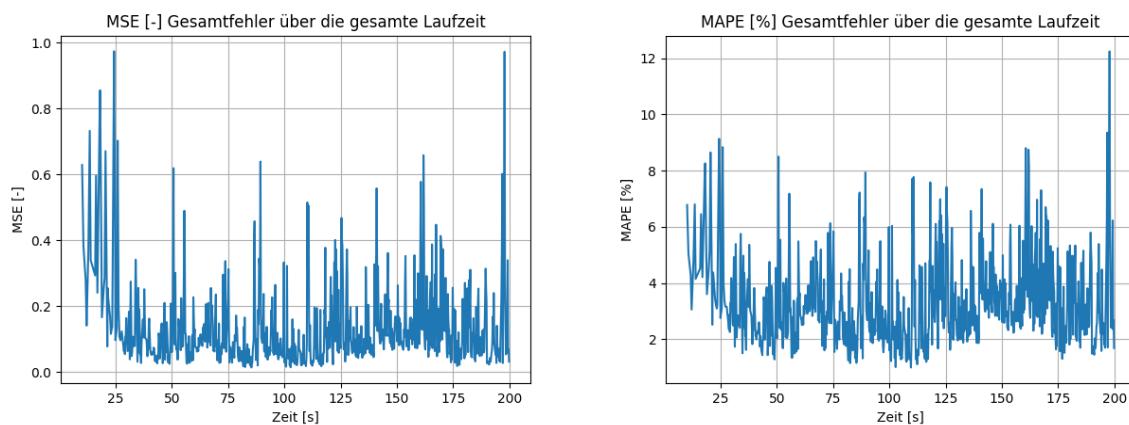
Abbildung 34 zeigt den Verlauf einer Simulation für  $k_s = 0,3$ , bei gleicher Fenstergröße ( $n = 4, m = 2, h = 4$ ). Im Vergleich einer Simulation mit  $k_s = 0,15$  (siehe Abbildung 24) ist die Regelung mit  $k_s = 0,3$  instabiler und weist höhere Schwankungen auf. Dies ist darauf zurückzuführen, dass ein Ausschlag der Stellgröße mit einem größeren  $k_s$  einen größeren Einfluss auf die Regelgröße hat.

#### 4.2.3 Einfluss von Störungen auf den Regelungserfolg

In diesem Abschnitt wird die Reaktion der Regel-KI auf verschiedene Störungen untersucht. Zunächst wird der Einfluss von hohem Rauschen untersucht. Anschließend wird die Stabilität der Regel-KI durch Anregung des Systems mit diskreten Störimpulsen überprüft. Abschließend wird die Reaktion der Regel-KI auf eine kontinuierliche, dynamische Störung untersucht.



**Abbildung 35: Simulationsverlauf für ein hohes Rauschen**



**Abbildung 36: MSE und MAPE für ein hohes Rauschen**

Abbildung 35 zeigt den Verlauf der Simulation für ein hohes Rauschen mit  $\sigma_r = 5$ . Im Grunde stellt das Rauschen die Auswirkung von kleinen Störimpulsen zu jedem Zeitschritt der Simulation dar. Es ist ersichtlich, dass hohe Schwankungen vorliegen und die Regel-KI nicht in der Lage ist, das System ausreichend zu

stabilisieren. Dennoch nähert sich die Regelgröße grob der Führungsgröße an. Abbildung 36 zeigt, dass der Vorhersagefehler volatil ist, was durch das hohe zufällige Rauschen zu erklären ist. Da es sich um zufällige Störungen handelt, ist die Regel-KI nicht in der Lage, die Regelgröße akkurat vorherzusagen.

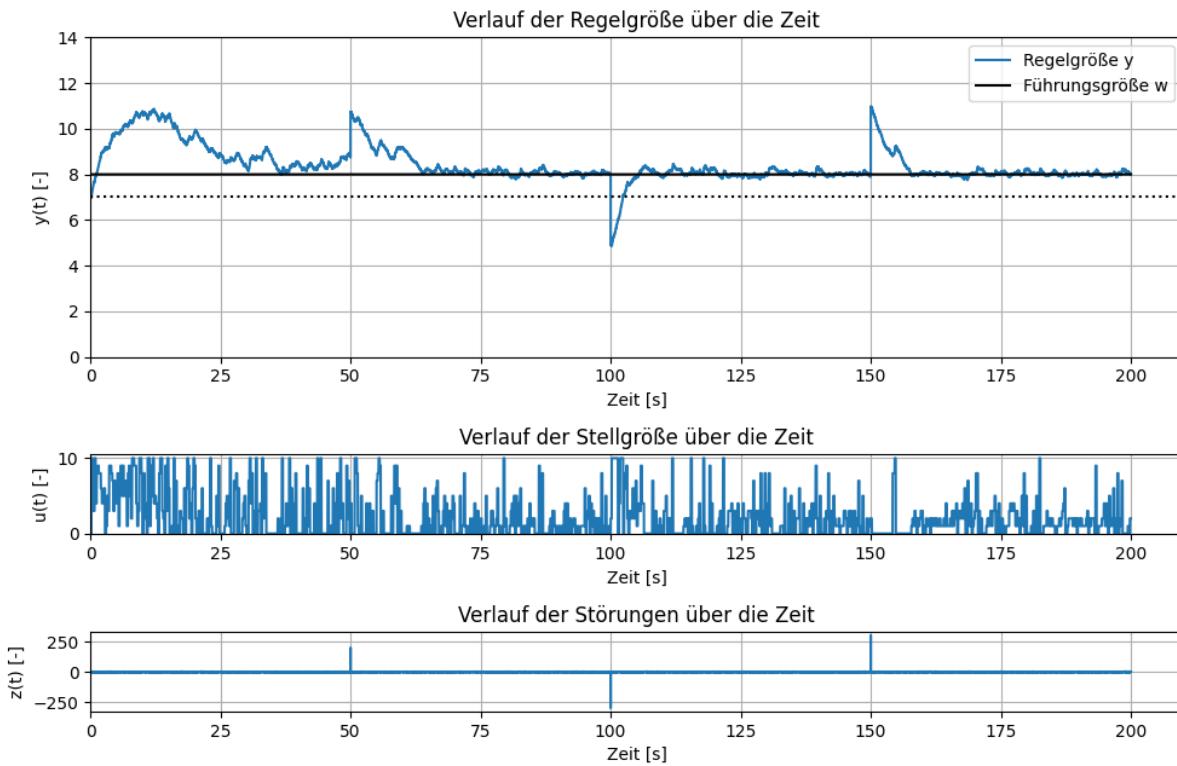


Abbildung 37: Simulationsverlauf beim Anregen durch Störimpulse

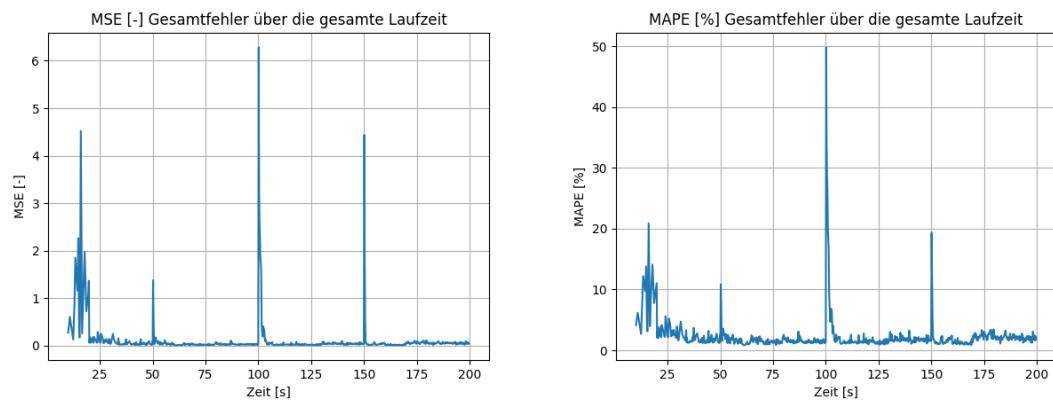


Abbildung 38: MSE und MAPE beim Anregen durch Störimpulse

Abbildung 37 zeigt den Verlauf einer Simulation, bei der zu den Zeitpunkten  $t = 50\text{s}$ ,  $t = 100\text{s}$  und  $t = 150\text{s}$  jeweils ein Störimpuls auftritt. In diesem Fall beträgt  $\sigma_r = 0,5$ . Es ist zu erkennen, dass die Regelgröße zu diesen Zeitpunkten einen deutlichen Ausschlag aufweist. Nach einigen Sekunden stabilisiert sich die Regelung jedoch wieder. Abbildung 38 zeigt, dass zu den Zeitpunkten, an denen ein Störimpuls auftritt, der Vorhersagefehler stark ansteigt, da die Regel-KI diesen nicht vorhersagen kann. Nach einigen Sekunden sinkt der Fehler jedoch wieder auf die ursprüngliche Größe zurück. Ein Störimpuls hat demnach keinen zukünftigen Einfluss auf den Vorhersagefehler, was eine Stabilisierung der Regelung nach einem Störimpuls ermöglicht.

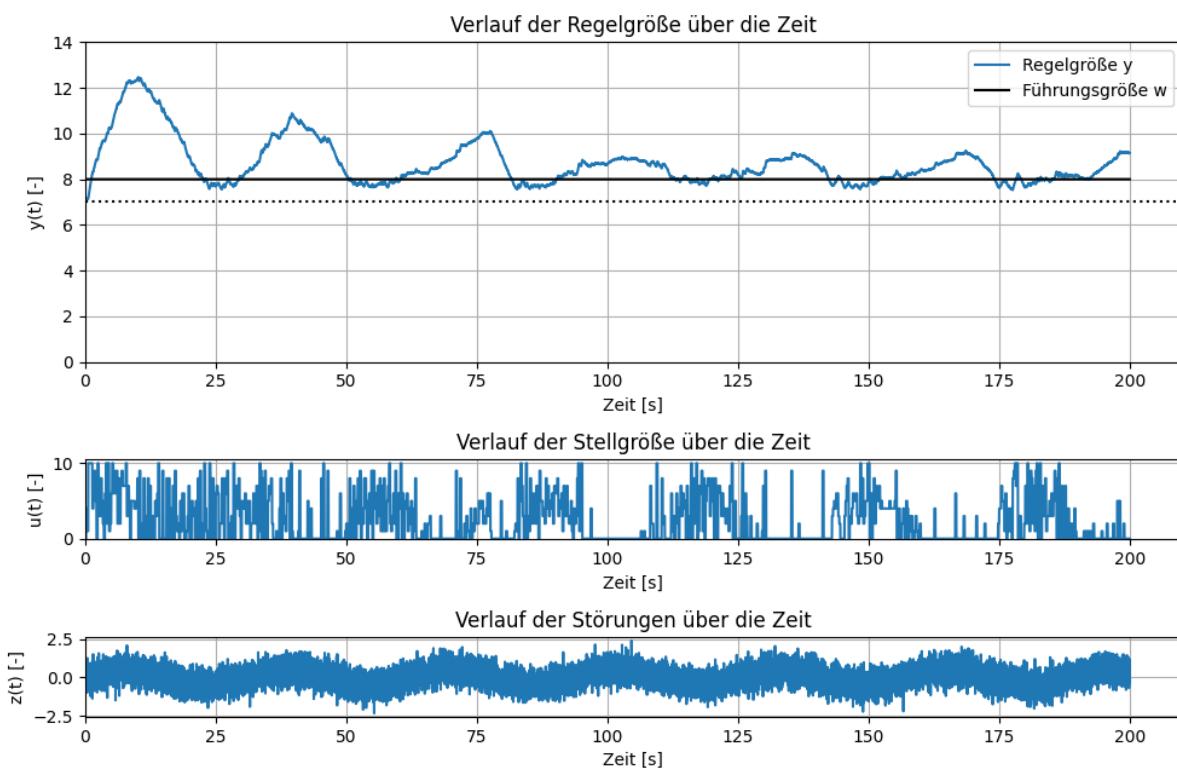


Abbildung 39: Simulationsverlauf bei einem dynamischen Störverhalten mit  $A = 0,5$

In Abbildung 39 ist der Verlauf einer Simulation bei einem dynamischen Störverhalten dargestellt. Die dynamische Störung wird in diesem Fall durch die Funktion

$$z_d(t) = A * \sin(0,2t) \quad (44)$$

mit der Amplitude  $A = 0,5$  bestimmt. Der Wert für  $\sigma_r$  ist hier wieder 0,5. Der Simulationsverlauf zeigt, dass die dynamische Störung den Erwartungswert  $\mu_r$  des zufälligen Rauschens verschiebt, da sich  $z_r(t)$  und  $z_d(t)$  zu einer Gesamtstörung  $z(t)$  aufsummieren. Es ist zu erkennen, dass die Regelung instabil ist und starke Schwankungen aufweist. Dies ist darauf zurückzuführen, dass die Bedingung (43) nicht erfüllt ist. Soll  $y(t)$  die Führungsgröße halten, so gilt  $y(t) = w(t) = 8$ . Der minimale Erwartungswert der Störung beträgt hier -0,5. Daraus ergibt sich mit  $y_s = 7$ ,  $k_1 = 0,2$  und  $k_2 = 0,05$  die Bedingung  $k_s u_{min} < -0,25$ . Da  $u_{min} = 0$  ist, wird diese Bedingung nicht erfüllt. Der Einfluss der dynamischen Störung übersteigt somit die Fähigkeit der Regel-KI, diese zu kompensieren. Wenn die maximale Amplitude der dynamischen Störung kleiner als 0,25 ist, ist die Bedingung erfüllt.

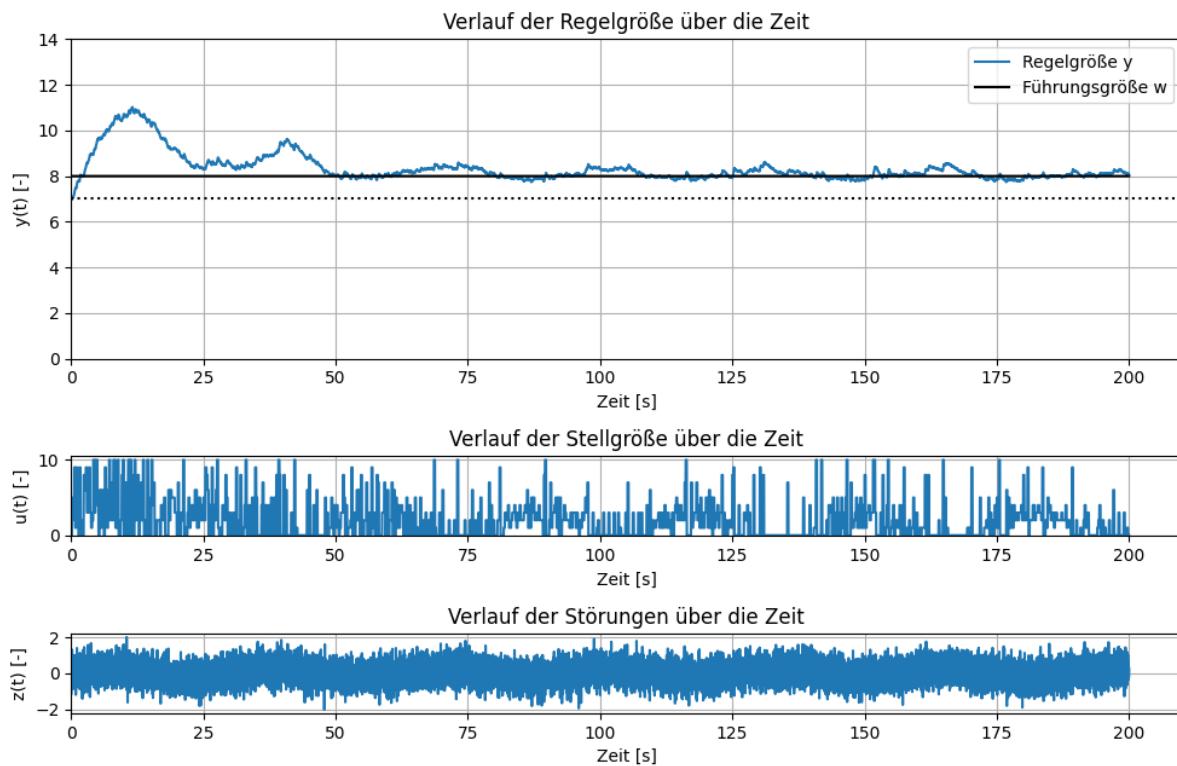


Abbildung 40: Simulationsverlauf bei einem dynamischen Störverhalten mit  $A = 0,2$

Der Simulationsverlauf für die dynamischen Störung (44) mit  $A = 0,2$  ist in Abbildung 40 dargestellt. Es ist zu erkennen, dass die Regelung bei Einhaltung der Bedingung (43) stabil ist.

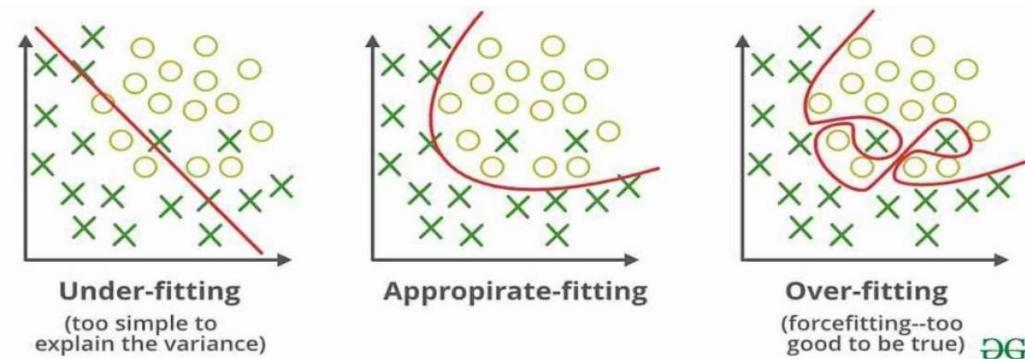


Abbildung 41: Darstellung von Underfitting und Overfitting [42]

Ein hohes Maß an zufälligen Störungen kann beim Training eines KNN zu „Overfitting“ führen. Dies geschieht, wenn das Modell versucht, Zusammenhänge zu erkennen, die in Wirklichkeit nicht existieren. Dies führt dazu, dass das Modell zu stark an die Trainingsdaten angepasst wird und dadurch seine Fähigkeit verliert, neue und unbekannte Daten zu generalisieren. Abbildung 41 zeigt den Unterschied zwischen „Underfitting“ (Unteranpassung) und „Overfitting“ (Überanpassung). Eine Methode, um Overfitting zu vermeiden besteht darin, die Komplexität des Modells zu reduzieren, indem die Anzahl der verborgenen Schichten verringert wird. Es kann auch die „Dropout“ Methode verwendet werden, bei der während des Trainings zufällig Neuronen gelöscht werden. Außerdem kann es sinnvoll sein, Daten mit hohem Rauschen vor dem Training zu filtern und zu normalisieren. [8]

Es sollte auch darauf geachtet werden, dass der Hyperparameter „Epochs“ nicht zu groß gewählt wird. Die Anzahl der Epochen gibt an, wie oft ein Trainingsdatensatz iteriert wird, wobei die Gewichte des KNN bei jeder Iteration aktualisiert werden. Eine hohe Anzahl von Epochen kann zu Overfitting führen, während eine zu niedrige Anzahl von Epochen zu Underfitting führen kann. Dies gilt insbesondere für diesen

Anwendungsfall, in dem das KNN kontinuierlich mit neuen und kleinen Datensätzen trainiert wird. In allen Simulationsläufen wurde die Anzahl der Epochen auf 50 gesetzt.

#### 4.2.4 Auswirkung der Nebenbedingung $\Delta u_{max}$ auf den Regelungserfolg

Im folgenden Abschnitt werden die Regelstrategien unter Berücksichtigung der Nebenbedingung (34) ausgewählt. Es werden nur die Regelstrategien betrachtet, bei denen die maximale Änderung der Stellgröße zwischen zwei Zeitpunkten kleiner als ein vorgegebenes  $\Delta u_{max}$  ist. Dadurch wird die Anzahl der möglichen Regelstrategien reduziert und somit auch der Rechenaufwand für die Vorhersage. Zudem kann es nicht mehr zu großen Sprüngen der Stellgröße kommen, was die Regelung stabiler macht.

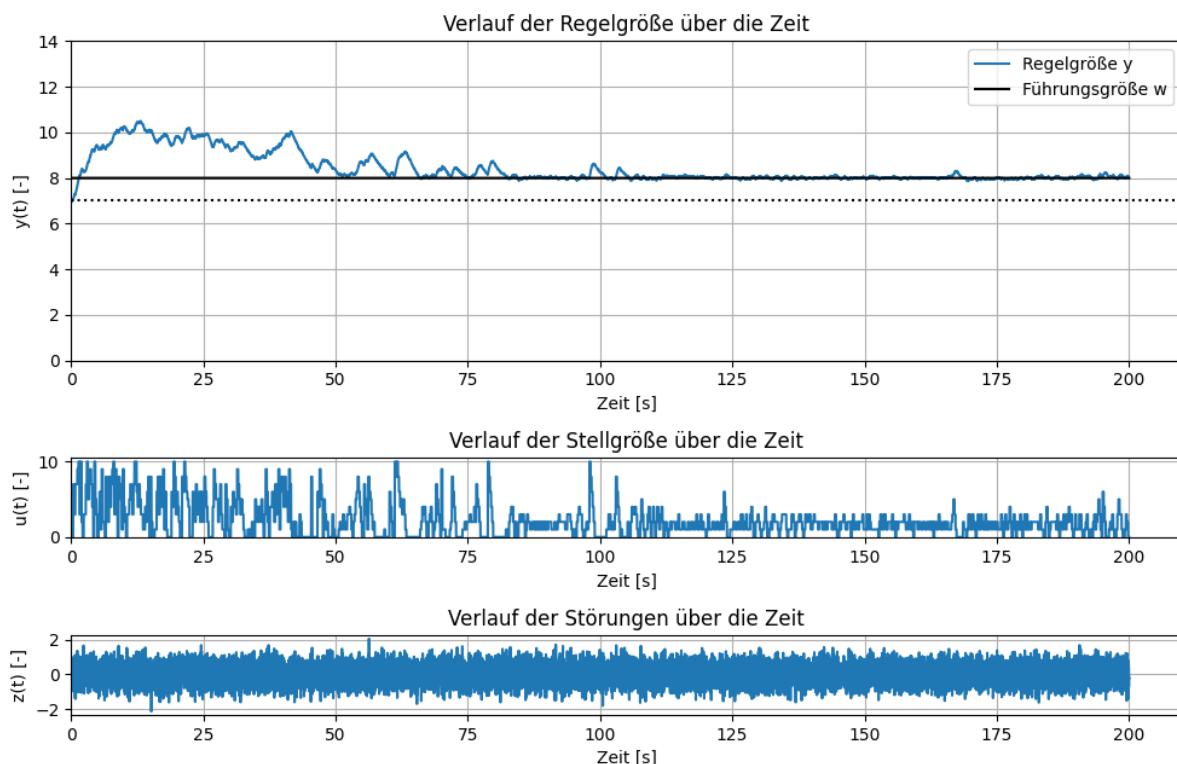


Abbildung 42: Simulationsverlauf für  $\Delta u_{max} = 2$

Abbildung 42 zeigt den Verlauf der Simulation für  $n = 4$ ,  $m = 2$ ,  $h = 4$  und  $\Delta u_{max} = 2$ . Es ist ersichtlich, dass es ab  $t > 110s$  kaum noch Ausschläge der Stellgröße gibt und

die Regelgröße die Führungsgröße nahezu exakt hält. Für einen Regelhorizont von  $m = 2$  und der gegebenen Menge  $U$  ergeben sich ohne Verwendung der Nebenbedingung  $11^3 = 1331$  mögliche Regelstrategien. Mit  $\Delta u_{max} = 2$  sinkt die Anzahl der möglichen Regelstrategien auf maximal 123. Durch die Einführung der Nebenbedingung wird die Regelung stabiler, während der Rechenaufwand sinkt.

#### 4.2.5 Einordnung der Einflussgrößen auf den Regelungserfolg

In diesem Abschnitt wird der Einfluss der verschiedenen Parameter auf den Regelungserfolg zusammengefasst. Dazu werden die Parameter nach der Größe ihres Einflusses auf den Regelungserfolg in eine Rangfolge gebracht.

Tabelle 9: Rangfolge der Einflussfaktoren auf den Regelungserfolg

Rangfolge	Einflussgrößen
1	Prozessparameter ( $k_1, k_2, k_s, U$ )
2	Nebenbedingung $\Delta u_{max}$
3	Größe des Rauschens
4	Differenz zwischen $h$ und $m$
5	Regelintervall $t_{act}$
6	Dynamische & Impulsartige Störungen
7	Trainingsintervall $t_{train}$
8	Größe von $n$

Tabelle 9 zeigt die Rangfolge der verschiedenen Einflussfaktoren auf den Regelungserfolg. Den größten Einfluss auf den Regelungserfolg haben die Prozessparameter, was auf die Bedingung (43) zurückzuführen ist. Damit eine

Regelung überhaupt erfolgreich sein kann, muss es mindestens eine diskrete Stellgröße geben, die die Regelgröße in Richtung der Führungsgröße bewegt.

Den zweitgrößten Einfluss hat die Nebenbedingung (34), die die maximale Anstiegsrate der Stellgröße begrenzt. Es wurde festgestellt, dass dies einen signifikanten Einfluss auf die Stabilität der Regelung und den erforderlichen Rechenaufwand hat.

Der drittgrößte Einfluss ist die Größe des Rauschens. Wenn das Rauschen zu groß ist, kann die Regel-KI die zeitlichen Zusammenhänge zwischen Stellgröße und Regelgröße nicht ausreichend lernen, wodurch die Regelung instabil wird.

Der viertgrößte Einfluss ist die Differenz zwischen dem Prognosehorizont und dem Regelhorizont. Je größer die Differenz zwischen Prognosehorizont und Regelhorizont ist, desto größer wird der Vorhersagefehler, insbesondere für weiter in der Zukunft liegende Zeitpunkte.

Der fünftgrößte Einfluss ist die Größe des Regelintervalls  $t_{act}$  der Regel-KI. Je größer das Regelintervall ist, desto seltener kann die Stellgröße geändert werden, was zu größeren Schwankungen führen kann. Ein größeres Regelintervall ermöglicht aber auch ein größeres Zeitfenster, was bei stark verzögertem Übertragungsverhalten von Vorteil sein kann.

Den sechstgrößten Einfluss hat das impulsartige und dynamische Störverhalten. Es wurde beobachtet, dass sich die Regel-KI erfolgreich an impulsartige Störungen anpassen kann. Die Anpassungsfähigkeit hängt dabei von der Häufigkeit und der Größe der auftretenden Störimpulse ab. Eine Anpassung an dynamische Störungen ist ebenfalls möglich, wenn die Bedingung (43) in jedem Fall erfüllt ist.

Der siebtgrößte Einfluss ist die Größe des Trainingsintervalls, die nur einen Einfluss darauf hat, wie schnell die Regelung die Führungsgröße erreicht.

Den geringsten Einfluss auf den Regelungserfolg hat die Anzahl der Werte aus der Vergangenheit, die für die Vorhersage verwendet werden. Es wurde festgestellt, dass die Regelung selbst bei  $n = 1$  erfolgreich ist.

Die Ergebnisse beziehen sich auf das in dieser Arbeit verwendete Proxy-System und können für andere Systeme unterschiedlich sein.

## 4.3 Regelung eines dynamischen Systems

In diesem Kapitel wird die Anpassungsfähigkeit der Regel-KI bei dynamischen Änderungen der Systemeigenschaften untersucht. Dabei wird zum einen die Reaktion der Regel-KI auf eine Änderung der Führungsgröße untersucht. Zum anderen wird die Anpassungsfähigkeit der Regel-KI getestet, indem die internen Zustandsgrößen  $k_1$  und  $k_2$  des Systems über die Zeit verändert werden.

### 4.3.1 Änderung der Führungsgröße

Die Regel-KI sollte sich möglichst schnell an eine Änderung der Führungsgröße anpassen können. Dazu wird im folgenden Abschnitt eine Simulation betrachtet, bei der die Führungsgröße zu diskreten Zeitpunkten sprunghaft geändert wird.

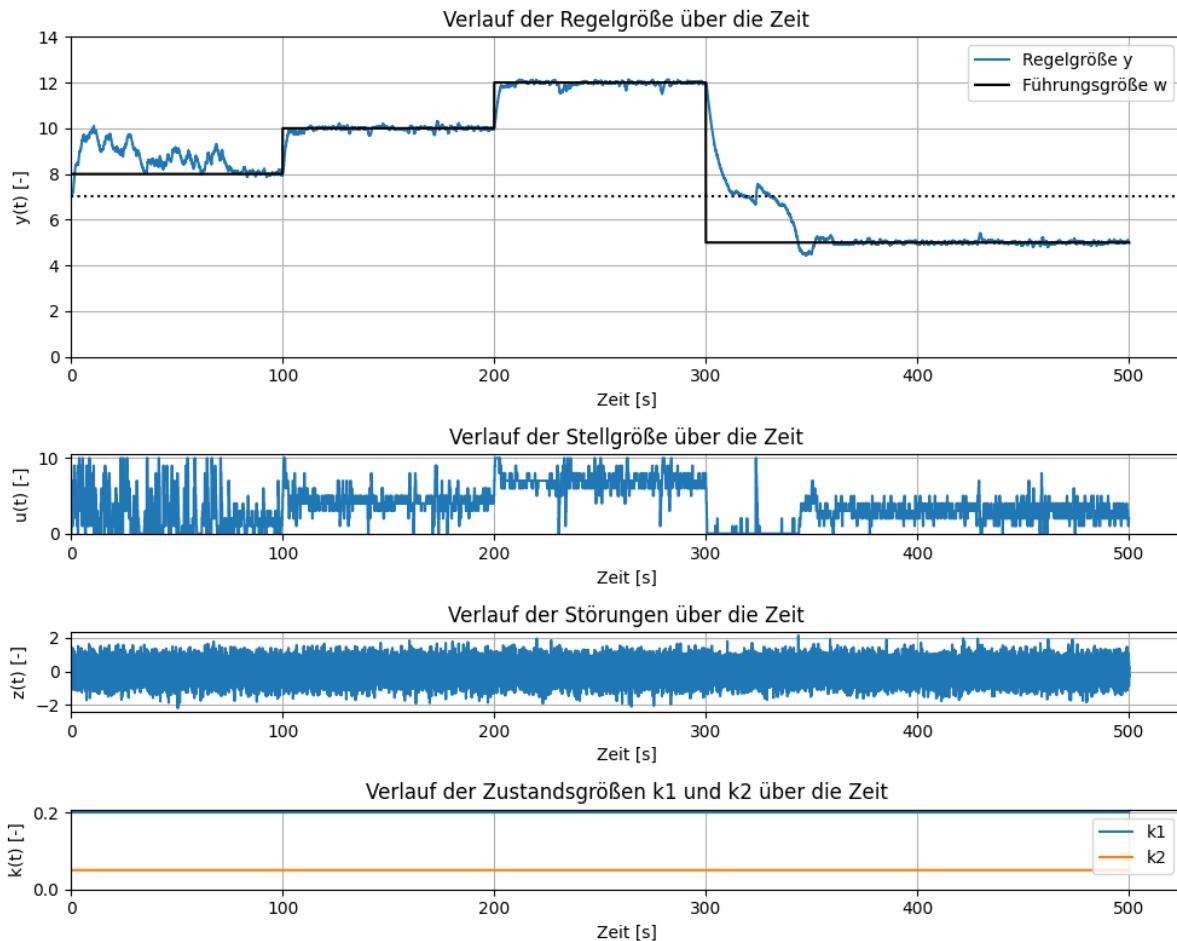


Abbildung 43: Simulationsverlauf bei Änderung der Führungsgröße

In Abbildung 43 ist der Verlauf einer Simulation dargestellt, bei der die Führungsgröße zu den Zeitpunkten  $t = 100\text{s}$ ,  $t = 200\text{s}$  und  $t = 300\text{s}$  sprunghaft auf 10, 12 und auf 5 verändert wird. Die Simulation basiert auf den in Kapitel 4.2 verwendeten Parametern unter zusätzlicher Berücksichtigung von  $\Delta u_{max} = 2$  und der Fenstergröße  $n = 4$ ,  $m = 2$ ,  $h = 4$ ,  $t_{act} = 200\text{ms}$ ,  $t_{train} = 10\text{s}$ . Wie aus der Abbildung ersichtlich ist, kann sich die Regel-KI nach kurzer Zeit erfolgreich an jede Änderung der Führungsgröße anpassen. Dies deutet darauf hin, dass die Regel-KI den zeitlichen Zusammenhang zwischen Stellgröße und Regelgröße lernt. Es ist jedoch zu beachten, dass die Erfolgsrate der Anpassung der Regel-KI an eine Änderung der Führungsgröße von der Komplexität des Systems abhängt. Bei einem komplexeren System kann es vorkommen, dass eine Anpassung nicht erfolgreich ist.

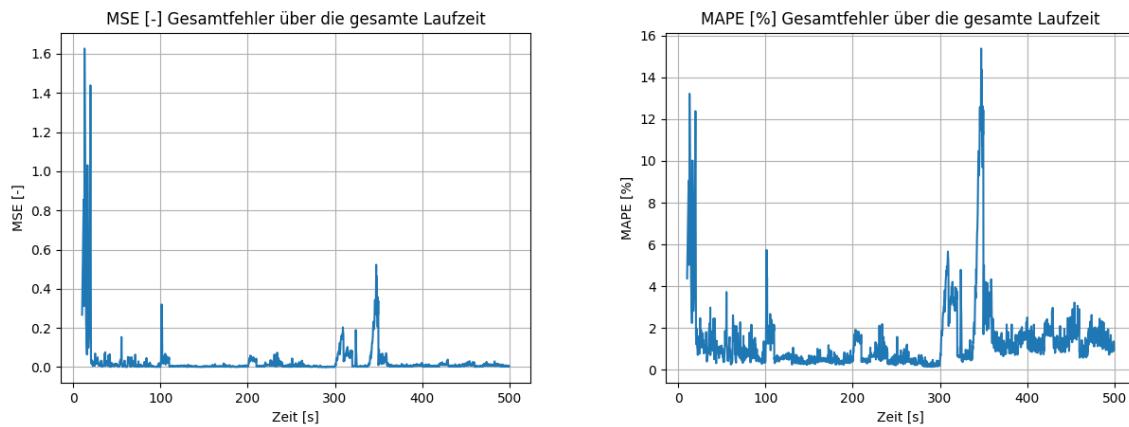


Abbildung 44: MSE und MAPE bei Änderung der Führungsgröße

Aus Abbildung 44 ist ersichtlich, dass der Gesamtfehler zu den Zeitpunkten, an denen die Führungsgröße geändert wird, ansteigt und kurz danach wieder abfällt. Dies ist ein Indikator dafür, dass die Regel-KI eine gewisse Zeit benötigt, um den neuen Systemzustand zu erlernen. In jedem Fall geht der Vorhersagefehler nach einer Änderung der Führungsgröße wieder zurück.

#### 4.3.2 Dynamische Zustandsgrößen $k_1$ und $k_2$

Die Regel-KI sollte außerdem in der Lage sein, sich an dynamische Änderungen der Systemeigenschaften anzupassen. Um diese Anpassungsfähigkeit zu testen, wird in diesem Abschnitt das Regelverhalten bei einer unvorhersehbaren und einer vorhersehbaren, dynamischen Änderung von  $k_1$  und  $k_2$  untersucht.

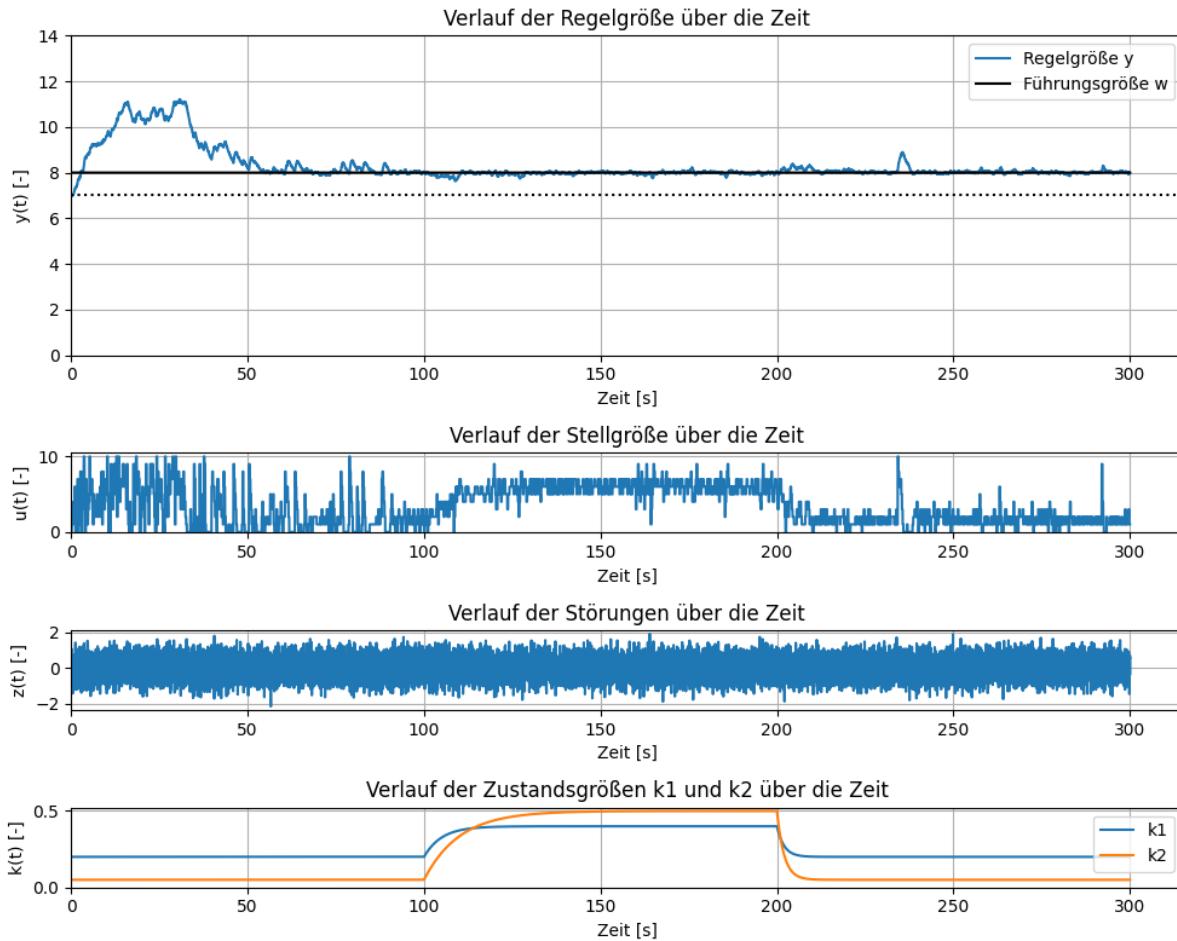


Abbildung 45: Simulationsverlauf bei verzögerter Sprungänderung von  $k_1$  und  $k_2$

Um eine unvorhersehbare Änderung der Systemeigenschaften zu untersuchen, werden  $k_1$  und  $k_2$  zu diskreten Zeitpunkten geändert. Diese Änderung wird durch ein PT<sub>1</sub>-Glied verzögert. Abbildung 45 zeigt den Verlauf der Simulation für eine Änderung der Zustandsgrößen  $k_1$  und  $k_2$  zu den Zeitpunkten  $t = 100\text{s}$  und  $t = 200\text{s}$ . Zum Zeitpunkt  $t = 100\text{s}$  wird  $k_1$  auf 0,4 und  $k_2$  auf 0,5 gesetzt. Zum Zeitpunkt  $t = 200\text{s}$  werden  $k_1$  und  $k_2$  wieder auf ihre ursprünglichen Werte gesetzt. Die verzögerte Sprungantwort des PT<sub>1</sub>-Glieds ist in diesem Fall

$$k(t) = k_{neu} + (k_{alt} - k_{neu}) e^{-\frac{t}{T}} \quad (45)$$

Für den ersten Sprung wird die Verzögerungszeit  $T$  für die Änderung von  $k_1$  auf 5s und für die Änderung von  $k_2$  auf 10s festgelegt. Der Wert von  $k_2$  wird also länger verzögert.

Beim Zurücksetzen der Werte werden beide mit einer Verzögerung von  $T = 2s$  geändert, um die Reaktion der Regel-KI auf plötzliche Änderungen zu untersuchen. Abbildung 45 zeigt, dass die Regel-KI in der Lage ist, sich an eine unvorhersehbare Änderung der internen Zustandsgrößen anzupassen und die Führungsgröße weiterhin zu halten. Bei dynamischen Änderungen des Systemzustandes muss weiterhin darauf geachtet werden, dass die Bedingung (43) erfüllt ist. In diesem Fall ergibt sich durch die Änderung von  $k_1$  auf 0,4 und  $k_2$  auf 0,5 und  $k_s = 0,15$  die Bedingung  $u_{max} > 6 > u_{min}$ , die mit  $U = \{0,1, \dots, 10\}$  erfüllt ist.

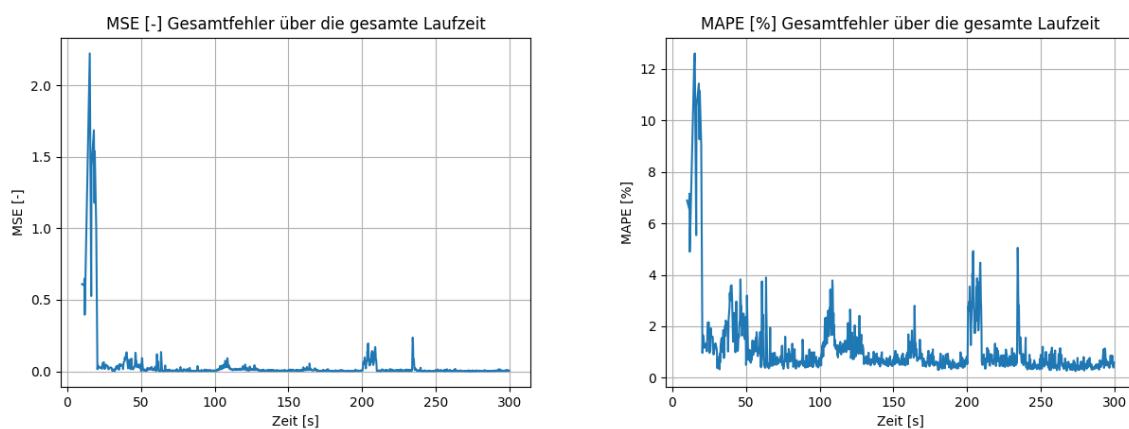


Abbildung 46: MSE und MAPE bei verzögterer Sprungänderung von  $k_1$  und  $k_2$

Abbildung 46 zeigt, dass der Vorhersagefehler, nach der Änderung von  $k_1$  und  $k_2$  bei  $t = 100s$  und  $t = 200s$  kurz ansteigt und nach einigen Sekunden wieder auf den ursprünglichen Wert absinkt. Dies deutet darauf hin, dass sich die Regel-KI an eine unvorhersehbare dynamische Änderung der Systemeigenschaften anpassen kann.

Neben unvorhersehbaren Änderungen des Systemzustands kann es auch vorhersehbare Änderungen geben, zum Beispiel wenn eine Zustandsgröße periodisch verläuft. Wenn sich die Regel-KI erfolgreich an eine vorhersehbare Änderung anpassen kann, ist zu erwarten, dass der Vorhersagefehler konstant bleibt und keine großen Schwankungen aufweist.

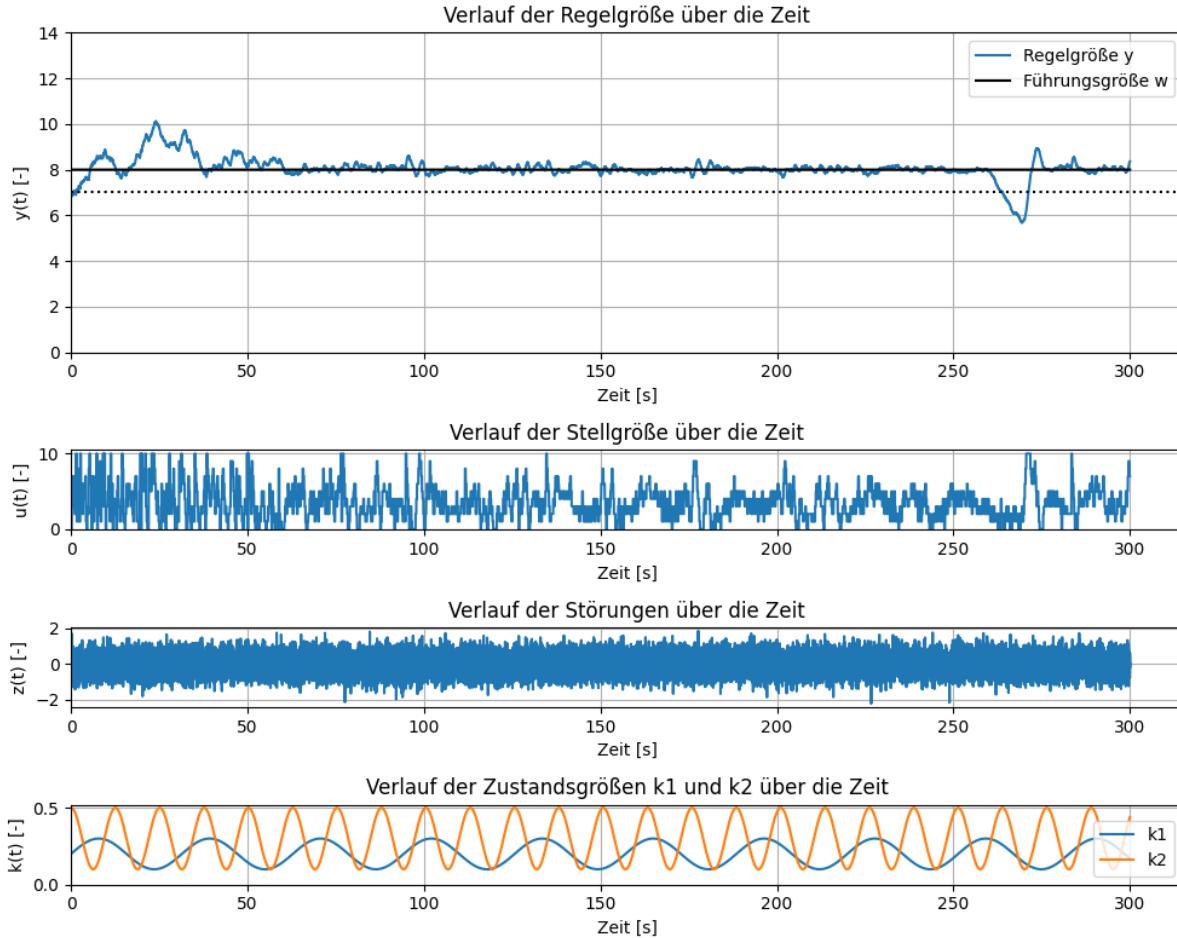


Abbildung 47: Simulationsverlauf bei periodischer Veränderung von  $k_1$  und  $k_2$

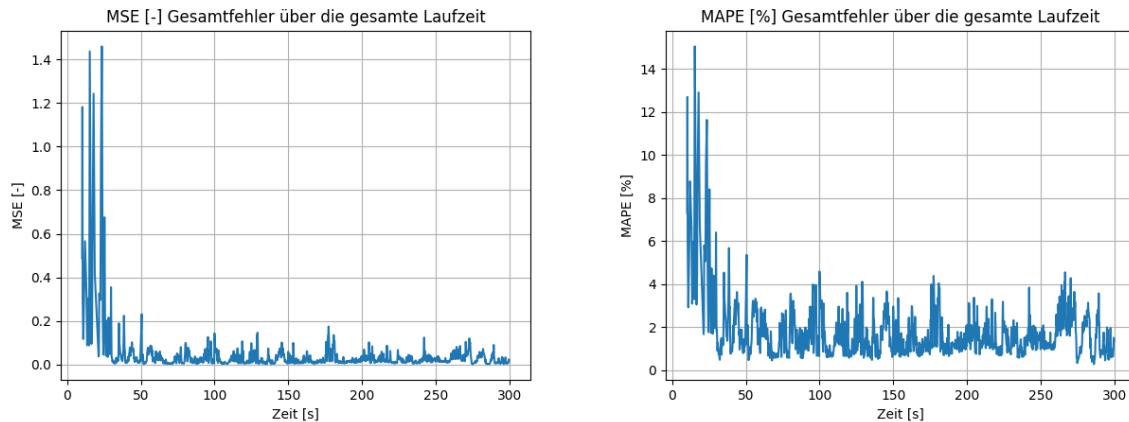
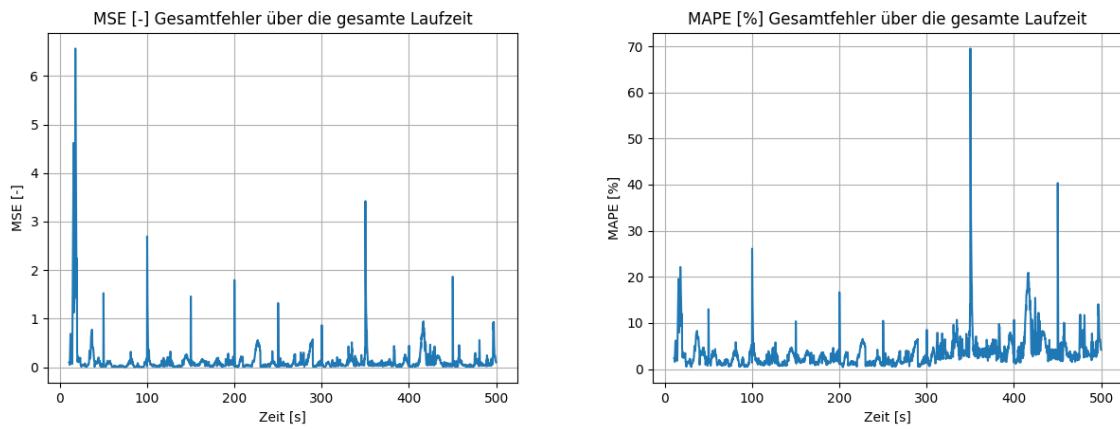


Abbildung 48: MSE und MAPE bei periodischer Veränderung von  $k_1$  und  $k_2$

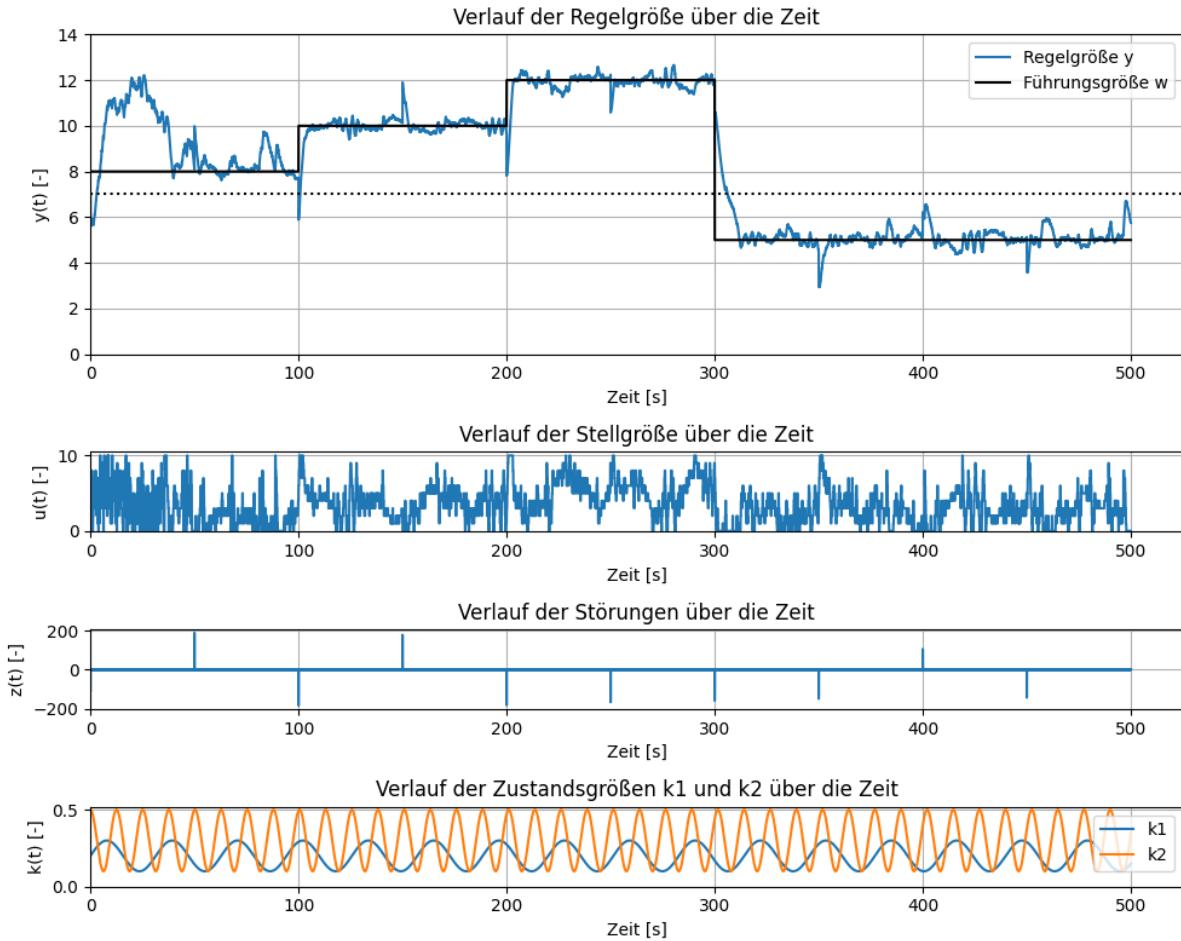
Abbildung 47 zeigt den Verlauf einer Simulation, bei der  $k_1$  durch eine Sinusfunktion und  $k_2$  durch eine Cosinusfunktion dargestellt wird. Aus der Abbildung ist zu erkennen, dass die Führungsgröße erreicht wird und bis auf einen Ausschlag bei  $t = 260s$  stabil gehalten wird. Auch nach einem großen Ausschlag ist die Regel-KI in der Lage, die Regelgröße zu stabilisieren. Im Vergleich zu einer unvorhersehbaren Änderung von  $k_1$  und  $k_2$  ist der Vorhersagefehler in Abbildung 48 insgesamt volatiler, was darauf hindeutet, dass sich die Regel-KI nicht perfekt an komplexe periodische Änderungen anpassen kann. Eine Variation der Fensterparameter hat in diesem Fall keinen signifikanten Einfluss auf den Vorhersagefehler. Damit sich die Regel-KI erfolgreich an eine dynamische Änderung der internen Zustandsgrößen anpassen kann, ist es sinnvoll diese zu messen und mit als Eingabeparameter für das KNN zu verwenden. Dadurch ist die Regel-KI in der Lage komplexere zeitliche Zusammenhänge zwischen den einzelnen Größen zu erkennen.

#### 4.3.3 Änderung der Führungsgröße und dynamische Zustandsgrößen

In diesem Abschnitt wird die Änderung der Führungsgröße mit dem periodischen Verhalten von  $k_1$  und  $k_2$  kombiniert. Zusätzlich werden alle Störeinflüsse aus Kapitel 4.2.3 (Rauschen, Impulsartig, Dynamisch) berücksichtigt. Die Standardabweichung des Rauschens beträgt weiterhin 0,5. Das System wird alle 50 Sekunden mit einem Störimpuls mit zufälliger Größe angeregt und das dynamische Störverhalten wird durch die Funktion (44) mit  $A = 0,2$  bestimmt.



**Abbildung 49: MSE und MAPE bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für  $n = 4, m = 3, h = 4$**

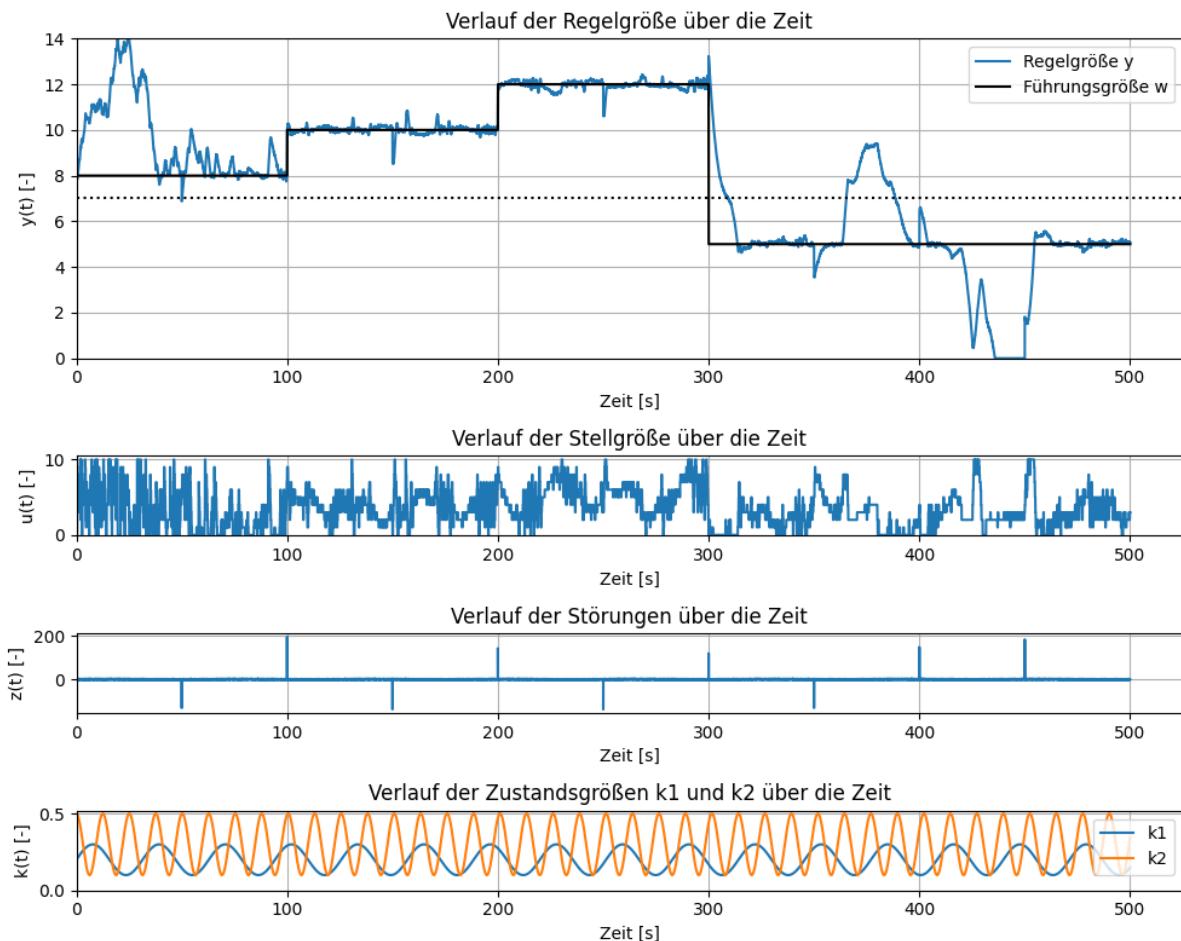


**Abbildung 50: Simulationsverlauf bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für  $n = 4, m = 3, h = 4$**

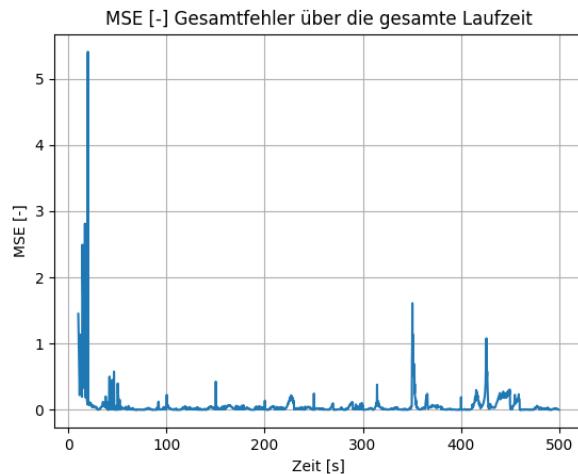
In Abbildung 50 ist der Verlauf der Simulation dargestellt, wobei die Änderung der Führungsgröße dem in Abschnitt 4.3.1 und der periodische Verlauf von  $k_1$  und  $k_2$  dem in Abschnitt 4.3.2 entspricht. Die einzige Änderung ist der Wert von  $k_s$ , der von 0,15 auf 0,25 erhöht wird, damit die maximale Führungsgröße von 12 für den Fall  $k_{1,max} = 0,3, k_{2,max} = 0,5$  und  $z_{d,max} = 0,2$  gehalten werden kann, beziehungsweise damit (43) erfüllt ist. Ansonsten würde in diesem Fall der maximale Wert für die Stellgröße  $u_{max} = 10$  nicht ausreichen um die Führungsgröße zu halten. Abbildung 49 zeigt, dass der Vorhersagefehler volatil ist und zu den Zeitpunkten, an denen das System mit einem Störimpuls angeregt wird, starke Ausschläge aufweist. Für die meisten Vorhersagen, liegt der Fehler mit MSE unter 0,2 und mit MAPE unter 5%. Dies

deutet darauf hin, dass die Regel-KI die zeitlichen Zusammenhänge zwischen Stell-, Stör-, Zustands- und Regelgröße ausreichend lernt und somit eine erfolgreiche Regelung mit einigen Schwankungen ermöglicht.

In einem dynamischen System hat die Wahl der Fenstergröße einen großen Einfluss auf die Stabilität der Regelung. Ist das Zeitfenster zu klein, werden die dynamischen Zusammenhänge zwischen den einzelnen Größen nicht ausreichend erfasst.



**Abbildung 51: Simulationsverlauf bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für  $n = 1, m = 1, h = 2$**



**Abbildung 52: MSE bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für  $n = 1, m = 1, h = 2$**

Anhand von Abbildung 51 wird deutlich, dass die Regelung mit einem kleinen Zeitfenster instabil werden kann und zeitweise komplett versagt. Es ist zu erkennen, dass die Führungsgröße bis  $t = 300s$  erreicht und gehalten wird, es jedoch beim Halten von  $w(t) = 5$  zu hohen Ausschlägen kommt, was daran liegt, dass zu wenig vergangene Werte für die Vorhersage benutzt werden. Abbildung 52 zeigt, dass der MSE-Vorhersagefehler für ein kleineres Zeitfenster kleiner und konstanter ist als für ein größeres Zeitfenster. Der MAPE-Fehler kann in diesem Fall nicht verwendet werden, da die Regelgröße 0 erreicht. Der geringe Vorhersagefehler ist darauf zurückzuführen, dass mit  $m = 1$  und  $\Delta u_{max} = 2$  wenige Regelstrategien vorhergesagt werden müssen, was zu weniger fehlerhaften Vorhersagen führt. Trotz des geringeren Vorhersagefehlers ist die Regelung mit einem kleinen Zeitfenster insgesamt instabiler. Es sollte daher darauf geachtet werden, dass das Zeitfenster, je nach Anwendungsfall der Regel-KI, nicht zu klein gewählt wird.

#### 4.4 Übertragung auf einen realen Prozess

Eine Übertragung dieses Regelungssystems auf reale Prozesse sollte mit Vorsicht erfolgen, da die Effektivität der Regel-KI in diesen Fällen noch nicht getestet wurde. Das in dieser Arbeit verwendete virtuelle Proxysystem ist im Vergleich zu den meisten realen Prozessen nicht sehr komplex, was eine erfolgreiche Regelung erleichtert. Die Regel-KI ist ungeeignet für Prozesse, die leicht instabil werden und möglicherweise komplett versagen können, da das Lernen der Regel-KI auf der Erkundung des Systems durch zufällige Aktionen basiert. Ein Beispiel hierfür wäre die Anwendung dieser Regel-KI zur Steuerung von Steuerelementen in einem Flugzeug, was aufgrund der hohen Anforderungen an Sicherheit und Zuverlässigkeit nicht geeignet ist.

Die implementierte Regel-KI kann unter Berücksichtigung eventueller Anpassungen auf einen Bioreaktor zur pH-Wert-Regelung übertragen werden. Um ein besseres Prozessmodell zu erhalten, ist es sinnvoll, neben der Regelgröße und der Stellgröße alle relevanten Größen, die einen Einfluss auf den pH-Wert haben, zu messen und als Eingangsgrößen für die Vorhersage zu verwenden. Zur Einhaltung von Sicherheitsbedingungen kann gegebenenfalls ein separater Regler installiert werden, der bei zu großen Schwankungen kurzzeitig die Regelung übernimmt, um Schäden am Bioreaktor durch zu hohe oder zu niedrige pH-Werte zu vermeiden. Andernfalls können weitere Sicherheitsmechanismen eingebaut werden, wie zum Beispiel das Abschalten des Zulaufventils, wenn der pH-Wert einen bestimmten Grenzwert überschreitet. Es sollte auch darauf geachtet werden, ein ausreichend großes Zeitfenster zu wählen, um die Diffusion des Zulaufs zu berücksichtigen.

Durch die Verwendung eines KNN als Modell für MPC, können komplexe Zusammenhänge und nichtlineare Beziehungen zwischen Stell- und Regelgröße ohne vorherige Kenntnis der Systemeigenschaften erfasst werden. KNNs sind zudem robust gegenüber Änderungen der Prozessparameter und können sich an dynamische Prozesse anpassen. Es besteht jedoch das Risiko, dass sich ein KNN zu sehr an alte Daten anpasst und neue Systemzustände nicht generalisieren kann, was zum Versagen der Regelung führen kann. Außerdem ist der Rechenaufwand bei der

Verwendung eines KNN als Modell höher als bei statistischen Modellen, was zu höheren Energiekosten für die Regelung führen kann. Im Hinblick auf die Übertragbarkeit auf reale Prozesse ist auch die Wirtschaftlichkeit zu berücksichtigen. Die Anpassungsfähigkeit einer Regel-KI auf unterschiedliche Systeme kann zwar die Kosten für den Entwurf eines spezifischen Reglers einsparen, erfordert aber den Einsatz teurerer Rechner, die zudem einen höheren Energiebedarf haben. Für Prozessregelungen, die mit älteren Rechnern betrieben werden, ist die Übertragung dieser Regel-KI wahrscheinlich ungeeignet, da in diesen Fällen die Rechenleistung fehlt, um eine große Anzahl von Regelstrategien in Echtzeit zu bewerten und gleichzeitig das KNN kontinuierlich mit neuen Daten zu trainieren.

## 5 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine adaptive Regel-KI implementiert, die mit Hilfe eines gleitenden Zeitfensters die zeitlichen Zusammenhänge zwischen der Stell- und Regelgröße eines Systems lernt und aus möglichen Regelstrategien die Optimale identifizieren kann. Dazu wurden zunächst die theoretischen Grundlagen der Prozessregelung, der KI und des verwendeten Regelungsverfahrens MPC erläutert. Anschließend wurde die Regel-KI in Python implementiert und an einem virtuellen Stellvertretersystems getestet.

Es wurde festgestellt, dass die Verwendung der ReLU-Aktivierungsfunktion zu toten Neuronen führen kann. Um dies zu vermeiden, wurde die Leaky-ReLU-Funktion verwendet. Weiterhin wurde beobachtet, dass die Größe des Zeitfensters bei diesem System keinen großen Einfluss auf den Regelungserfolg hat. Um den Rechenaufwand zu reduzieren und die Stabilität der Regelung zu erhöhen, wurde bei der Berechnung der optimalen Regelstrategie eine Nebenbedingung eingeführt, die die Änderung der Stellgröße zwischen zwei Zeitpunkten begrenzt. Es konnte beobachtet werden, dass die Regel-KI in der Lage ist, sich nach Störungen im System wieder zu stabilisieren. Zusätzlich wurde die Anpassungsfähigkeit der Regel-KI an eine Änderung der Führungsgröße bei dynamischen Änderungen der internen Zustandsgrößen und unter Einfluss verschiedener Störgrößen getestet. Dabei wurde festgestellt, dass die Regel-KI in der Lage ist, sich an dynamische Änderungen anzupassen und die Führungsgröße mit nur geringen Schwankungen stabil zu erreichen und zu halten. Den größten Einfluss auf eine erfolgreiche Regelung haben die Prozessparameter des Proxysystems, die so gewählt werden müssen, dass eine erfolgreiche Regelung überhaupt erst möglich ist. Im Vergleich zu einem statischen System, kommt der Größe  $n$  in einem dynamischen System eine größere Bedeutung zu, da zeitliche Zusammenhänge in einem dynamischen System mit kleinem  $n$  nur bedingt gelernt werden.

Diese Ergebnisse zeigen das Potential für die Integration der Regel-KI in bestimmte physikalische Prozesse. Für eine Regelung in Echtzeit ist es wichtig, dass genügend

Rechenleistung zur Verfügung steht. Ein Grafikprozessor (GPU) mit hoher Speicherkapazität, eine CPU mit mehreren Kernen und ein ausreichend großer Arbeitsspeicher für die Speicherung der Zeitreihen sollten verwendet werden. Der Grafikprozessor spielt die wichtigste Rolle, da er die parallele Ausführung vieler rechenintensiver Matrixoperationen ermöglicht, was eine schnelle Vorhersage und ein schnelles Training des KNN gewährleistet.

Es ist jedoch zu beachten, dass weitere Tests und Analysen durchgeführt werden müssen, um die Wirksamkeit und die Anwendbarkeit der Regel-KI in realen Prozessen zu bestätigen. Zu diesem Zweck kann die Regel-KI in einer zukünftigen Arbeit an einem Bioreaktor zur pH-Wert-Regelung eingesetzt und getestet werden, wobei mögliche Anpassungen durch den Einbau zusätzlicher Sicherheitsbedingungen zu berücksichtigen sind. Darüber hinaus kann getestet werden, ob ein LSTM-Netz für die Vorhersage einer Zeitreihe besser geeignet ist als ein normales DNN, da LSTM-Netze für die Vorhersage sequentieller Daten optimiert sind.

## 6 Literaturverzeichnis

- [1] Corriou, J.-P., *Process Control: Theory and Applications*, 2nd edn., Springer International Publishing; Imprint: Springer, Cham, 2018.
- [2] Hewing, L., Wabersich, K. P., Menner, M., Zeilinger, M. N., Learning-Based Model Predictive Control: Toward Safe Learning in Control. *Annu. Rev. Control Robot. Auton. Syst.* 2020, 3, 269–296, DOI: 10.1146/annurev-control-090419-075625.
- [3] Grimble, Michael J., Johnson, Michael A., Camacho, E. F., Bordons, C., *Model Predictive control*, Springer London, London, 2007.
- [4] Lunze, J., *Regelungstechnik 1: Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen*, 5th edn., Springer, Berlin, 2006.
- [5] Horn, J., Strukturbild und Übertragungsglieder, in: Plaßmann, W., Schulz, D. (Ed.). *Handbuch Elektrotechnik*, Springer Fachmedien Wiesbaden, Wiesbaden, 2016, pp. 847–852.
- [6] John McCarthy, Marvin L. Minsky, Nathaniel Rochester, and Claude E. Shannon, A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence.
- [7] Gugerty, L., Newell and Simon's Logic Theorist: Historical Background and Impact on Cognitive Modeling.
- [8] Thon, C., Finke, B., Kwade, A., Schilde, C., Artificial Intelligence in Process Engineering. *Advanced Intelligent Systems* 2021, 3, 2000261, DOI: 10.1002/aisy.202000261.
- [9] Samuel, A. L., Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. & Dev.* 1959, 3, 210–229, DOI: 10.1147/rd.33.0210.
- [10] Ayon Dey, Machine Learning Algorithms: A Review. *International Journal of Computer Science and Information Technologies* 2016, 1174–1179.
- [11] Alzubi, J., Nayyar, A., Kumar, A., Machine Learning from Theory to Algorithms: An Overview. *J. Phys.: Conf. Ser.* 2018, 1142, 12012, DOI: 10.1088/1742-6596/1142/1/012012.
- [12] Steven L. Brunton, J. Nathan Kutz, Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control.

- [13] Lorberfeld, A., Machine Learning Algorithms In Layman's Terms, Part 1. *Towards Data Science*, 2 March 2019. <https://towardsdatascience.com/machine-learning-algorithms-in-laymans-terms-part-1-d0368d769a7b>. Accessed 8 December 2022.
- [14] Cohen, G., Afshar, S., Tapson, J., van Schaik, A., EMNIST: Extending MNIST to handwritten letters, in: *2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2017, pp. 2921–2926.
- [15] Sutton, R. S., Barto, A., *Reinforcement learning: An introduction*, The MIT Press, Cambridge, Massachusetts, London, England, 2018.
- [16] Blackburn, Reinforcement Learning Markov-Decision Process (Part 1). *Towards Data Science*, 18 July 2019. <https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da>. Accessed 9 December 2022.
- [17] Kaelbling, L. P., Littman, M. L., Moore, A. W., Reinforcement Learning: A Survey. *jair* 1996, 4, 237–285, DOI: 10.1613/jair.301.
- [18] Christoph Thon, Marvin Röhl, Somayeh Hosseinhosseini, Arno Kwade, Carsten Schilde, Artificial Intelligence and Evolutionary Approaches in Particle Technology. *KONA*.
- [19] Anders Krogh, What are artificial neural networks? *computational Biology*.
- [20] Sydenham, P. H., Thorn, R., *Handbook of measuring system design: 129: Artificial Neural Networks*, Wiley, Chichester England, 2005.
- [21] Sharma, S., Sharma, S., Athaiya, A., Activation Functions In Neural Networks. *IJEAST* 2020, 04, 310–316, DOI: 10.33564/ijeast.2020.v04i12.054.
- [22] Nielsen, M. A., *Neural Networks and Deep Learning*, Determination Press, 2015.
- [23] Agarap, A. F., *Deep Learning using Rectified Linear Units (ReLU)*, 2018.
- [24] Saurabh, Backpropagation – Algorithm For Training A Neural Network. *Edureka*, 7 December 2017. <https://www.edureka.co/blog/backpropagation/>. Accessed 12 December 2022.
- [25] Google Developers, Classification: True vs. False and Positive vs. Negative &nbsp;|&nbsp; Machine Learning &nbsp;|&nbsp; Google Developers.

<https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative?hl=en>. Accessed 13 December 2022.

[26] Google Developers, Classification: Accuracy &nbsp;|&nbsp; Machine Learning &nbsp;|&nbsp; Google Developers. <https://developers.google.com/machine-learning/crash-course/classification/accuracy?hl=en>. Accessed 13 December 2022.

[27] Google Developers, Classification: Precision and Recall &nbsp;|&nbsp; Machine Learning &nbsp;|&nbsp; Google Developers. <https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall?hl=en>. Accessed 13 December 2022.

[28] Hyndman, R. J., Koehler, A. B., Another look at measures of forecast accuracy. *International Journal of Forecasting* 2006, 22, 679–688, DOI: 10.1016/j.ijforecast.2006.03.001.

[29] M.V. Shcherbakov, A. Brebels, N.L. Shcherbakova, A.P. Tyukov *et al.*, A survey of forecast error measures. *World Applied Sciences Journal* 2013, 24, 171–176, DOI: 10.5829/idosi.wasj.2013.24.itmies.80032.

[30] Bontempi, G., Ben Taieb, S., Le Borgne, Y.-A., Machine Learning Strategies for Time Series Forecasting, in: Aufaure, M.-A., Zimányi, E. (Ed.). *Business Intelligence*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 62–77.

[31] H.S. Hota, Richa Handa, A.K. Shrivastava, Time Series Data Prediction Using Sliding Window Based RBF Neural Network. *International Journal of Computational Intelligence Research* 2017.

[32] Thomas G. Dietterich, Machine Learning for Sequential Data: A Review.

[33] Taieb, S. B., Bontempi, G., Atiya, A., Sorjamaa, A., *A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition*, 2011.

[34] K. S. Holkar, L. M. Waghmare, An Overview of Model Predictive Control. *International Journal of Control and Automation International Journal of Control and Automation* 2010.

- [35] Brownlee, J., Time Series Forecasting as Supervised Learning. *Machine Learning Mastery*, 4 December 2016. <https://machinelearningmastery.com/time-series-forecasting-supervised-learning/>. Accessed 1 December 2022.
- [36] Brownlee, J., How to Convert a Time Series to a Supervised Learning Problem in Python. *Machine Learning Mastery*, 7 May 2017. <https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/>. Accessed 1 December 2022.
- [37] Team, K., Keras documentation: About Keras. <https://keras.io/about/>. Accessed 12 January 2023.
- [38] GeeksforGeeks, Deque in Python - GeeksforGeeks. <https://www.geeksforgeeks.org/deque-in-python/>. Accessed 9 January 2023.
- [39] GeeksforGeeks, Epsilon-Greedy Algorithm in Reinforcement Learning - GeeksforGeeks. <https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/>. Accessed 12 January 2023.
- [40] ECOOP '95 - object-oriented programming: 9th European conference, Århus, Denmark, August 7 - 11, 1995 ; proceedings, Springer, 1995.
- [41] Jastram, A., Langschwager, F., Kragl, U., Reaktoren für spezielle technisch-chemische Prozesse: Biochemische Reaktoren, in: Reshetilowski, W. (Ed.). *Handbuch Chemische Reaktoren*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2019, pp. 1–39.
- [42] ML | Underfitting and Overfitting. GeeksforGeeks, 23 November 2017. <https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/>. Accessed 21 January 2023.

## 7 Abbildungsverzeichnis

Abbildung 1: Blockdiagramm eines Prozesses .....	12
Abbildung 2: Struktur eines Regelkreises. Erstellt auf Basis von [4].....	13
Abbildung 3: Sprungantwort eines Systems zweiter Ordnung [4] .....	15
Abbildung 4: Darstellung des Rechteckimpuls und des Dirac-Impuls [4].....	16
Abbildung 5: Impulsantwort eines Systems zweiter Ordnung [4] .....	16
Abbildung 6: Sprungantwort von schwingfähigen PT <sub>2</sub> -Gliedern mit kleiner Dämpfung [4] .....	18
Abbildung 7: Anzahl der Veröffentlichungen von verschiedenen ML-Methoden auf Scopus und qualitative Darstellung der jeweiligen KI-Epochen über die Zeit [8] .....	20
Abbildung 8: Klassen des maschinellen Lernens mit den jeweiligen Algorithmen. Erstellt auf Basis von [8, 12, 13]. .....	22
Abbildung 9: Ablauf vom Reinforcement Learning [16].....	24
Abbildung 10: Neuron eines Säugetiers [20] .....	25
Abbildung 11: Links: Darstellung eines künstlichen neuronalen Netzes. Rechts: Darstellung eines einzelnen Neurons mit den gewichteten Eingaben und der Aktivierungsfunktion. [18].....	26
Abbildung 12: Darstellung der Sigmoid, Tanh und ReLU Aktivierungsfunktion .....	27
Abbildung 13: Einfluss der Gewichtung auf den Fehler. Erstellt auf Basis von [24]..	29
Abbildung 14: Beispiel von einem gleitenden Zeitfenster .....	35
Abbildung 15: Grundlegende Struktur von MPC. Erstellt auf Basis von [3].....	38

Abbildung 16: Vorhersage einer Regelstrategie über ein gleitendes Zeitfenster bei MPC.....	41
Abbildung 17: Ablauf des Hauptprogramms .....	44
Abbildung 18: Struktur eines Zeitfensters zum Zeitpunkt t mit Stell- und Regelgröße .....	47
Abbildung 19: Struktur der Eingabe- und Ausgabeschicht der Regel-KI .....	49
Abbildung 20: Schematische Darstellung eines Bioreaktors in Form eines Rührkessels [41].....	53
Abbildung 21: Einfluss von $k_1$ und $k_2$ auf das System mit $u(t) = 0$ und $z(t) = 0$ .....	54
Abbildung 22: MSE und MAPE für die einzelnen Zeitpunkte, bei dem drei Ausgabe-Neuronen permanent null sind.....	58
Abbildung 23: Simulationsverlauf für eine Regel-KI bei der drei Ausgabe-Neuronen permanent null sind .....	59
Abbildung 24: Simulationsverlauf für $n = 4$ , $m = 2$ , $h = 4$ , $t_{act} = 200\text{ms}$ , $t_{train} = 10\text{s}$ ....	61
Abbildung 25: MSE und MAPE für $n = 4$ , $m = 2$ , $h = 4$ , $t_{act} = 200\text{ms}$ , $t_{train} = 10\text{s}$ .....	61
Abbildung 26: Quadratischer Fehler über alle Regelstrategien und Zeitschritte der ersten Vorhersage für $t = 11,2\text{s}$ .....	63
Abbildung 27: Quadratischer Fehler über alle Regelstrategien und Zeitschritte der letzten Vorhersage für $t = 199,8\text{s}$ .....	63
Abbildung 28: MAPE für $t_{train} = 5\text{s}$ (links) und $t_{train} = 20\text{s}$ (rechts) .....	65
Abbildung 29: Verlauf der Stellgröße für $t_{train} = 5\text{s}$ (oben) und $t_{train} = 20\text{s}$ (unten) ....	65
Abbildung 30: MSE und MAPE für $t_{act} = 500\text{ms}$ .....	66

**Error! Use the Home tab to apply Überschrift 1 to the text that you want to appear here.**

---

here.	100
Abbildung 31: MAPE-Gesamtfehler und MAPE für jeden Zeitpunkt für $h = 8$ und $m = 2$ .....	67
Abbildung 32: MAPE-Gesamtfehler und MAPE für jeden Zeitpunkt für $h = 8$ und $m = 4$ .....	67
Abbildung 33: MAPE-Gesamtfehler für $n = 1$ (links) und $n = 8$ (rechts) .....	68
Abbildung 34: Simulationsverlauf für $k_s = 0,3$ .....	70
Abbildung 35: Simulationsverlauf für ein hohes Rauschen .....	71
Abbildung 36: MSE und MAPE für ein hohes Rauschen .....	71
Abbildung 37: Simulationsverlauf beim Anregen durch Störimpulse .....	72
Abbildung 38: MSE und MAPE beim Anregen durch Störimpulse .....	72
Abbildung 39: Simulationsverlauf bei einem dynamischen Störverhalten mit $A = 0,573$	
Abbildung 40: Simulationsverlauf bei einem dynamischen Störverhalten mit $A = 0,274$	
Abbildung 41: Darstellung von Underfitting und Overfitting [42] .....	75
Abbildung 42: Simulationsverlauf für $\Delta u_{\max} = 2$ .....	76
Abbildung 43: Simulationsverlauf bei Änderung der Führungsgröße .....	80
Abbildung 44: MSE und MAPE bei Änderung der Führungsgröße .....	81
Abbildung 45: Simulationsverlauf bei verzögerter Sprungänderung von $k_1$ und $k_2$ ...	82
Abbildung 46: MSE und MAPE bei verzögerter Sprungänderung von $k_1$ und $k_2$ ....	83
Abbildung 47: Simulationsverlauf bei periodischer Veränderung von $k_1$ und $k_2$ .....	84
Abbildung 48: MSE und MAPE bei periodischer Veränderung von $k_1$ und $k_2$ .....	84

Abbildung 49: MSE und MAPE bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für n = 4, m = 3, h = 4 .....	86
Abbildung 50: Simulationsverlauf bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für n = 4, m = 3, h = 4 .....	87
Abbildung 51: Simulationsverlauf bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für n = 1, m = 1, h = 2 .....	88
Abbildung 52: MSE bei Änderung der Führungsgröße, dynamischer Veränderung der Zustandsgrößen und Einfluss von Störungen für n = 1, m = 1, h = 2.....	89

## **8 Tabellenverzeichnis**

Tabelle 1: Funktionalbeziehungen und Blocksymbole von Proportional Gliedern [4, 5] .....	17
Tabelle 2: Funktionalbeziehung und Blocksymbol vom I-Glied [4, 5].....	18
Tabelle 3: Funktionalbeziehungen und Blocksymbole von Differenziergliedern [4, 5] .....	19
Tabelle 4: Funktionalbeziehung und Blocksymbol vom Totzeitglied [4, 5] .....	20
Tabelle 5: Mögliche Resultate einer binären Klassifizierung.....	31
Tabelle 6: Beispiel einer univariaten Zeitreihe .....	45
Tabelle 7: Beispiel einer Umwandlung einer univariaten Zeitreihe in überwachtes Lernen .....	45

---

**Error! Use the Home tab to apply Überschrift 1 to the text that you want to appear here.**

102

Tabelle 8: Multivariate Zeitreihe mit Stell- und Regelgröße ..... 46

Tabelle 9: Rangfolge der Einflussfaktoren auf den Regelungserfolg ..... 77

## 9 Anhang

### 9.1 Python Skript aus “main.py”

```
from simulation.simulation import SimKeys
from simulation.simulation import Simulation
from simulation.simulation import SimConfig
from export import Data
from control.control_ai import Agent, AgentConfig


def run_simulation(s_config: SimConfig, agent_configs):
    sim_time = 200_000
    sim_steps = int(sim_time / s_config.dt)

    for agent_config in agent_configs:
        sim = Simulation(s_config)
        agent = Agent(agent_config, SimKeys.ACTION, SimKeys.VALUE)
        data = Data(plots_every=50 * pow(10, 3), agent_config=agent_config,
sim_config=s_config)

        target, time = sim.reset()

        for _ in range(sim_steps):
            # Change reference here if needed#
            ##########
            action, predictions = agent.choose_action(target, time,
sim.current_u)

            if predictions is not None:
                # Calculate actual trajectories
                values =
sim.calculate_strategies(agent.get_allowed_strategies(sim.current_u),
agent.h,
                                         agent.act_every)
                data.record_errors(predictions, values, time)

            zi = 0
            # Add disturbance impulse z here if needed #
            ##########
            target_, time, temp = sim.step(action, zi, agent.reference)
            if action is not None:
                agent.record(temp)
                agent.train_model(time)
            target = target_
            data.finish(sim)

def get_agent_configs():
    # Create all the Configs which should be compared

    c1 = AgentConfig(reference=8, action_space=[0, 1, 2, 3, 4, 5, 6, 7, 8,
```

```
9, 10])
c1.n = 4
c1.m = 2
c1.h = 4
c1.act_every = 200
c1.train_every = 10_000

return [c1]

if __name__ == '__main__':
    sim_config = SimConfig()
    sim_config.noise = 0.5
    sim_config.dt = 10 # Time in ms
    sim_config.start_value = 7
    sim_config.ks = 0.15
    sim_config.stable_value = 7

run_simulation(sim_config, get_agent_configs())
```

## 9.2 Python Skript der Regel-KI aus „control\_ai.py“

```
import numpy as np
import control.utils as utils
import pandas as pd
from collections import deque
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, LeakyReLU
from tensorflow.python.keras.optimizer_v2.adam import Adam

class AgentConfig:

    def __init__(self, reference, action_space, du_max=None, sys_lim=None):
        self.max_mem_size = 10_000
        self.n = 2
        # h needs to be greater than m and > 0, m can be 0
        # Control horizon
        self.m = 2
        # Prediction horizon
        self.h = 4
        self.epsilon = 1
        self.epsilon_dec = 0.7
        self.epsilon_min = 0.01
        self.batch_size = 10
        self.epochs = 50
        # Train model every 10 seconds
        self.train_every = 10_000
        # Act every 200ms
        self.act_every = 200
        self.alpha = 0.7
        self.reference = reference
        self.action_space = action_space
        self.du_max = du_max
        self.sys_lim = sys_lim

    def build_network(input_size, output_size, n_hidden1, n_hidden2):
        model = Sequential([
            Dense(n_hidden1, input_shape=(input_size,)),
            LeakyReLU(alpha=0.1),
            Dense(n_hidden2),
            LeakyReLU(alpha=0.1),
            Dense(output_size),
            LeakyReLU(alpha=0.1)
        ])
        model.compile(optimizer=Adam(learning_rate=0.0001), loss='mse')

        return model

class Agent:
    def __init__(self, config: AgentConfig, c_key, t_key):
        assert config.h > config.m
        self.input_size = (config.n + 1) * 2 + config.m
```

```
# System limits (upper limit and lower limit) -> if not None the
# data will be normalized to that range
    self.sys_lim = config.sys_lim
    self.action_space = config.action_space
    self.du_max = config.du_max
    self.alpha = config.alpha
    self.epsilon = config.epsilon
    self.epsilon_dec = config.epsilon_dec
    self.epsilon_min = config.epsilon_min
    self.batch_size = config.batch_size
    self.epochs = config.epochs
    self.reference = config.reference
    self.n = config.n
    self.control_key = c_key
    self.target_key = t_key
    self.m = config.m
    self.h = config.h
    self.train_every = config.train_every
    self.last_trained = 0
    self.act_every = config.act_every
    self.last_acted = 0
    self.memory = deque(maxlen=config.max_mem_size)
    self.training_history = []
    # Temporary memory is cleared after every training iteration
    self.temp_memory = []
    self.model = build_network(self.input_size, self.h, 256, 128)
    self.control_strategies =
utils.compute_all_possible_strategies(self.m, self.action_space,
du_max=self.du_max)

def _get_prediction_data(self, current_y, allowed_strategies):
    # Get the target and control values for the last n time steps from
    # the memory
    data = [[self.memory[i][self.target_key],
    self.memory[i][self.control_key]] for i in
            range(len(self.memory) - self.n, len(self.memory))]
    data = np.array(data, dtype=np.float)

    if self.sys_lim is not None:
        # Normalize Data to system limits
        data[:, 0] = utils.normalize(data[:, 0], self.sys_lim[0],
self.sys_lim[1])
        data[:, 1] = utils.normalize(data[:, 1],
min(self.action_space), max(self.action_space))
        allowed_strategies = utils.normalize(allowed_strategies,
min(self.action_space), max(self.action_space))

        # Flatten array since its currently nested
    data = data.flatten()
    # Add current y at the end of the array
    if self.sys_lim is not None:
        data = np.append(data, utils.normalize(current_y,
self.sys_lim[0], self.sys_lim[1]))
    else:
```

```
    data = np.append(data, current_y)

    # Expand dimensions of array so that the row can be copied
    data = np.expand_dims(data, axis=0)
    # Copy the row to fit the same shape as the control strategies so
    that they can be concatenated
    data = np.repeat(data, repeats=allowed_strategies.shape[0], axis=0)
    data = np.concatenate((data, allowed_strategies), axis=1)

    return data

def get_allowed_strategies(self, current_u):
    if self.du_max is None:
        return self.control_strategies

    # Filter possible control strategies with constraint du <= du_max
    diff = np.abs(current_u - self.control_strategies[:, 0])
    return self.control_strategies[diff <= self.du_max]

def record(self, state):
    # Save current target and control values in the long term and
    temporary memory
    inputs = [self.target_key, self.control_key]
    y = {k: state[k] for k in inputs}
    self.memory.append(y)
    self.temp_memory.append(y)

def choose_action(self, current_y, t, u):
    # Returns next action and all the predictions the ANN made
    if t < self.last_acted + self.act_every:
        return None, None

    self.last_acted = t
    rand = np.random.random()
    if len(self.memory) < self.n or rand < self.epsilon:
        # Pick random action
        action = np.random.choice(self.action_space)
        return action, None

    allowed_strategies = self.get_allowed_strategies(u)
    inputs = self._get_prediction_data(current_y,
    np.copy(allowed_strategies))
    predictions = self.model.predict(inputs, batch_size=4096)
    if self.sys_lim is not None:
        predictions = utils.denormalize(predictions, self.sys_lim[0],
    self.sys_lim[1])

    # Evaluate control strategies and find the best one (minimal cost)
    costs = _cost_function(predictions, self.reference, self.alpha)
    rand = np.random.random()
    if rand <= 0.1:
        # Pick a random strategy of the best 10
        idx = np.argpartition(costs, min(10,
    allowed_strategies.shape[0] - 1))
        min_cost_idx = np.random.choice(idx)
```

```
        else:
            min_cost_idx = np.argmin(costs)

            best_strategy = allowed_strategies[min_cost_idx]
            print('BEST STRATEGY: ', best_strategy, ' COST: ',
costs[min_cost_idx], ' EPSILON: ', self.epsilon)

            # Only use first action of control strategy for the next timestep,
            after that repeat this process
            return best_strategy[0], predictions

    def train_model(self, t):
        if len(self.memory) < self.n + self.h + 1 + self.batch_size or t <
self.last_trained + self.train_every:
            return

        df = pd.DataFrame.from_records(self.temp_memory)

        # Normalize data
        if self.sys_lim is not None:
            df[self.target_key] = utils.normalize(df[[self.target_key]],
self.sys_lim[0], self.sys_lim[1])
            df[self.control_key] = utils.normalize(df[[self.control_key]],
min(self.action_space),
                                         max(self.action_space))

        x_data, y_data = utils.convert_input_data_training(df, self.n,
self.m, self.h, self.control_key,
                                         self.target_key)

        history = self.model.fit(x_data, y_data,
batch_size=self.batch_size, epochs=self.epochs, verbose=0)
        self.training_history.append(history)

        self.epsilon = self.epsilon * self.epsilon_dec if self.epsilon >
self.epsilon_min else self.epsilon_min
        self.temp_memory.clear()
        self.last_trained = t

    def _cost_function(trajectories, reference, alpha):
        """
        Calculate cost of trajectories
        :param trajectories: Predicted trajectories over a given horizon of
len(trajectory)
        :param reference: Value where the system should end up
        :return: cost of all trajectories
        """
        h = trajectories.shape[1]

        # Calculate weights for each element in the trajectory
        weights = np.power(alpha, np.flip(np.arange(h)))
        return np.sum((trajectories - reference) ** 2 * weights, axis=1)
```

### 9.3 Python Skript aus “utils.py”

```
import itertools
import numpy as np
import pandas as pd

def convert_input_data_training(df: pd.DataFrame, n, m, h, c_key, t_key):
    assert h >= m
    assert h >= 1

    inputs = [t_key, c_key]
    cols, names = [], []
    # Input sequence (t-n, ... t-1)
    for i in range(n, -1, -1):
        cols.append(df.shift(i))
        if i == 0:
            names += [f'{x}(t)' for x in inputs]
        else:
            names += [f'%s(t-%d)' % (x, i) for x in inputs]

    # Input sequence (u_t, u_{t+1} ..., u_{t+m})
    for i in range(1, m + 1):
        cols.append(df[c_key].shift(-i))
        names += ['%s(t+%d)' % (c_key, i)]

    target_index = len(names)

    # Output sequence (t+1, ... t+h)
    for i in range(1, h + 1):
        cols.append(df[t_key].shift(-i))
        names += ['%s(t+%d)' % (t_key, i)]

    # Put it all together
    data = pd.concat(cols, axis=1)
    data.columns = names
    # Drop rows with NaN values
    data.dropna(inplace=True)
    data.reset_index(drop=True, inplace=True)
    data = data.astype(float)

    return data.iloc[:, :target_index].copy(), data.iloc[:, target_index:].copy()

def compute_all_possible_strategies(m, action_space: list, du_max=None):
    strategies = np.array(list(itertools.product(action_space, repeat=m + 1)), dtype=np.float)

    if du_max is not None:
        # Filter out strategies that dont match the constraint
        diff = np.abs(np.diff(strategies, axis=1))
        strategies = strategies[np.max(diff, axis=1) <= du_max]

    return strategies
```

```
def normalize(x, min_x, max_x):  
    return (x - min_x) / (max_x - min_x)  
  
def denormalize(y, min_x, max_x):  
    return y * (max_x - min_x) + min_x
```

## 9.4 Python Skript der Simulation aus “simulation.py“

```
import numpy as np
from scipy.integrate import odeint

class SimKeys:
    TIME = "time"
    VALUE = "value"
    CHANGE = "change"
    DELTA = "delta"
    ACTION = "action"
    REFERENCE = "reference"
    K1 = "k1"
    K2 = "k2"
    Z = "z"

def k1_function(t):
    # if t < 100:
    #     return 0.2
    # elif t < 200:
    #     return 0.4 - 0.2 * math.exp(-(t - 100) / 5)
    # else:
    #     return 0.2 + 0.2 * math.exp(-(t - 200) / 2)

    #return 0.1 * math.sin(0.2*t) + 0.2
    return 0.2

def k2_function(t):
    # if t < 100:
    #     return 0.05
    # elif t < 200:
    #     return 0.5 - 0.45 * math.exp(-(t - 100) / 10)
    # else:
    #     return 0.05 + 0.45 * math.exp(-(t - 200) / 2)

    #return 0.2 * math.cos(0.5*t) + 0.3
    return 0.05

def zd_function(t):
    #return 0.2 * math.sin(0.1 * t)
    return 0

class SimConfig:
    # dt in ms
    def __init__(self):
        self.dt = 10
        self.k1 = lambda t: k1_function(t)
        self.k2 = lambda t: k2_function(t)
        self.zd = lambda t: zd_function(t)
        self.ks = 0.15
```

```
self.min_value = 0
self.max_value = 14
self.stable_value = 7
self.start_value = 7
self.noise = 0.5

class Simulation:
    def __init__(self, simulation_config: SimConfig):
        self.dt = simulation_config.dt
        self.start_val = simulation_config.start_value
        self.current_val = simulation_config.start_value
        self.stable_val = simulation_config.stable_value
        self.k1 = simulation_config.k1
        self.k2 = simulation_config.k2
        self.ks = simulation_config.ks
        self.zd = simulation_config.zd
        self.min_val = simulation_config.min_value
        self.max_val = simulation_config.max_value
        self.noise = simulation_config.noise
        self.df = []
        self.current_t = 0
        self.current_u = 0

    def step(self, u, zi, w):
        if u is not None:
            self.current_u = u

        # Calculate disturbance
        noise = np.random.normal(0, self.noise, 1)
        z = noise[0] + zi + self.zd(self.current_t / 1000)
        change = self._calculate_change(self.current_u, z)

        # Old system state (before step)
        temp = {
            SimKeys.TIME: self.current_t,
            SimKeys.VALUE: self.current_val,
            SimKeys.DELTA: self.current_val - w,
            SimKeys.CHANGE: change,
            SimKeys.ACTION: self.current_u,
            SimKeys.REFERENCE: w,
            SimKeys.K1: self.k1(self.current_t / 1000),
            SimKeys.K2: self.k2(self.current_t / 1000),
            SimKeys.Z: z
        }
        self.df.append(temp)

        self.current_val += change

        # Cap value between min and max
        if self.current_val < self.min_val:
            self.current_val = self.min_val
        elif self.current_val > self.max_val:
            self.current_val = self.max_val
```

```
    self.current_t += self.dt

    return self.current_val, self.current_t, temp

def reset(self):
    self.current_t = 0
    self.current_u = 0
    self.df.clear()
    self.current_val = self.start_val

    return self.current_val, self.current_t

def _calculate_change(self, u, z):
    # Divide by 1000 to convert from ms -> s
    return (-self.k1(self.current_t / 1000) * abs(self.stable_val -
self.current_val) - self.k2(
        self.current_t / 1000) + self.ks * u + z) * self.dt / 1000

def calculate_strategies(self, strategies: np.array, p_horizon,
act_every):
    # Calculate the actual influence of the different control
    # strategies on the system
    def model(y, t, u, dt, t0):
        return -(self.k1(t0 + t) * abs(y - self.stable_val) +
self.k2(t0 + t)) + self.ks * u[:, min(int(t / dt), u.shape[1] - 1)] +
self.zd(t0 + t)

    delta_t = act_every / 1000
    t_eval = np.arange(0, p_horizon + 1, 1) * delta_t
    y0 = np.ones(shape=strategies.shape[0]) * self.current_val

    values = odeint(model, y0, t_eval, args=(strategies, delta_t,
self.current_t / 1000))
    # Transpose matrix, because somehow it is in the wrong order
    values = np.array(values).T
    # Drop first column since it is the one for t=0 which we don't need
    values = values[:, 1:]

    return values
```

## 9.5 Python Skript für die Visualisierung der Ergebnisse aus „export.py“

```
import numpy as np
import pandas as pd
import os
import pathlib
import datetime
import os.path
import matplotlib.pyplot as plt
from simulation.simulation import SimKeys, Simulation
import matplotlib.cm as cm

class ExpKeys:
    TIME = "Time (ms)"
    MSE_TOTAL = "mse_total"
    MAPE_TOTAL = "mape_total"
    MAE_TOTAL = "mae_total"

class Data:
    def __init__(self, plots_every, agent_config, sim_config):
        self.plots_every = plots_every
        self.last_plotted = None
        self.last_individual_error = None
        self.last_individual_time = 0
        self.total_errors = []
        self.mse_tsteps = None
        self.mape_tsteps = None
        self.mae_tsteps = None
        self.agent_config = agent_config
        self.sim_config = sim_config

        self.path, self.image_path = self._create_folder()
        self._create_meta_data()

    def _create_folder(self):
        parent_path = os.path.join(pathlib.Path().resolve(), "data")
        if not os.path.exists(parent_path):
            os.makedirs(parent_path)

        folder_name = "Simulation_" + datetime.datetime.now().strftime("%Y-%m-%d %H-%M-%S")
        dir_name = os.path.join(parent_path, folder_name)
        os.makedirs(dir_name)

        im_path = os.path.join(dir_name, "Graphs")
        os.makedirs(im_path)

        return dir_name, im_path

    def _create_meta_data(self):
```

```
path1 = os.path.join(self.path, "sim_config.txt")
path2 = os.path.join(self.path, "agent_config.txt")

    with open(path1, "w") as f1, open(path2, "w") as f2:
        f1.write(''.join(["%s = %s\n" % (k, v) for k, v in
self.sim_config.__dict__.items()]))
        f2.write(''.join(["%s = %s\n" % (k, v) for k, v in
self.agent_config.__dict__.items()]))

def finish(self, sim: Simulation):
    dir_name = os.path.join(self.image_path, "summary")
    os.makedirs(dir_name)

    create_sim_linechart(sim, dir_name)

    if len(self.total_errors) == 0:
        return
    total_error = pd.DataFrame.from_records(self.total_errors)
    time = total_error[[ExpKeys.TIME]].div(1000)
    if self.mse_tsteps is not None and self.mape_tsteps is not None and
self.mae_tsteps is not None:
        create_error_linechart_steps(time.to_numpy(), self.mse_tsteps,
"MSE [-]", dir_name)
        create_error_linechart_steps(time.to_numpy(), self.mape_tsteps,
"MAPE [%]", dir_name)
        create_error_linechart_steps(time.to_numpy(), self.mae_tsteps,
"MAE [-]", dir_name)

        create_error_linechart_total(time.to_numpy(),
total_error[[ExpKeys.MSE_TOTAL]].to_numpy(), "MSE [-]", dir_name)
        create_error_linechart_total(time.to_numpy(),
total_error[[ExpKeys.MAPE_TOTAL]].to_numpy(), "MAPE [%]",
dir_name)
        create_error_linechart_total(time.to_numpy(),
total_error[[ExpKeys.MAE_TOTAL]].to_numpy(), "MAE [-]", dir_name)

    if self.last_individual_error is not None:
        f_name_heatmap = "error_heatmap.png"
        f_name_scatter = "3d_scatter_error.png"
        create_error_scatter_3d(self.last_individual_error,
self.last_individual_time,
                                         os.path.join(dir_name, f_name_scatter))
        create_error_heatmap(self.last_individual_error,
self.last_individual_time,
                                         os.path.join(dir_name, f_name_heatmap))

    def record_errors(self, predictions: np.array, actual_values: np.array,
time):
        mse_total = np.mean((predictions - actual_values) ** 2)
        mape_total = np.mean(np.abs((actual_values - predictions) /
actual_values)) * 100
        mae_total = np.mean(np.abs(actual_values - predictions))

        total = {
            ExpKeys.TIME: time,
```

```
        ExpKeys.MSE_TOTAL: mse_total,
        ExpKeys.MAPE_TOTAL: mape_total,
        ExpKeys.MAE_TOTAL: mae_total
    }
    self.total_errors.append(total)

    mse_tsteps = np.mean((predictions - actual_values) ** 2, axis=0)
    mape_tsteps = np.mean(np.abs(actual_values - predictions) /
actual_values), axis=0) * 100
    mae_tsteps = np.mean(np.abs(actual_values - predictions), axis=0)

    # Add different error measures for each time step to an array
    if self.mse_tsteps is None:
        self.mse_tsteps = np.copy(mse_tsteps)
        self.mse_tsteps = np.expand_dims(self.mse_tsteps, axis=0)
    else:
        self.mse_tsteps = np.vstack([self.mse_tsteps, mse_tsteps])

    if self.mape_tsteps is None:
        self.mape_tsteps = np.copy(mape_tsteps)
        self.mape_tsteps = np.expand_dims(self.mape_tsteps, axis=0)
    else:
        self.mape_tsteps = np.vstack([self.mape_tsteps, mape_tsteps])

    if self.mae_tsteps is None:
        self.mae_tsteps = np.copy(mae_tsteps)
        self.mae_tsteps = np.expand_dims(self.mae_tsteps, axis=0)
    else:
        self.mae_tsteps = np.vstack([self.mae_tsteps, mae_tsteps])

    quad_individual = (predictions - actual_values) ** 2
    self.last_individual_error = quad_individual
    self.last_individual_time = time

    # Graphs are only saved after a set intervall
    if self.last_plotted is None or time >= self.last_plotted +
self.plots_every:
        self.last_plotted = 0 if self.last_plotted is None else time

        mse_strategies = np.mean((predictions - actual_values) ** 2,
axis=1)
        mape_strategies = np.mean(np.abs(actual_values - predictions) /
actual_values), axis=1) * 100
        mae_strategies = np.mean(np.abs(actual_values - predictions),
axis=1)

        f_name = f"Time_{str(time / 1000)}s"
        dir_name = os.path.join(self.image_path, f_name)
        os.makedirs(dir_name)
        f_name_heatmap = "error_heatmap.png"
        f_name_scatter = "3d_scatter_error.png"

        create_error_heatmap(quad_individual, time,
os.path.join(dir_name, f_name_heatmap))
        create_error_scatter_3d(quad_individual, time,
```

```
os.path.join(dir_name, f_name_scatter))

        create_error_bar_chart_strategies(mse_strategies, "MSE [-]",
time, dir_name)
        create_error_bar_chart_strategies(mape_strategies, "MAPE [%]",
time, dir_name)
        create_error_bar_chart_strategies(mae_strategies, "MAE [-]",
time, dir_name)

        create_error_bar_chart_steps(mse_tsteps, "MSE [-]", time,
dir_name)
        create_error_bar_chart_steps(mape_tsteps, "MAPE [%]", time,
dir_name)
        create_error_bar_chart_steps(mae_tsteps, "MAE [-]", time,
dir_name)

def create_error_heatmap(errors: np.array, time, path):
    # First Axis are the different strategies, second axis are the future
time step
    plt.ioff()
    x_labels = [f"t+{str(i + 1)}" for i in range(errors.shape[1])]
    fig, ax = plt.subplots(figsize=(7, 6))
    im = ax.imshow(errors, interpolation='nearest', cmap='viridis',
aspect='auto')
    ax.set_xticks(np.arange(len(x_labels)), labels=x_labels)
    ax.set_xlabel("Zeitschritt [-]")
    ax.set_ylabel("Strategie Index [-]")
    plt.colorbar(im)
    plt.title(f"Individueller Quadratischer Fehler für t={str(time / 1000)}s")
    plt.savefig(path)
    plt.clf()
    plt.close()

def create_error_scatter_3d(errors: np.array, time, path):
    plt.ioff()
    fig = plt.figure()
    ax = plt.axes(projection="3d")
    x_data = np.arange(0, errors.shape[0], 1)
    y_data = np.arange(0, errors.shape[1], 1)
    X, Y = np.meshgrid(x_data, y_data)
    y_labels = [f"t+{str(i + 1)}" for i in range(errors.shape[1])]
    ax.scatter(X, Y, errors, s=2, c=Y, vmin=0, vmax=errors.shape[1],
cmap=cm.get_cmap("plasma"))
    ax.set_yticks(np.arange(len(y_labels)), labels=y_labels)
    ax.set_xlabel("Strategie Index [-]")
    ax.set_ylabel("Zeitschritt [-]")
    ax.set_zlabel("Quadratischer Fehler [-"])

    plt.title(f"Individueller Quadratischer Fehler für t={str(time / 1000)}s")

    plt.savefig(path)
```

```
plt.clf()
plt.close()

def create_error_bar_chart_strategies(errors: np.array, error_type, time,
path):
    plt.ioff()
    plt.figure()
    plt.bar(np.arange(0, errors.shape[0], 1), errors)
    plt.ylabel(error_type)
    plt.xlabel("Strategie Index")
    plt.title(error_type + " für die Vorhersage aller Strategien für t=" +
str(time / 1000) + "s")
    f_name = error_type + "_strategies.png"
    plt.savefig(os.path.join(path, f_name))
    plt.clf()
    plt.close()

def create_error_bar_chart_steps(errors: np.array, error_type, time, path):
    x_labels = [f"t+{str(i + 1)}" for i in range(errors.shape[0])]

    plt.ioff()
    plt.figure()
    plt.bar(x_labels, errors)
    plt.ylabel(error_type)
    plt.xlabel("Zeitschritt")
    plt.title(error_type + " für die Vorhersage aller Zeitschritte für t=" +
str(time / 1000) + "s")
    f_name = error_type + "_steps.png"
    plt.savefig(os.path.join(path, f_name))
    plt.clf()
    plt.close()

def create_error_linechart_steps(time: np.array, errors: np.array,
error_type, path, ylim=None):
    # Here errors is a 2d array that includes the error of every time step
    # over all predictions
    plt.ioff()
    plt.figure()
    legend = [f"t+{str(i + 1)}" for i in range(errors.shape[1])]

    for timestep in errors.T:
        plt.plot(time, timestep)

    if ylim is not None:
        plt.ylim(ylim)
    plt.grid(visible=True)
    plt.ylabel(error_type)
    plt.xlabel("Zeits [s]")
    plt.title(error_type + " aller Vorhersagen für die einzelnen
Zeitpunkte")
    plt.legend(legend)
    f_name = error_type + "_line_steps.png"
```

---

```
plt.savefig(os.path.join(path, f_name))
plt.clf()
plt.close()

def create_error_linechart_total(time: np.array, errors: np.array,
error_type, path, ylim=None):
    # Here errors is a 2d array that includes the error of every time step
over all predictions
    plt.ioff()
    plt.figure()
    plt.plot(time, errors)
    plt.grid(visible=True)
    plt.ylabel(error_type)
    plt.xlabel("Zeit [s]")
    if ylim is not None:
        plt.ylim(ylim)
    plt.title(error_type + " Gesamtfehler über die gesamte Laufzeit")
    f_name = error_type + "_line_total.png"
    plt.savefig(os.path.join(path, f_name))
    plt.clf()
    plt.close()

def create_sim_linechart(sim: Simulation, path):
    plt.ioff()
    df = pd.DataFrame.from_records(sim.df)
    time = df[SimKeys.TIME].div(1000)

    fig, axs = plt.subplots(4, 1, sharex='row', figsize=(10, 8),
gridspec_kw={'height_ratios': [4, 1, 1, 1]})

    axs[0].plot(time, df[SimKeys.VALUE])
    axs[0].plot(time, df[SimKeys.REFERENCE], c='k')
    axs[0].set_ylim(bottom=sim.min_val, top=sim.max_val)
    axs[0].set_xlim(left=0)
    axs[0].axhline(sim.stable_val, color='black', linestyle='dotted')
    axs[0].legend(["Regelgröße y", "Führungsgröße w"])
    axs[0].set_xlabel("Zeit [s]")
    axs[0].set_ylabel("y(t) [-]")
    axs[0].grid(visible=True)
    axs[0].set_title("Verlauf der Regelgröße über die Zeit")

    axs[1].plot(time, df[SimKeys.ACTION])
    axs[1].set_xlabel("Zeit [s]")
    axs[1].set_ylabel("u(t) [-]")
    axs[1].set_ylim(bottom=-0.1)
    axs[1].set_xlim(left=0)
    axs[1].grid(visible=True)
    axs[1].set_title("Verlauf der Stellgröße über die Zeit")

    axs[2].plot(time, df[SimKeys.Z])
    axs[2].set_xlabel("Zeit [s]")
    axs[2].set_ylabel("z(t) [-]")
    axs[2].set_xlim(left=0)
```

```
axs[2].grid(visible=True)
axs[2].set_title("Verlauf der Störungen über die Zeit")

axs[3].plot(time, df[SimKeys.K1])
axs[3].plot(time, df[SimKeys.K2])
axs[3].set_ylim(bottom=0)
axs[3].set_xlabel("Zeit [s]")
axs[3].set_ylabel("k(t) [-]")
axs[3].set_xlim(left=0)
axs[3].legend(["k1", "k2"])
axs[3].grid(visible=True)
axs[3].set_title("Verlauf der Zustandsgrößen k1 und k2 über die Zeit")

plt.tight_layout()

fname = "sim_linechart.png"
plt.savefig(os.path.join(path, fname))
plt.clf()
plt.close()
```