Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Optimierte dynamische Updatestrategien für verteilte Simulationen unter der Verwendung von Vorhersagemodellen

Suat Aydin

Studiengang: Informatik

Prüfer/in: Prof. Dr. Kurt Rothermel

Betreuer/in: Johannes Kässinger, M.Sc.

Beginn am: 2022-05-03

Beendet am: 2022-11-02

Kurzfassung

Die Verwendung von großen Neuronalen Netzen (NN) ermöglichen die Berechnung von sehr genauen Ergebnissen für Simulationen. Allerdings eignen sich diese nur begrenzt für den Einsatz auf mobilen Endgeräten, aufgrund möglicherweise unzureichender Rechenleistung der Endgeräte und einem damit einhergehenden hohen Stromverbrauch, der eine verkürzte Akkulaufzeit zur Folge hat.

Im Rahmen dieser Bachelorarbeit wurde ein verteiltes System zur Simulation einer Armbewegung, bestehend aus einem Client und einem Server entwickelt. Auf dem Server wird ein großes NN betrieben, welches auf Eingabe der Inputwerte der Armbewegung als Ergebnis die Punkte auf dem Oberarmmuskel in einer hohen Qualität berechnet und als Outputwerte ausgibt. Sowohl auf dem Client als auch dem Server werden gespiegelt Vorhersagemethoden betrieben, um auf beiden die zukünftigen Inputwerte der Armbewegung, wie auch auf beiden die zukünftigen Oberarmpunkte als Outputwerte vorauszusagen. Bei der verwendeten Vorhersagemethode handelt es sich um die *erweiterte gedämpfte Holt's Methode*, die eine Weiterentwicklung der *gedämpften Holt's Methode* durch [BM22] darstellt. Bei diesem handelt es sich um eine Vorhersagemethode, die aus einer Reihe von vergangenen Historienwerten, die zukünftigen Werte voraussagt. Auf dem Client steht zudem ein kleineres NN als Backup-Lösung zur Verfügung, welches etwas weniger genaue Outputwerte produzieren kann, falls ein vom Server berechnetes Ergebnis des großen NN nicht vorhanden ist.

Für den Einsatz dieser Vorhersagemethoden wurden verschiedene Updatestrategien realisiert, wann und wie sich Client und Server gegenseitig updaten. Im Rahmen der Updates werden nicht die Historienwerte der Input- und Outputwerte selbst übermittelt, sondern die Modelldaten aus denen die Vorhersagemethoden wieder rekonstruiert werden können. Überdies wurde ein Mechanismus entwickelt, der Bewegungsstillstände feststellen kann und in der Folge die Kommunikation reduziert.

Die Multizielsetzung des verteilten Systems besteht darin, soweit wie möglich die lokale Berechnung auf dem Client zu vermeiden und mit möglichst wenig Kommunikationsvorgängen und der Verwendung von Bandbreite qualitativ bestmögliche Outputergebnisse zu erreichen.

Das verteilte System ist in der Lage, in Abhängigkeit von der gewählten Updatestrategie, den Fehler im Vergleich zu einer rein lokalen Berechnung durch die Backup-Lösung um ca. 90% zu reduzieren. Die lokale Berechnung wird dabei weitestgehend vermieden. Die dazu erfolgten Kommunikationsvorgänge, durchschnittlichen Vorhersagedauern der Vorhersagemethoden und benötigten Bandbreiten des verteilten Systems variieren teilweise stark zwischen den verwendeten Updatestrategien und werden in der Evaluation ausführlich betrachtet und miteinander verglichen.

Abstract

The usage of large NN allow the calculation of very accurate results for simulations. However, their usage suit only limited for mobile devices, due to possibly insufficient computation power of the mobile devices and consequently higher electricity demand, which results in less battery life.

In this thesis a distributed system was developed for the simulation of an arm movement, consisting of a client and a server. A high quality NN is used on the server, which gets the input values of an arm movement and calculates the corresponding points of the upper arm‘s muscle surface in high quality and outputs them as result. The forecasting methods are operated mirrored on the client as well as the server to forecast on both the future input values of the arm movement and the future upper arm points as output values. The *extended damped Holt‘s method* is used as forecasting method, which is a further development of the *damped Holt‘s method*, developed by [BM22]. This is a time-series forecasting method which uses the past history values to forecast the future values. A low quality NN is available on the client as backup-solution, which produces less accurate output values, if a solution computed by the high quality NN of the server is not present.

Several update strategies were realised for using the forecasting methods, when and how the client and the server update each other. As part of the updates, the model data of the forecasting methods are sent, which allow the exact reconstruction of the forecasting methods, instead of the history values. Furthermore, a mechanism was developed, which is able to detect if the arm stands still and as a result lowers the communication.

The distributed system‘s multi-objective consists of avoiding the local execution on the client as far as possible and reaching the highest possible quality of output values by using as less as possible communication processes and bandwidth.

The distributed system is able to reduce the error by approx. 90% compared to the only execution of client‘s backup solution, depending on the chosen update strategy. The local execution is avoided there largely. The communication processes, average forecasting durations of the forecasting methods and required bandwidths of the distributed system which have taken place vary partially great between the different used update strategies and will be looked closely and compared with each other in the evaluation.

Inhaltsverzeichnis

1. Einleitung **17**

2. Verwandte Arbeiten **19**
2.1. Bidirektionale Kommunikation 19
2.2. Unidirektionale Kommunikation 23

3. Problembeschreibung **25**

4. Entwurf **27**
4.1. Systemmodell 27
4.2. Vorhersagemethode 31
4.3. Parameterbestimmungen 33

5. Implementierung **35**
5.1. Vorhersagemethode 35
5.2. Parameterbestimmungen 38
5.3. Verteiltes System 48

6. Evaluation **59**
6.1. Evaluationssetup 59
6.2. Testdatensätze und Fehlermaße 59
6.3. Empirische Ergebnisse 61
6.4. Empirische Ergebnisse mit Bandbreitenoptimierung der Outputparameter 76
6.5. Diskussion und Interpretation 78

7. Fazit **81**

Literaturverzeichnis **83**

A. Anhang **85**
A.1. Parameterbestimmungen für alternative Vorhersagehorizonte 85
A.2. Implementierungseinstellung *mit Löschen* 89
A.3. Implementierungseinstellung *ohne Löschen* 100
A.4. 30 Zeitschritte bandbreitenoptimierter Input und Output *ohne Löschen* 105
A.5. 30 Zeitschritte bandbreitenoptimierter Output *ohne Löschen* 116

Abbildungsverzeichnis

4.1. Systemarchitektur 28

6.1. Kommunikation (ohne Löschen) 63
6.2. Vorhersagedauern (ohne Löschen) 65
6.3. Bandbreite (ohne Löschen) 67
6.4. Gesamtqualität (ohne Löschen) 71
6.5. Qualität aller Updatestrategien in jedem Zeitschritt (ohne Löschen) 74
6.6. Qualität Updatestrategie (2, 1) (ohne Löschen) 75

A.1. Kommunikation (mit Löschen) 89
A.2. Vorhersagedauern (mit Löschen) 90
A.3. Bandbreite (mit Löschen) 91
A.4. Gesamtqualität (mit Löschen) 92
A.5. Qualität aller Updatestrategien in jedem Zeitschritt (mit Löschen) 93
A.6. Qualität Updatestrategie (0, 0) (mit Löschen) 94
A.7. Qualität Updatestrategie (0, 1) (mit Löschen) 95
A.8. Qualität Updatestrategie (1, 0) (mit Löschen) 96
A.9. Qualität Updatestrategie (1, 1) (mit Löschen) 97
A.10. Qualität Updatestrategie (2, 0) (mit Löschen) 98
A.11. Qualität Updatestrategie (2, 1) (mit Löschen) 99
A.12. Qualität Updatestrategie (0, 0) (ohne Löschen) 100
A.13. Qualität Updatestrategie (0, 1) (ohne Löschen) 101
A.14. Qualität Updatestrategie (1, 0) (ohne Löschen) 102
A.15. Qualität Updatestrategie (1, 1) (ohne Löschen) 103
A.16. Qualität Updatestrategie (2, 0) (ohne Löschen) 104
A.17. Kommunikation mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 105
A.18. Vorhersagedauern mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 106
A.19. Bandbreite mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 107
A.20. Gesamtqualität mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 108
A.21. Qualität aller Updatestrategien in jedem Zeitschritt mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 109
A.22. Qualität Updatestrategie (0, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 110
A.23. Qualität Updatestrategie (0, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 111

A.24. Qualität Updatestrategie (1, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 112
A.25. Qualität Updatestrategie (1, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 113
A.26. Qualität Updatestrategie (2, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 114
A.27. Qualität Updatestrategie (2, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen) 115
A.28. Kommunikation mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 116
A.29. Vorhersagedauern mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 117
A.30. Bandbreite mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) . 118
A.31. Gesamtqualität mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 119
A.32. Qualität aller Updatestrategien in jedem Zeitschritt mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 120
A.33. Qualität Updatestrategie (0, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 121
A.34. Qualität Updatestrategie (0, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 122
A.35. Qualität Updatestrategie (1, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 123
A.36. Qualität Updatestrategie (1, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 124
A.37. Qualität Updatestrategie (2, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 125
A.38. Qualität Updatestrategie (2, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen) 126

Tabellenverzeichnis

6.1. Parameterwerte der Inputvorhersage (4 Zeitschritte) 61
6.2. Spannweite der Parameterwerte der Outputvorhersage (4 Zeitschritte) 62
6.3. Bandbreitenbedarf der Updatestrategien im Verhältnis zum 100% offloading . . . 69
6.4. Benötigte Bandbreite für die Synchronisationsvariablen (Absolut) 70
6.5. Benötigte Bandbreite für die Synchronisationsvariablen (Verhältnis) 70
6.6. Fehlerreduzierung der Updatestrategien im Verhältnis zur Backup-Lösung 72

A.1. Parameterwerte der Inputvorhersage (5 Zeitschritte) 85
A.2. Spannweite der Parameterwerte der Outputvorhersage (5 Zeitschritte) 85
A.3. Parameterwerte der Inputvorhersage (10 Zeitschritte) 86
A.4. Spannweite der Parameterwerte der Outputvorhersage (10 Zeitschritte) 86
A.5. Parameterwerte der Inputvorhersage (20 Zeitschritte) 87
A.6. Spannweite der Parameterwerte der Outputvorhersage (20 Zeitschritte) 87
A.7. Parameterwerte der Inputvorhersage (30 Zeitschritte) 88
A.8. Spannweite der Parameterwerte der Outputvorhersage (30 Zeitschritte) 88

Verzeichnis der Listings

5.1. Aufbau des Vorhersagemodells 35
5.2. Rekonstruktion des Vorhersagemodells 37
5.3. Berechnung von γ 38
5.4. Berechnung von m 40
5.5. Iterative Berechnung von γ und m 41
5.6. Bestimmung des Startschwellenwertes 43
5.7. Bestimmung der Schwellenwerte 44
5.8. Abbruchkriterium der Schwellenwerte 46
5.9. Absenkung der Schwellenwerte 47
5.10. Überprüfung Bewegungsrichtungsänderung 49
5.11. Ankunft eines Serverupdates 52
5.12. Löschen von Servermodellen 54
5.13. Verwalten von Servermodellen 55

Abkürzungsverzeichnis

AR Augmentierten Realität. 25

FPS Frames Pro Sekunde. 49

mae mean absolute error. 60

mmse mean modified squared error. 39

ms Millisekunden. 49

mse mean squared error. 42

NN Neuronalen Netzen. 3

TCP Transmission Control Protocol. 19

UDP User Datagram Protocol. 19

Symbolverzeichnis

α - Glättungsparameter der Levelgleichung
β - Glättungsparameter der Trendgleichung
γ - Glättungsparameter der Beschleunigungsgleichung
ϕ - Dämpfungsparameter
b - Trendwert
b_t - Trendwert zum Zeitpunkt t
c - Beschleunigungswert
c_t - Beschleunigungswert zum Zeitpunkt t
dt - Trendkorrektur für verspätete Updates
l - Levelwert
l_t - Levelwert zum Zeitpunkt t
m - Länge/Größe der Historie
m_0 - Länge/Größe der Inputhistorien
m_1 - Länge/Größe der Outputhistorien
n - Vorhersagehorizont (Wieviele zukünftige Werte vorhergesagt werden soll)
t - Aktueller Zeitschritt
x - Koordinatenanzahl der (indizierten) Outputergebnisse
y - Inputwert
y_t - Inputwert zum Zeitpunkt t
$\hat{y}$ - Vorausgesagter Wert
$\hat{y}_{t+n|t}$ - Vorausgesagter Wert zum Zeitpunkt t für den Zeitpunkt t+n

1. Einleitung

Neuronale Netze (NN) sind vielfältig einsetzbar und finden in immer mehr Bereichen Anwendung. Problematisch wird jedoch der Einsatz großer NN auf mobilen Endgeräten, die nur über beschränkte Ressourcen, wie Rechenkapazität oder Strom verfügen. Dies kann dazu führen, dass zeitkritische Anwendungen nicht rechtzeitig ausgeführt werden können oder der Stromverbrauch die Anwendung auf dem mobilen Endgerät unpraktisch macht. Hierzu könnte die Berechnung entweder auf einen Server mit mehr Rechenleistung ausgelagert werden oder die Häufigkeit der Berechnungen auf dem mobilen Endgerät reduziert werden.

Grundlage dieser Bachelorarbeit ist eine Vorhersagemethode zur Vorhersage zukünftiger Werte einer Muskelbewegung. Mit Verwendung einer solchen Vorhersagemethode ließe sich die Anzahl der Ausführungen eines NN reduzieren. In [BM22] wird hierzu die gedämpfte Holt‘s Methode um einen weiteren Parameter erweitert. Diese resultierende, sogenannte *erweiterte gedämpfte Holt‘s Methode* wird im Rahmen dieser Arbeit verwendet, um zukünftige Inputwerte einer Bewegungsaufnahme vorherzusagen und die zukünftigen Outputwerte eines großen NN, der die Aktivierungswerte von Oberarmmuskeln während einer Armbewegung berechnen soll.

Das große NN wird auf einem Server zur Berechnung akkurater Ergebnisse ausgeführt. Um die Kommunikation zwischen dem Endgerät und dem Server zu reduzieren, werden die Vorhersagemethoden auf beiden verwendet. Die Inputvorhersage soll hierbei zukünftige Inputwerte vorhersagen und die Outputvorhersage die zukünftigen Ergebnisse des großen NN. Sowohl die Inputvorhersage als auch die Outputvorhersage werden gespiegelt auf dem Endgerät und dem Server angewendet. Durch die gespiegelte Ausführung der Inputvorhersage kann das Endgerät die vorhergesagten Inputwerte mit den tatsächlichen Inputwerten abgleichen und dadurch bestimmen, wann ein neues Modell an den Server geschickt werden soll. Die Spiegelung der Outputvorhersage erfolgt analog dazu. Das Endgerät schickt die Modelldaten seiner Inputvorhersage an den Server, sodass dessen Inputvorhersage gespiegelt dasselbe Vorhersagemodell bilden kann und dieselben zukünftigen Inputwerte vorhersagt wie das Endgerät. Umgekehrt schickt der Server die Modelldaten seiner Outputvorhersage an das Endgerät, sodass dessen Outputvorhersage gespiegelt dasselbe Vorhersagemodell bilden und dieselben zukünftigen Outputwerte vorhersagt wie der Server. Dieser gespiegelte Einsatz der Vorhersagemethoden auf dem Endgerät und dem Server soll für eine Reduzierung der Kommunikation zwischen beiden sorgen. In diesem hier erläuterten verteilten System handelt es sich nicht um ein verteiltes System, welches aus mehreren physischen Geräten besteht, sondern einem Server auf dem das Endgerät simuliert mitbetrieben wird.

Es wurden verschiedene Updatestrategien realisiert, wann ein neues Vorhersagemodell an die gespiegelte Vorhersagemethode des Clients bzw. Servers gesendet werden soll. Die Auswertung richtet sich mit Blick auf die resultierende Anzahl der Kommunikationsvorgänge, der damit einhergehenden benötigten Bandbreite, der durchschnittlichen Vorhersagedauer der Vorhersagemodelle, sowie der Qualität der Ergebnisse.

Im zweiten Kapitel erfolgt zunächst ein Literaturüberblick. In diesem wird auf verwandte Arbeiten eingegangen, die getrennt nach Art der Kommunikation behandelt werden. Die Trennung der verwandten Arbeiten in Abschnitte erfolgt danach, ob diese eine bidirektionale oder unidirektionale Kommunikation aufweisen. Nach jeder behandelten Arbeit wird eine Abgrenzung der vorliegenden Arbeit zu dieser vorgenommen.

Im anschließenden dritten Kapitel wird auf die Problembeschreibung eingegangen. Hier werden das zugrunde liegende Problem und die konzeptionellen Lösungsansätze aufgezeigt. Damit wird die Motivation hinter der Arbeit ersichtlich.

Dieses wird gefolgt vom vierten Kapitel, in dem der Entwurf behandelt wird. In diesem wird zunächst im ersten Abschnitt das konzeptionelle Systemmodell aufgezeigt, dass es später umzusetzen gilt. Im anschließenden Abschnitt wird die verwendete Vorhersagemethode im Detail betrachtet. Den Abschluss des Kapitels machen die Parameterbestimmungen aus, die für die Verwendung der Vorhersagemethode ermittelt werden müssen.

Aufbauend auf dem Entwurf erfolgt im Folgenden fünften Kapitel die Implementierung. Im Rahmen dessen wird im ersten Abschnitt die Umsetzung der Vorhersagemethode behandelt. Im Anschluss hieran wird auf die Implementierung der Parameterbestimmungen, die für die Vorhersagemethoden gebraucht werden, eingegangen. Den Abschluss des Kapitels macht als letzten Abschnitt die Betrachtung des verteilten Systems aus.

Die Evaluation der vorangegangenen Implementierung, erfolgt im sechsten Kapitel. In diesem wird zunächst kurz im ersten Abschnitt das Evaluationssetup genannt. Diesem folgt der Abschnitt, der die verwendeten Testdatensätze und Fehlermaße behandelt. Diesem schließt sich die eigentliche Auswertung der empirischen Ergebnisse als Hauptteil des Kapitels an. Dem folgt ein Abschnitt mit einer weiteren Auswertung mit Bandbreitenoptimierung der Outputparameter. Schließlich erfolgt im abschließenden Abschnitt eine Diskussion und Interpretation der Ergebnisse.

Im Fazit erfolgt eine Zusammenfassung der Arbeit. Hier werden auch die wichtigsten Ergebnisse und Erkenntnisse rekapituliert, die sich ergeben haben. Zum Schluss wird ein kurzer Ausblick auf zukünftige Arbeiten gegeben.

2. Verwandte Arbeiten

Nachfolgend wird ein Überblick über verschiedene verwandte Arbeiten gegeben und wie sich diese von dieser Arbeit abgrenzen. In *Abschnitt 2.1 Bidirektionale Kommunikation* werden Arbeiten betrachtet, die eine bidirektionale Kommunikation aufweisen, in der Client und Server miteinander kommunizieren. Im darauf folgenden *Abschnitt 2.2 Unidirektionale Kommunikation* werden Arbeiten betrachtet, deren verteiltes System eine unidirektionale Kommunikation aufweisen, wo eine Vielzahl von Sensoren, Messwerte an einen Server senden.

2.1. Bidirektionale Kommunikation

Dieser Abschnitt betrachtet Arbeiten, die eine Client-Server-Architektur enthalten, in der Client und Server miteinander kommunizieren.

Fiedler betrachtet das Problem der Synchronisation einer Physiksimulation, die auf zwei Rechnern exakt identisch ablaufen soll [Fie14b]. Der Anwender steuert mit seinen Eingaben einen Würfel, der dabei andere kleinere Würfel bei Kollision verschiebt, mit diesen verklebt oder herumschleudert, falls diese getroffen werden. Damit diese Simulation exakt gleich auf einem anderen Rechner ablaufen kann, werden die drei verschiedenen Möglichkeiten des *deterministic lockstep*, *snapshot interpolation* und *state synchronization* vorgestellt.

Beim *deterministic lockstep* werden die Eingaben für jeden Frame, sowie der Frame selbst (als Synchronisationsvariable) auf den sich die Eingabe bezieht an den anderen Rechner gesendet [Fie14a]. Auf diese Art kann sehr viel Bandbreite eingespart werden, da sich so die Bandbreite proportional zur Größe des Inputs verhält und nicht bspw. zur Anzahl der Objekte in der Simulation. Damit diese nun in der richtigen Reihenfolge und flüssig auf dem anderen Rechner ablaufen kann, wie auf dem Rechner, auf dem die tatsächliche Anwendereingabe stattfindet, wird eine Übertragung mittels User Datagram Protocol (UDP) und Redundanzübertragung und einem Puffer realisiert. Sobald der erste Input für den allerersten Frame eintrifft, wird die Ankunftszeit gespeichert und jedes weitere Frame vom Puffer so geliefert, als ob es zu dem Zeitpunkt verwendet werden soll plus 100ms. Obwohl Transmission Control Protocol (TCP) eine verlässliche Ankunft in richtiger Reihenfolge sicherstellt, wird die Übertragung mittels UDP realisiert. Das liegt daran, dass bei nicht angekommenen Paketen, TCP die zweifache round trip time warten muss, weshalb die Simulation pausiert werden müsste, da ohne Input n, Frame n nicht abgespielt werden kann und diese somit auf dem zweiten Rechner nicht flüssig ablaufen würde. (Eine Vergrößerung des Puffers wird aus Gründen der Benutzerfreundlichkeit nicht empfohlen.) Um nun im Falle von UDP sicherzustellen, dass der Input auch ankommt, wird allen UDP Paketen mit geringfügig mehr Bandbreite der Input der nächsten zwei Sekunden hinzugefügt, bevor ein Verbindungsabbruch festgestellt werden würde. Dafür schickt der Empfängerrechner jedes Frame eine Bestätigung des letzten erhaltenen Paketes bzw. Inputs an den Senderechner. Diese Bachelorarbeit unterscheidet sich zum einen darin,

dass neben den Synchronisationsvariablen (Zeitschritt und Nummer der Inputvorhersage) nur die Modelldaten der Vorhersagemodelle gesendet werden und nicht die Historienwerte, die dem Input entsprechen würden. Zum anderen treffen die Daten immer vollständig und rechtzeitig (also ohne Verzögerung der Übertragung) ein, da das verteilte System nur simuliert wird. Überdies werden keine exakt gleichen Input- bzw. Outputwerte verlangt mit denen die Input- bzw. Outputvorhersage beim Client und Server arbeiten sollen. Diese können auch bis zu einem Schwellenwert vom tatsächlichen Input bzw. Output abweichen.

Falls Determinismus der Simulation nicht sichergestellt ist oder die Anzahl der Anwender zu groß ist, wird die Möglichkeit der *snapshot interpolation* eingeführt [Fie14c]. Hierbei werden als Input alle nötigen Zustandsinformationen der Würfel, wie Position und Richtung geschickt. Auf dem Empfängerrechner wird jedoch keine Simulation ausgeführt, sondern dies als Schnappschuss gerendert. Um korrekte und fließende Übergänge zu gewährleisten werden die verschiedenen Schnappschüsse mit einer Hermite-Interpolation interpoliert. Um bei diesem Verfahren, jedoch der ausufernden Bandbreite Herr zu werden, werden verschiedene Möglichkeiten die Bandbreite zu reduzieren vorgestellt [Fie15a]. Der Hauptansatz von dem Gebrauch gemacht wird Bandbreite einzusparen, besteht darin, bei Schnappschüssen nicht alle Zustandsinformationen zu senden, sondern nur die relative Veränderung zu einem Referenzschnappschuss, da die meisten Würfel zwischen zwei Schnappschüssen nicht verändert werden. Kleinere Optimierungsmöglichkeiten bestehen darin, aus mehreren Werten, bei dem einer nicht mitgesendet wird, logisch einen fehlenden (der eingespart wurde) abzuleiten. Weitere Möglichkeiten bestehen in unterschiedlichen Codierungen oder dem versenden von Würfelindizes, falls es weniger Bandbreite kostet Indizes zu schicken statt nicht veränderte Würfel als nicht geändert zu codieren. Im Rahmen dieser Bachelorarbeit findet auf dem Client die tatsächliche Eingabe statt, die jedoch auch simuliert auf dem Server ausgeführt wird. Die Verarbeitung des echten wie auch simulierten Inputs erfolgt ebenfalls auf beiden. Genauso erfolgt auch die Input- als auch die Outputvorhersage gespiegelt sowohl auf dem Client und dem Server. Auch im Rahmen dieser Arbeit wird versucht im Falle einer individuellen Updatestrategie dahingehend Bandbreite einzusparen, dass überprüft wird, ob es günstiger ist, alle Vorhersagemodelle zu schicken oder nur jene zu schicken, die sich verändert haben und eine dazugehörige Indizesliste.

Als dritte Möglichkeit wird die *state synchronization* ausgeführt [Fie15b]. Genauso wie beim *deterministic lockstep* wird die Simulation auf beiden Rechnern ausgeführt. Jedoch wird im Gegensatz zum *deterministic lockstep* neben dem Input auch der Zustand mitgeschickt. Um Bandbreite einzusparen, erfolgt das Update der Würfel mittels eines absteigend geordneten Arrays, dass die Würfelprioritäten enthält. Diese Würfelprioritäten enthalten nicht nur die Prioritäten der Würfel gemäß des aktuellen Zeitschrittes, sondern werden alle über die Zeitschritte hinweg akkumuliert. Das Update in jedem Zeitschritt erfolgt für die ersten n Würfel, gemäß dieses Prioritätsarrays. Anschließend werden die Prioritäten der Würfel, die geupdated wurden, wieder auf Null gesetzt. Damit beim Empfängerrechner, die Simulation flüssig abläuft wird auch bei diesem Ansatz ein Puffer benötigt, der die Pakete gemäß ihrer Sequenznummer periodisch ausgibt. Ansonsten wird die Bandbreite mit Bandbreitenkompressionen wie in der zuvor ausgeführten *snapshot interpolation* durchgeführt. Der Unterschied dieser Arbeit besteht darin, dass nur die Modelldaten (die sich auf den Input oder Output beziehen,) zwischen Client und Server geschickt werden. Das Update erfolgt gemäß der jeweiligen Updatestrategie für den Input bzw. den Output. Abschließend muss konstatiert werden, dass diese Arbeit sich von allen drei Ansätzen dahingehend

unterscheidet, dass das hiesige verteilte System nur simuliert wird, was zur Folge hat, dass Updates immer in richtiger Reihenfolge und ohne Verzögerung der Übertragung ankommen, weshalb sich Fragen nach bzw. Probleme bei der Umsetzung mit TCP/UDP/etc. nicht stellen.

Gambetta untersucht die Synchronisation von Online-Multiplayer Spielen, in der das Spiel sehr schnell, hochdynamisch und mit mehreren Spielern, worunter auch Betrüger sein können, abläuft [Gam22a]. Der Client schickt seine Eingaben an den Server. Noch bevor der Server diesen prozessiert, führt der Client diesen lokal bei sich aus und rendert diesen. Der Server schickt als Antwort seine berechneten Ergebnisse an den Client, die dieser lokal bei sich haben muss. Sollte das Client-Ergebnis von dem des Servers abweichen, muss der Client sein Ergebnis dahingehend korrigieren. Um lokal korrekte Ergebnisse, die jedoch noch vom Server prozessiert werden, nicht fälschlicherweise korrigieren zu müssen (da diese zu einem früheren Zeitschritt noch nicht richtig sind), werden den Benachrichtigungen des Clients Sequenznummern hinzugefügt und denen des Servers die letzte bearbeitete Sequenznummer. Ein Update der gesamten Spielwelt bei Ankunft einer jeden beliebigen Spielereingabe, würde zu viel Berechnungszeit und Bandbreite kosten [Gam22b]. Aus diesem Grund sammelt der Server die Inputs aller Spieler, welche dann periodisch, z.B. alle 100ms, verarbeitet und für alle Spielwelten aktualisiert werden. Um dadurch wiederum dem Problem ruckartiger, nicht fließender Bewegungen in der Darstellung von Spielern bei allen anderen Clients zu lösen, werden die beiden Lösungsmöglichkeiten der losen Kopplung und der Entitäteninterpolation vorgestellt. Bei der losen Kopplung wird ausgehend von der aktuellen Bewegung die Position bis zum Eintreffen des nächsten periodisch vom Server verschickten exakten Ergebnisses vorausgesagt. Lose Kopplung ist jedoch nicht praktikabel, bei plötzlicher starker oder gar gegensätzlicher Veränderung der Bewegungsrichtung. Diese würde periodisch zu teleportationsartigen Bewegungen in der Darstellung der anderen Spieler führen, was das Spiel unspielbar machen würde. Aus diesem Grund wird für solche hochdynamischen Spiele als Konzept die Entitäteninterpolation vorgeschlagen. Dabei werden die anderen Spieler mit vergangenen Positions- und Bewegungsdaten angezeigt. D.h. es wird die Position und Bewegung die zwischen dem vorletztem und letztem Serverergebnis erhalten wurde, lokal beim Client angezeigt. Damit läuft die Darstellung der anderen Spieler um einen Zeitschritt hinter der tatsächlichen Bewegung nach, also bspw. 100ms. Ansonsten wird jedoch eine ruckelfreie Bewegung angezeigt. Im Rahmen dieser Bachelorarbeit werden keine exakten Input- und Outputergebnisse verlangt. Die vorausgesagten Input- und Outputergebnisse dürfen bis zu einem gewissen Schwellenwert vom tatsächlichen Input bzw. Output abweichen. Überdies werden die Updates nicht periodisch, sondern nur bei Bedarf, gemäß der Updatestrategien für die Input- bzw. Outputvorhersage verschickt.

Chan et al. beschreiben ein verteiltes System in dem virtuelle Objekte von einem Server an den Client geschickt werden [CLN01]. Statt vor Beginn der Anwendung alle Objekte umfangreich herunterzuladen, wird mit Hilfe von Vorhersagemethoden zur Laufzeit ermittelt, welche Objekte vermutlich benötigt werden, die dann entsprechend vom Server angefordert und bis zu deren Visualisierung im Speicher vorgehalten werden. Hierbei werden den Objekten Scoring-Werte in Abhängigkeit von Entfernung und dem Winkel des Objektes zur Benutzerblickrichtung vergeben. Je höher die Entfernung und der Winkel sind, desto kleiner ist der Scoring-Wert des Objektes diesen vom Server abzurufen bzw. im begrenzten Speicher vorzuhalten. Welche Objekte in das Blickfeld des Benutzers zukünftig geraten werden, erfolgt über eine Bewegungsvorhersage der Mauszeigerposition, um die zukünftige Position des Benutzers zu ermitteln. Konkret werden zwei verschiedene Vorhersagemethoden verwendet. Bei einer hohen Bewegungsgeschwindigkeit wird ein elliptisches Vorhersagemodell verwendet. Die Mausbewegung wird dabei im Rahmen einer elliptischen Gleichung interpretiert. Bei einer niedrigen Bewegungsgeschwindigkeit wird

auf das lineare Vorhersagemodell gewechselt. Dieses obige verteilte System mit diesem hybriden Vorhersagemodell stellt eine Erweiterung vorangegangener Arbeiten von Chim et al. [CGL+98] und [CLS+98] dar. Diese verwenden in ihrem verteilten System verschiedene Vorhersagemethoden zur Vorhersage der zukünftigen location. Alle haben gemein, dass diese ein Zeitfolgenmodell (time series) verwenden in dem die zukünftigen Werte durch eine Reihe vorangegangener Werte vorhergesagt werden. Der Unterschied in dieser Bachelorarbeit besteht zum einen darin, dass immer dieselbe Vorhersagemethode verwendet wird, nämlich die *erweiterte gedämpfte Holt's Methode* und nicht in Abhängigkeit eines Parameters zwischen verschiedenen Vorhersagemethoden gewechselt wird. Zum Anderen sollen der Client und der Server Modelldaten zum Aufbau der Vorhersagemodelle austauschen und keine Objekte.

Faerman et al. stellen die Verwendung eines Vorhersagemodells zur Performanzvorhersage bei der Versendung von Dateien in netzgebundenen datenintensiven verteilten Systemen vor [FSWB99]. Dies ist deshalb wichtig, um im Falle von Rechenoperationen zu bestimmen, ob diese beim Client oder dem Server ausgeführt werden sollen, wie z.B. Filteroperationen. Falls die Anfrage zunächst beim Server gefiltert wird, müssen weniger Daten über das Netzwerk an den Client geschickt werden. Falls der Server jedoch stark ausgelastet ist, würde es sich bei guter Latenz und Bandbreite eignen, die Daten an den Client zu schicken und diesen lokal bei diesem zu filtern. Falls wiederum die Verbindung zwischen Client und Server schlecht ist, würde sich eine Kompression der Daten beim Server und eine Dekompression beim Client anbieten. Dies würde zwar sowohl den Client als auch den Server belasten, erzielt jedoch aufgrund der Vermeidung der Übertragung großer Datenmengen bei schlechter Verbindung ein besseres Gesamtergebnis. Nach der Dekompression auf Client-Seite könnte der Client die Filteroperation übernehmen. Außerdem ist es möglich, dass der Scheduler zwischen mehreren Servern den besten für die Bearbeitung der Anfrage auswählt. Die Vorhersage zielt darauf ab Performanz und Rechenauslastung zu ermitteln, um das Gesamtziel der Performanzmaximierung zu erreichen. Bei der Vorhersagemethode handelt es sich um ein angepasstes Regressionsmodell, dass mit Vergangenheitswerten die Vorhersage aufbaut. Die Regressionskoeffizienten werden dynamisch zur Laufzeit berechnet, falls eine Vorhersage gebraucht wird. Nach jeder Dateiübertragung wird die Regressionsfunktion neu berechnet und mit dem neueren Modell die nächste Dateiübertragung vorhergesagt. Die Parameter der vorangegangenen Regressionsmodelle werden gespeichert und für zukünftige Vorhersagen verwendet, wobei mit jedem neuen Regressionsmodell das älteste Datum gelöscht wird. Grundsätzlich sind zwar die Anzahl der vergangenen Parameter für den initialen Aufbau des Vorhersagemodells, sowie jedes weitere frei zu wählen, jedoch haben sich für den initialen Aufbau die vorangegangenen zwei Vergangenheitswerte herausgestellt und für den weiteren Verlauf die letzten 8 bis 20 Vergangenheitswerte. In dieser Bachelorarbeit finden die Vorhersagen gespiegelt, sowohl auf dem Client als auch auf dem Server statt. Das Ziel deren Einsatzes besteht nicht nur in der Performanzmaximierung, sondern auch der Kommunikationsreduktion und in deren Konsequenz einer damit angestrebten Bandbreitenreduktion. Die Parameter für die Verwendung der Vorhersagemodelle werden einmalig, offline vor der Simulation des verteilten Systems berechnet und ändern sich nicht zur Laufzeit. Das initiale Vorhersagemodell wird aufgebaut, sobald die Historien aller für die Vorhersage benötigten Parameter vollständig gefüllt sind.

Wolski et al. treffen Vorhersagen in verteilten Systemen zur Verfügbarkeit von dynamisch ändernden Leistungsmerkmalen, die verteilt in diesen vorliegen [WSH99]. Es werden kurzfristige Vorhersagen über Auslastung und Rechenressourcenverfügbarkeit auf Grundlage historischer Leistungsmessungen getroffen. Die Anfragen von Benutzern werden über einen Forecaster process, der als proxy fungiert weitergeleitet. Zur Leistungsmessung werden sowohl vorhandene Dienstprogramme als

auch die aktive Ressourcenbelegung verwendet. Es werden u.a. die TCP connection time, end-to-end TCP network latency und end-to-end TCP network bandwidth gemessen. Sensoren im verteilten System messen verschiedene Performanzkriterien und diese werden mit Hilfe von Persistent State Prozessen mit Zeitstempel versehen als Ring gespeichert, was bei einer kontinuierlichen Aktualisierung immer die jüngsten Einträge bei einer Anfrage liefert. Für die Vorhersage fragt der Forecaster process die Messhistorie vom Persistent State process an, welcher diese als Zeitreihe geordnet zuschickt. Diese enthält neben den time stamp-measurements pairs keine Modellierungsinformationen. Die Vorhersage wird alleine aufgrund der Messhistorie gebildet und ist damit nicht beschränkt auf bestimmte Zeitreihen. Es werden verschiedene Vorhersagemethoden auf die gesamte Reihe angewendet, unter denen jenes dynamisch zur Vorhersage ausgewählt wird, welches im Vergleich zu den jüngsten Messungen am genauesten war. Genauer liefert jede der Vorhersagemethoden eine Vorhersage für jede Messung, wobei der Forecaster process das kumulierte Fehlermaß für jedes festhält. Es wird jene Vorhersagemethode zur Vorhersage der zukünftigen Messwerte ausgewählt, dass für die bekannten Messungen den geringsten Vorhersagefehler erzeugt. Diese Bachelorarbeit unterscheidet sich bei der Vorhersagemethode dadurch, dass nur eine Vorhersagemethode verwendet wird. Es handelt sich bei dieser ausschließlich um die *erweiterte gedämpfte Holt's Methode* und steht bereits von Beginn an fest. Die Client-Server-Architektur hat eine 1 zu 1 Kardinalität, indem also genau ein Endgerät mit genau einem Server kommuniziert. Nachfolgend werden Architekturen betrachtet, in dem diese Kommunikation nicht mehr wechselseitig, sondern nur noch einseitig erfolgt.

2.2. Unidirektionale Kommunikation

In diesem Abschnitt werden Arbeiten betrachtet, die eine Sensor-Server-Architektur enthalten, bei dem eine Vielzahl von Sensoren, Messdaten an einen Server senden. Die folgenden Arbeiten unterscheiden sich alle von dieser Bachelorarbeit in der einseitigen Kommunikation hin zum Server, da es sich um Sensoren und Überwachungsgeräte handelt.

Barrios et al. beschreiben ein intervehicle collision avoidance system für Kraftfahrzeuge [BMH16]. Das System arbeitet mit einem dynamischen Vorhersagemodell, um den zukünftigen Fahrverlauf eines Fahrzeuges zu bestimmen. Drei Sensoren werden verwendet zur Messung von Position, Geschwindigkeit und Beschleunigung. Die ersten beiden senden im Sekundentakt die Position und Geschwindigkeit an das System. Die Beschleunigung wird im Zehntel Sekundentakt verschickt. Das System läuft in der Frequenz des schnellsten Sensors, d.h. im 0,1 Sekundentakt. Zur Vorhersage von Position, Geschwindigkeit und Beschleunigung werden drei dynamische lineare Modelle verwendet. Falls ein Sensor zeitweise ausfallen sollte, wird dessen Wert mit Hilfe der verbliebenen Sensoren ermittelt. Dies sorgt für genauere Ergebnisse als jene auf Basis des letzten bekannten gelesenen Wertes den aktuell gültigen zu bestimmen. Insgesamt sorgt der Ansatz das System in der Frequenz des schnellsten Sensors laufen zu lassen, Verbesserungen in der Vorhersage deutlich schneller zu treffen als wenn das System gezwungen wäre in der Frequenz des langsamsten Sensors laufen zu müssen. Im Gegensatz zu dieser unidirektionalen Kommunikation bzw. Versendung von Informationen wird im Rahmen dieser Bachelorarbeit eine Client-Server-Architektur mit bidirektionaler Kommunikation zwischen beiden realisiert. Überdies werden die Inputwerte der Muskelbewegung (Winkel, Geschwindigkeit, Beschleunigung und Gewicht) in derselben Frequenz

versendet, d.h. es stehen immer alle Inputwerte zum selben Messzeitpunkt zur Verfügung oder theoretisch keines, falls bspw. das Tracking der Bewegung oder die Verbindung zum Server abbrechen würde.

Liu et al. beschreiben die Vorhersage des Strombedarfs innerhalb des Stromversorgungsnetzes in einer südchinesischen Provinz [LZL+16]. Statt den Bedarf zentral zu bestimmen, wird das Stromnetz in Abhängigkeit von lokalen Wetterdaten (v.a. Temperatur und Luftfeuchtigkeit) in kleinere Subnetze unterteilt für die einzeln der Bedarf in Abhängigkeit der lokalen Wetterdaten vorhergesagt wird. (Die Unterteilung in Subnetze sorgt für ein genaueres Ergebnis und lässt Rückschlüsse über meteorologische Einflussfaktoren zu.) All diese lokalen Bedarfe werden anschließend zu einem aggregierten Ergebnis zusammengefasst, die den Gesamtbedarf darstellen. Konkret erfolgt die Aufteilung und Aggregation mit Hilfe von MapReduce. Die Map-Methode unterteilt das Netz in Subnetze aufgrund geografischer Verteilung und unterschiedlichen lokalen Wetterverhältnissen. Je nachdem wie ähnlich die Belastungskurve der verschiedenen Subnetze ist, führt die Reduce-Methode diese schließlich wieder zusammen. Entsprechend werden zwei verschiedene Vorhersagemethoden verwendet. Für die Vorhersage des lokalen Strombedarfes, also innerhalb eines Subnetzes, wird ein NN verwendet. Dazu werden die lokalen Wetterdaten wie Temperatur, Luftfeuchtigkeit, Preis, Tagestyp, etc. herangezogen. Für die Vorhersage des Gesamtbedarfs werden die kumulierten Subnetzbedarfe mit einem Gewicht von etwas über 1 gewichtet, da der tatsächliche Gesamtbedarf bspw. aufgrund von Übertragungsverlusten knapp über dem kumulierten Bedarf liegt. Der Koeffizient mit dem gewichtet wird, wird über die vergangene Durchschnittsdifferenz gewichtet aufsummiert und exponentiell abnehmend über die vergangenen Einträge der Zeitreihe geglättet.
Der Unterschied dieser Bachelorarbeit zu diesem Ansatz besteht darin, dass das System nur mit Input-Daten eines Endgerätes gespeist wird, wohingegen im Ansatz von Liu et al. die Inputdaten über eine Vielzahl von Messgeräten, die sich verteilt über einem großen räumlichen Gebiet befinden, gesammelt werden.

3. Problembeschreibung

In diesem Kapitel wird auf die Motivation dieser Bachelorarbeit eingegangen, wozu das zugrunde liegende Problem genau betrachtet, sowie der konzeptionelle Lösungsansatz aufgezeigt wird.

Mobile Endgeräte ermöglichen heutzutage die Verwendung einer Vielzahl von Anwendungen zur Augmentierten Realität (AR), die dem Benutzer das Ergebnis der hierfür betriebenen Simulation in Echtzeit anzeigen kann. Diese AR Anwendungen ermöglichen Simulationen mit sehr genauen Ergebnissen, benötigen hierfür jedoch zumeist sehr viel Rechenleistung, die auf einem mobilen Endgerät eventuell nicht zur Verfügung steht. In diesem Zusammenhang geht damit ein hoher Stromverbrauch einher, der auf einem mobilen Endgerät, aufgrund der in der Folge enstehenden kürzeren Akkulaufzeit nicht wünschenswert ist.

Diesem Umstand kann damit begegnet werden, dass bspw. die Simulationsberechnungen vollständig auf einen entfernten Server ausgelagert werden, der aufgrund seiner größeren Rechenleistung, sehr genaue Ergebnisse in Echtzeit berechnet.

Eine weitere Möglichkeit besteht darin, wie in [BM22] auf einem entfernten Server ein NN als Ersatzmodell zu betreiben, welcher die Simulationsergebnisse näherungsweise voraussagt. Um die Kommunikation und die Anzahl der Aufrufe für dieses auf dem entfernten Server betriebene Ersatzmodell zu reduzieren, werden Vorhersagemethoden verwendet, die den Output des Ersatzmodells für einen gewissen Zeitraum voraussagen können. Die dort verwendete Vorhersagemethode stellt eine Erweiterung der gedämpften Holt‘s Methode zur sogenannten *erweiterten gedämpften Holt‘s Methode* dar.

Das Ziel besteht zum einen darin, die Kommunikation zwischen mobilem Endgerät und Server möglichst gering zu halten. Zum anderen sollen die umfangreichen Berechnungen, die durch die Verwendung von NN entstehen, möglichst nicht lokal durchgeführt werden müssen und überdies in der Konsequenz Strom und Akkulaufzeit gespart werden, sodass das mobile Endgerät über einen längeren Zeitraum verwendet werden kann.

Konkret handelt es sich bei der dort betrachteten Simulation, um eine Projektion der Oberarmmuskeln mit entsprechender Färbung der Muskelaktivierungswerte, die aus einer Armbewegung heraus berechnet werden. Dies führt schlussendlich zu folgender Forschungsfrage:

"Lässt sich mithilfe der in [BM22] vorgestellten Vorhersagemethode die lokale Berechnung einer Armbewegung reduzieren und die anfallende Kommunikation innerhalb des hierzu implementierten verteilten Systems bei möglichst hoher Genauigkeit der Simulationsergebnisse einsparen."

4. Entwurf

In diesem Kapitel wird auf die konzeptionellen Grundlagen dieser Arbeit eingegangen, die es zu implementieren gilt. Hier liegt der Fokus ausschließlich auf der Konzeption, um damit ein Verständnis für deren Implementierung zu schaffen, deren Betrachtung gesondert im anschließenden *Kapitel 5 Implementierung* erfolgt. Zunächst wird das Systemmodell des zu implementierenden verteilten Systems aufgezeigt. Anschließend wird die verwendete Vorhersagemethode betrachtet. Den Abschluss machen die Parameterbestimmungen für die in der Vorhersagemethode verwendeten Parameter.

4.1. Systemmodell

In diesem Abschnitt wird auf die konzeptionelle Systemarchitektur eingegangen, deren Implementierungsdetails gesondert in *Abschnitt 5.3 Verteiltes System* betrachtet werden.

4.1.1. Systemarchitektur

In *Abbildung 4.1* wird der Aufbau des Gesamtsystems visualisiert. Unterhalb der horizontalen Trennlinie befindet sich der Anteil, der auf den Client entfällt und oberhalb der Trennlinie der Anteil des Servers. Wie bereits augenscheinlich am Schaubild erkannt werden kann, machen die Vorhersagen (Inputvorhersage und Outputvorhersage) das Herzstück dieser Arbeit aus, weshalb diese ausführlich im nachfolgenden *Abschnitt 4.2 Vorhersagemethode* untersucht werden, sodass in diesem Abschnitt nicht näher auf diese eingegangen wird.

Inputvorhersage des Clients

Der Client erhält periodisch die neuen Inputwerte der aktuellen Armbewegung. Bei den Inputparametern handelt es sich um den Winkel, den Ober- und Unterarm bilden, die Winkelgeschwindigkeit, die Winkelbeschleunigung und den Gewichtsparameter, wie stark der Arm belastet wird. Der Gewichtsparameter kann ignoriert werden, da für diesen keine detaillierten Simulationsdaten vorliegen. Diese Inputparameter gehen zum einen in die Inputvorhersage des Clients, die Schwellenwertüberprüfung der Inputvorhersage und die Outputausgabe.

Die Inputvorhersage des Clients besitzt eine Historie mit einer Größe m, d.h. diese enthält zeitlich geordnet die letzten m angekommenen Inputwerte. Auf Grundlage dieser Historie wird der erste bis n-te nächste Inputwert vorhergesagt. Die initialen Vorhersagemodelle zu allen Inputparametern werden gebildet, sobald die Historien von allen Inputparametern vollständig gefüllt sind, d.h. nach $\max(m_0)$ Zeitschritten wird auf einmal für jeden Inputparameter dessen Vorhersagemodell gebildet.

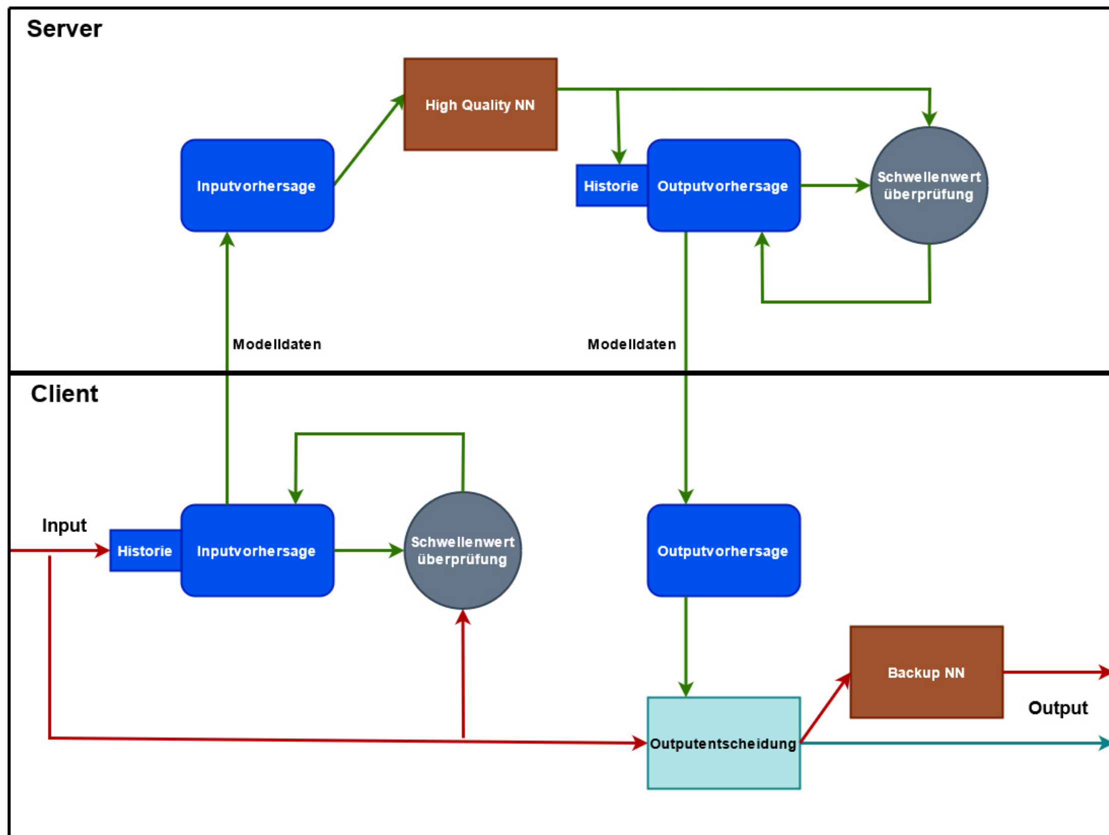

Abbildung 4.1.: Systemarchitektur

Die anschließende Vorhersage erfolgt periodisch mit der Ankunft der Inputwerte. Sobald ein neuer Inputwert ankommt wird überprüft, ob der vorhergesagte Wert innerhalb eines bestimmten Schwellenwertes liegt. Falls dies der Fall ist, wird das Vorhersagemodell nicht neu gebildet, sondern wird auch im nächsten Zeitschritt verwendet. Sollten jedoch der vorhergesagte Wert und der tatsächliche Inputwert über einen Schwellenwert hinaus voneinander abweichen, wird ein neues Vorhersagemodell auf Grundlage dieser veränderten Historie gebildet. Dies erfolgt unabhängig voneinander für jeden Inputparameter. Auf die Parameter der Vorhersagemodelle wird genauer in *Abschnitt 4.3 Parameterbestimmungen* eingegangen.

Auf die Frage wann nach der initialen Erstellung und Verschickung der Vorhersagemodelle an den Server, Folgemodelle an den Server geschickt werden, wird gesondert in *Unterabschnitt 4.1.2 Updatestrategien* eingegangen.

Um Bandbreite einzusparen werden jedoch nicht die Historienwerte verschickt, sondern die Modelldaten des Vorhersagemodells. Bei den Modelldaten handelt es sich um den *Stützwert*, den *Trend* und die *Beschleunigung* mit denen die gespiegelte Vorhersage exakt dasselbe Vorhersagemodell rekonstruieren kann. Statt also m viele Float-Zahlen als Historienwerte, werden genau drei Floatzahlen als Modelldaten verschickt. Damit sich dieser Ansatz lohnt, müssen die Vorhersagen durchschnittlich über drei Zeitschritte halten. Bei durchschnittlich genau drei Zeitschritten würde es keinen Unterschied machen, ob man die angesprochenen drei Modelldaten oder immer periodisch in

jedem Zeitschritt die aktuellen Inputwerte verschickt. Falls ein Vorhersagemodell durchschnittlich weniger als drei Zeitschritte hält, würde sich dieser Ansatz umgekehrt sogar nicht nur nicht lohnen, sondern würde mehr Kommunikation verursachen.

Server

Die Inputvorhersage des Servers stellt die simulierte Realität dar. Diese liefert die Inputwerte der Armbewegung, die bis zu einem Schwellenwert von den tatsächlichen Inputwerten abweichen dürfen. Dadurch ist es möglich, dass der Client nicht in jedem Zeitschritt die tatsächlichen Inputwerte an den Server schicken muss. Tatsächlich würde bei einem Inputmodell, dass theoretisch immer gültig ist und nie aufgrund einer zu großen Abweichung erneut ein Vorhersagemodell bilden müsste, das erstmalige verschicken der Modelldaten an den Server ausreichen, sodass danach nie wieder der Client etwas an den Server schicken müsste. Dies ist prinzipiell möglich, jedoch gar nicht erforderlich. Es reicht wie oben bereits erwähnt, wenn die Vorhersagen im Durchschnitt mindestens über drei Zeitschritte halten.

Die Inputvorhersage des Servers liefert die Eingaben an das High Quality NN, welches auf dem Server läuft. Dabei handelt es sich um zwei sequentiell geschaltete NN. Das Activation NN erhält die vier Inputparameter der Armbewegung (Winkel, Winkelgeschwindigkeit, Winkelbeschleunigung und Gewicht) als Input und gibt als Output die fünf Aktivierungswerte für die Muskeln Bizeps Brachii, Brachialis, Brachioradialis, Trizeps Brachii & Anconeus aus, d.h. wie stark diese fünf Muskeln belastet werden. Der Output dieses Activation NN stellt den Input des direkt dahinter geschalteten Deformation NN dar. Dieses berechnet aus diesen fünf Muskelaktivierungen, 2.809 Deformationspunkte (mit jeweils x-, y- und z-Koordinate) auf dem Oberarm, um diesen sehr genau visualisieren zu können. Von diesen 2.809 Deformationspunkten werden jedoch nur 30 indizierte Punkte (bzw. 90 dazugehörigen Koordinatenwerte) ausgewählt und als Outputparameter geführt, um die Bandbreite auf einem Minimum zu halten.

Outputvorhersage des Clients

Diese 90 Koordinatenwerte werden an die Outputvorhersage und die Schwellenwertüberprüfung der Outputvorhersage übergeben. Die Outputvorhersage auf dem Server arbeitet auf dieselbe Art wie die Inputvorhersage des Clients. Nachdem die Historien von allen Outputparametern vollständig gefüllt sind, d.h. nach $\max(m_1)$ Zeitschritten werden auf einmal für alle 90 Outputparameter (30 Punkte mit je 3 Koordinaten) die initialen Vorhersagemodelle gebildet und an die Outputvorhersage des Clients geschickt. Die Outputvorhersage des Clients produziert also periodisch die Outputergebnisse, die innerhalb eines Schwellenwertes von den tatsächlichen Outputergebnissen, die auf dem Server berechnet werden abweichen.

Die Outputentscheidung des Clients wird sowohl von der Outputvorhersage des Clients als auch durch die tatsächlichen Inputwerte gespeist. Falls kein Ergebnis durch die Outputvorhersage vorliegt, kann der Client bzw. genauer die Outputentscheidung immer auf das Backup NN zurückgreifen. Hierbei handelt es sich wieder um zwei sequentiell geschaltete NN. Beim ersten handelt es sich wie auch zuvor beim Server, um das Activation NN. Es bekommt also genauso wie jenes auf dem Server als Input, die Inputwerte der Armbewegung und gibt als Output die Muskelaktivierungswerte aus. Das dahinter geschaltete Backup NN bekommt zwar auch als Input die fünf Muskelaktivierungswerte,

gibt jedoch als Output nur 30 Punkte (mit jeweils x-, y- und z-Koordinate) aus. Diese 30 Punkte der Backup-Lösung sind jedoch nicht so genau, wie die auf Seiten des Servers berechneten. Sofern also ein vom Server berechnetes Ergebnis vorliegt, soll die Outputentscheidung immer dieses verwenden und nur bei dessen nicht vorliegen mit den tatsächlichen Inputwerten final die Backup-Lösung berechnen und ausgeben. Bis ein Ergebnis des Servers vorliegt, dauert es mehr als $\max(m_0) + \max(m_1)$ Zeitschritte. In dieser Zeit muss in jedem Fall das Backup NN verwendet werden.

4.1.2. Updatestrategien

Nach der initialen Erstellung und Versendung der Vorhersagemodelle für die Inputvorhersage an den Server bzw. umgekehrt der Vorhersagemodelle für die Outputvorhersage an den Client, können für den weiteren Verlauf verschiedene Updatestrategien realisiert werden, wann ein neues Vorhersagemodell an den Server bzw. Client geschickt werden soll. Hierzu wurden drei verschiedene Updatestrategien für die Inputvorhersage und zwei verschiedene Updatestrategien für die Outputvorhersage entwickelt. Bei den drei Updatestrategien, handelt es sich um die *Individual-*, *Master-* und *All-* Strategie.

Im Falle der *Individualstrategie* wird das verschicken der Parameter individuell behandelt. Falls ein neues Vorhersagemodell gebildet werden muss, wird dieses auch sofort an die Vorhersage des Clients bzw. Servers gesendet, sodass sowohl auf dem Client als auch auf dem Server gespiegelt immer dieselben Vorhersagemodelle laufen.

In der *Masterstrategie* dagegen, wird nur auf einen zuvor als Master bestimmten Parameter geachtet. Nur falls für diesen Masterparameter ein neues Vorhersagemodell erstellt werden muss, werden für diesen, wie auch für alle anderen, deren Vorhersagemodelle an den Client bzw. Server geschickt. Für Parameter, für die in demselben Zeitschritt keine neuen Vorhersagemodelle erstellt wurden, werden diese trotz dessen, dass die Vorhersagemodelle noch halten, auf Grundlage ihrer jetzigen neuen Historien erneut gebildet, um die bisherigen Vorhersagemodelle für diese, welche gespiegelt auf dem Client bzw. Server laufen zu ersetzen. Ansonsten würden die gespiegelten Vorhersagemodelle für solche Parameter u.U. sinnloserweise, dieselben bereits auf diesen laufenden Vorhersagemodelle erhalten oder in jedem Fall veraltete Vorhersagemodelle verwenden.

Schließlich löst bei der *Allstrategie* die Neuerstellung jedes beliebigen Parameters das verschicken von neuen Vorhersagemodellen für alle Parameter an den Client bzw. Server aus. Auch hier müssen die Vorhersagemodelle für Parameter, die in diesem Zeitschritt nicht bereits neu erstellt wurden, ein neues Vorhersagemodell erfahren.

Die *Masterstrategie* bietet sich nur im Falle der Inputvorhersage an. Für die Inputvorhersage kann der Winkelparameter von überragender Bedeutung identifiziert werden, da der Winkelparameter den mit Abstand größten Einfluss auf die NN hat, weshalb eine gesonderte Betrachtung dessen sich anbietet. Im Falle der Outputvorhersage fehlt ein solcher Parameter, der einen solch maßgeblichen Einfluss hat, weshalb die *Masterstrategie* sich nicht für diese anbietet.

Daraus ergeben sich kombiniert sechs verschiedene (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1)

4.1.3. Zusatzliche Anforderungen

Das in den vorangegangenen Unterabschnitten ausgeführte verteilte System stellt den Rumpf an Anforderungen dar. Es soll überdiese hinaus in der Lage sein, Bewegungsstillstände und in einem begrenzten Maße Verbindungsabbrüche zu identifizieren und diese auf sinnvolleweise in die Gesamtverarbeitung mit einzubeziehen.

Falls der Arm still steht, sollen keine neuen Inputvorhersagemodelle an den Server geschickt werden. Es würde eine Verschwendung von Rechenleistung und Bandbreite darstellen, da beim Stillstehen des Armes immer derselbe Output ausgegeben werden soll. Diese neuen Vorhersagemodelle können unbeabsichtigt durch einen zittrigen oder wackligen Arm verursacht werden. Mit der Ausgabe immer desselben Outputergebnisses würde in der Konsequenz das Wackeln bzw. Zittern nicht gezeigt werden.

Abschließend muss ein Mechanismus entwickelt werden, der in einem begrenzten Maße Verbindungsabbrüche behandelt. Da das verteilte System nur simuliert wird, können Verbindungsabbrüche zum Server nicht auftreten. Es muss jedoch eine Möglichkeit geschaffen werden, dass bei einem theoretischen Verbindungsabbruch weiterhin ein Outputergebnis geliefert wird. Falls die Verbindung zum Server abbricht, muss automatisch auf die Backup-Lösung zurückgegriffen werden, was es auch tut, wie es auch zuvor schon beschrieben wurde.

Überdies werden zwei verschiedene Implementierungseinstellungen vorgenommen, nämlich *mit Löschen* und *ohne Löschen*. Für die Einstellung *mit Löschen* müssen ankommende Outputmodelle des Servers, die veraltet sind oder sich auf eine zurückliegende und inzwischen veraltete Armbewegung beziehen vom Client erkannt und bei Bedarf verworfen werden. Sobald sich die Bewegungsrichtung des Armes verändert, d.h. der Geschwindigkeitsparameter, der neben der eigentlichen Geschwindigkeit auch die Richtung mit angibt, das Vorzeichen wechselt, sollen alle Server-Outputmodelle beim Client für ungültig erklärt werden. Das nächste valide Servermodell könnte dann frühestens zwei Zeitschritte später verwendet werden, sofern nach einer Bewegungsrichtungsänderung, sofort ein neues Inputvorhersagemodell an den Server geschickt wird, welches auf Grundlage der neuen gültigen Armbewegung ein gültiges Outputergebnis produziert und dieses sofort an den Client zurückschickt. Ansonsten müsste der Client jedes zuvor berechnete Outputergebnis des Servers verwerfen, da dieses auf Grundlage einer nicht gültigen Armbewegung berechnet wurde. Die Einstellung *ohne Löschen* enthält diese Funktionalität entsprechend nicht, da der Richtungswechsel am Nullpunkt durch den Trend der Vorhersagemethode vorhergesagt werden kann.

4.2. Vorhersagemethode

Nachdem in *Abschnitt 4.1 Systemmodell* die konzeptionelle Systemarchitektur betrachtet wurde, erfolgt in diesem eine genaue Betrachtung der verwendeten Vorhersagemethode, die zuvor ausgeklammert wurde. Dessen Implementierung erfolgt gesondert in *Abschnitt 5.1 Vorhersagemethode*.

Es wird eine Vorhersagemethode benötigt, die aus einer Reihe von m bekannten Vergangenheitswerten, den ersten bis n-ten zukünftigen Wert vorhersagen kann. Hierzu wird die *erweiterte gedämpfte Holt‘s Methode* verwendet. Als Grundlage wird jedoch zunächst die gedämpfte Holt‘s Methode betrachtet, die diese erweitert. Bei dieser handelt es sich um ein lineares Regressionsmodell zur Zeitreihenanalyse, welches eine exponentielle Glättung verwendet. Da die Holt‘s Methode, welche

von der gedämpften Holt's Methode erweitert wird, aufgrund der linearen Erweiterung des Trends zum Zeitpunkt der Modellbildung, den Vorhersagewert mit zunehmender Zeit zu stark ansteigen lässt, verwendet die gedämpfte Holt's Methode zusätzlich einen Dämpfungsparameter für den Trend [Tay03].

$$\hat{y}_{t+n|t} = l_t + \sum_{n=1}^{n} \phi^n b_t \quad (4.1)$$

$$l_t = \alpha y_t + (1 - \alpha)(l_{t-1} + \phi b_{t-1}) \quad (4.2)$$

$$b_t = \beta(l_t - l_{t-1}) + (1 - \beta)\phi b_{t-1} \quad (4.3)$$

Die Formel aus *Formel 4.1* stellt die eigentliche Berechnung des vorhergesagten Wertes zum Zeitschritt t, für den Zeitschritt $t+n$, mit $n \geq 1$ dar. Dieser setzt sich aus dem Levelwert l_t und dem n Schritte in die Zukunft fortgesetzten Trendwert b_t zusammen. Hierbei wird der Trendwert aufgrund des bereits oben erwähnten zu starken Anstieges des vorhergesagten Wertes bei rein linearer Erweiterung des Trends, noch um einen Parameter ϕ gedämpft. D.h. ausgehend von l_t, dem ermittelten Levelwert für den Zeitschritt t, wird der Trendwert geglättet n-Fach auf diesen addiert.

Der Levelwert l_t aus der folgenden Hilfsgleichung wird durch y_t, den Inputwert zum Zeitpunkt t bestimmt. Der Inputwert y fließt mit einem Gewicht von α in die Levelgleichung ein und mit einem Faktor von (1 - α) setzt sich dieser aus der Summe der vorangegangenen Level- und Trendwerte zusammen. Hierbei wird der vorangegangene Trendwert noch gedämpft, um einen zu starken Anstieg zu verhindern. Zusammengefasst stellt der Levelwert l_t, den ermittelten bzw. geschätzten Inputwert mit dessen dazugehörigem Trend zum Zeitpunkt t dar.

Der Trendwert b_t setzt sich aus dem vorangegangenen Level- und Trendwert zusammen. Mit einem Gewicht von β wird die Differenz des aktuellen und vorangegangenen Levelwertes berücksichtigt und einem Gewicht von (1 - β) der vorangegangene Trend gedämpft übernommen. D.h. der Trend b_t als ermittelte Veränderung zwischen den Levelwerten zu den Zeitpunkten t und $t - 1$, stellt den Grad der Veränderung der ermittelten/geschätzten Inputwerte dar. Da sich die ermittelten Levelwerte, aufgrund der nicht (immer) exakt linear verändernden Inputwerte abweichen können, wird anteilig der Trend zum vorangegangenen Zeitpunkt $t - 1$ mit einbezogen.

Es gilt für die Glättungsparameter $0 \leq \alpha, \beta \leq 1$, sowie $0 < \phi < 1$ für den Dämpfungsparameter.

Da die gedämpfte Holt's Methode in einigen Szenarien immer noch einen zu starken Anstieg aufweist, haben [BM22] diesen um einen weiteren Beschleunigungsparameter zur weiteren Glättung ergänzt. Die daraus resultierende *erweiterte gedämpfte Holt's Methode*, die nachfolgend vorgestellt wird, stellt die ausschließliche in dieser Bachelorarbeit verwendete Vorhersagemethode dar.

$$(4.4)\quad \hat{y}_{t+n|t} = l_t + b_t n + c_t * \sum_{n=1}^{n} \phi^n$$

$$(4.5)\quad l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1} * dt + \phi c_{t-1} * \frac{dt^2 - dt}{2})$$

$$(4.6)\quad b_t = \beta(\frac{l_t - l_{t-1}}{dt}) + (1-\beta)(b_{t-1} + \phi c_{t-1} * dt)$$

$$(4.7)\quad c_t = \gamma(\frac{b_t - b_{t-1}}{dt}) + (1-\gamma)\phi c_{t-1}$$

Zusätzlich zum Levelwert l und Trendwert b wird noch ein Beschleunigungsparameter c, der für eine weitere Glättung sorgt für die Berechnung des vorhergesagten Wertes $\hat{y}_{t+n|t}$ verwendet. Damit ändert sich die Gleichung dahingehend, dass der Trendwert linear auf den Levelwert dazugerechnet wird und der Beschleunigungsparameter durch eine Summenformel über n geglättet wird. Die beiden Hilfsgleichungen aus der vorangegangenen Formel aus *Formel 4.1* ändern sich analog mit, dass nicht mehr der Trendwert geglättet wird. Abschließend wird der Beschleunigungswert c durch die Summe aus Trendentwicklung und der vorangegangenen gedämpften Beschleunigung berechnet. D.h. der Beschleunigungsparameter c_t stellt die Veränderung der Geschwindigkeiten zu den Zeitpunkten t und $t-1$ dar. Da sich die Geschwindigkeit nicht (immer) exakt linear verändert, wird anteilig über den Parameter γ noch die Geschwindigkeit zum zurückliegenden Zeitpunkt $t-1$ mit berücksichtigt. Der dt Parameter wurde eingeführt zur Trendkorrektur bei verspäteten Updates. Da verspätete Updates im Rahmen eines simulierten verteilten Systems nicht auftreten können, wird der Parameter ignoriert und nicht näher behandelt.

4.3. Parameterbestimmungen

In diesem abschließenden Abschnitt wird zunächst rekapituliert, welche Parameter es zu bestimmen gilt und worauf bei deren Setzung geachtet werden muss. Wie auch in den vorangegangenen Abschnitten, wird die Implementierung gesondert in *Abschnitt 5.2 Parameterbestimmungen* betrachtet.

Im Falle der Glättungs- und Dämpfungsparameter α, β und ϕ, wurden die Forschungsergebnisse von [BM22] herangezogen. Diese wurden dort für den selben Anwendungsfall bestimmt und gelten hier entsprechend. Konkret wurden $\alpha = 0{,}93$, $\beta = 0{,}86$ und $\phi = 0{,}92$ übernommen.

Ein möglichst hohes $\alpha \to 1$ ergibt sich dadurch, dass der Levelwert maßgeblich von einem neuen Inputwert beeinflusst wird. Da dieser neue Wert u.U. auch stark von einem eventuell erwarteten abweichen kann, sich also nicht (zwangsläufig) linear über die Zeit verändert, würde ein kleines α und damit eine Überbewertung des Trends bei möglicherweise zwei aufeinanderfolgenden Inputwerten, die abweichen, die Differenz zwischen erwartetem und tatsächlichem Ergebnis stark verfälschen.

Für β wird ebenfalls die Verwendung eines hohen Wertes empfohlen, wenn auch nicht ganz so hoch wie das für α gewählte. Damit werden die Levelwerte, die hauptsächlich aus den Inputwerten beeinflusst werden, stärker berücksichtigt und lediglich mit einem kleinen Anteil die Beschleunigung als Gradmesser für den Trendwert herangezogen.

Die Bestimmung von ϕ findet in einem engeren Auswahlbereich statt. Zunächst einmal werden die Werte 0 und 1 ausgeschlossen, da ein Wert von 0 den eingeführten Beschleunigungsparameter obsolet machen würde. Ein Wert von 1 dagegen würde zur ungedämpften Holt's Methode führen und $\phi > 1$ zu einem exponentiellen Trend [Tay03]. Es sollte für ϕ ein Wert nahe unterhalb von 1 gewählt werden [HA18]. Konkret wurde der Wert $\phi = 0{,}92$ übernommen [BM22].

Die noch fehlenden Parameter, die bestimmt werden müssen, sind γ, m, sowie die *Schwellenwerte*.

Je konstanter die Geschwindigkeit der Veränderungen ist, desto höher sollte γ gewählt werden. Umgekehrt sollten bei einer unkonstanteren Veränderung die vergangenen Beschleunigungswerte herangezogen werden.

Der Parameter m besagt, wie groß die Historie gewählt wird, d.h. die wieviel letzten Inputwerte für die Berechnung herangezogen werden sollen. Mit steigendem m werden zwar mehr vorangegangene Inputwerte für die Berechnung berücksichtigt, jedoch mit einem exponentiell abnehmenden Gewicht, der zum Ergebnis addiert wird. D.h. ein zu großes m würde sich im zweifelsfall nicht nur nicht lohnen, sondern würde über den, mit optimaler Länge m ermittelten, vorhergesagten Wert hinaus schießen. Ein zu kleines m würde umgekehrt unterhalb des optimalen vorhergesagten Wertes bleiben.

Abschließend besagt der Parameter *Schwellenwert*, wie groß die Abweichung zwischen dem vorhergesagten und tatsächlichen Inputwert sein darf, bevor ein neues Vorhersagemodell (mit dem aktuellen, nicht mehr innerhalb eines Schwellenwertes korrekt vorhergesagten Inputwertes als neuester Historienwert) gebildet werden muss.

Die vorher genannten Parameter α, β und ϕ wurden mit den Werten $\alpha = 0{,}93$, $\beta = 0{,}86$ und $\phi = 0{,}92$ übernommen und für die Berechnung der fehlenden drei Parameter fixiert. Alle drei Parameter γ, m und *Schwellenwert* wurden für jeden einzelnen Input- und Outputparameter individuell bestimmt. Wie diese letztendlich genau berechnet wurden, wird in *Abschnitt 5.2 Parameterbestimmungen* ausgeführt.

5. Implementierung

In diesem Kapitel erfolgt die Implementierung der zuvor in *Kapitel 4 Entwurf* betrachteten konzeptionellen Grundlagen dieser Arbeit. Zu Beginn wird die Implementierung der Vorhersagemethode aufgezeigt. Im anschließenden Abschnitt wird die Implementierung der noch zu ermittelnden Parameter γ, m und *Schwellenwert* aufgezeigt. Den Abschluss bildet das verteilte System.

5.1. Vorhersagemethode

In diesem Abschnitt wird die Implementierung des Vorhersagemodells betrachtet, einschließlich der Rekonstruktion dieser aus seinen drei zugrundeliegenden Modelldaten. Die Implementierung der Vorhersagemethode wird der Parameterbestimmung und dem verteilten System vorangestellt, da diese in beiden verwendet wird.

5.1.1. Aufbau Vorhersagemodell

Zunächst wird die Implementierung des Aufbaus des Vorhersagemodells betrachtet.

In *Listing 5.1* wird das Vorhersagemodell implementiert. Die Methode bekommt die Historie und einen Integer, der Aufschluss darüber gibt, zu welchem Input- bzw. Outputparameter die Historie gehört, übergeben und berechnet neben dem Aufbau des Vorhersagemodells, den Vorhersagewert für den ersten zukünftigen Zeitschritt.

```
1  def extendedDampedHolts_BuildModel(data, input_parameter):
2      global valueList
3      global trend_List
4      global accelerationList
5      global trendValues
6
7      trendValues[input_parameter] = 1
8
9      trendList = []
10     levelList = []
11     accelerationCalculationList = []
12     foreCastList = []
13
14     levelList.append(data[0])
15     trendList.append(0)
16     accelerationCalculationList.append(0)
17     foreCastList.append(0)
18     foreCastList.append(levelList[0] + trendList[0] + accelerationCalculationList[0])
19
```

```
20      for x in range(1, len(data)):
21          levelList.insert(x, ((alpha * data[x]) + ((1 - alpha) * (levelList[x - 1] + (trendList
    [x - 1] * dt) + (phi * (accelerationCalculationList[x - 1] * ((math.pow(dt, 2) - dt) / 2))))))
    )
22          trendList.insert(x, ((beta * (levelList[x] - levelList[x - 1]) / dt) + ((1 - beta) *
    ((trendList[x - 1]) + (phi * accelerationCalculationList[x - 1] * dt)))))
23          accelerationCalculationList.insert(x, ((parameter_gamma[input_parameter] * (trendList[
    x] - trendList[x - 1]) / dt) + ((1 - parameter_gamma[input_parameter]) * (phi *
    accelerationCalculationList[x - 1]))))
24          foreCastList.insert(x + 1, (levelList[x] + trendList[x] + accelerationCalculationList[
    x]))
25
26      foreCastList.__delitem__(-1)
27
28      foreCastValue = levelList[len(levelList) - 1] + (trendList[len(trendList) - 1]) + (phi *
    accelerationCalculationList[len(accelerationCalculationList) - 1])
29
30      valueList[input_parameter] = levelList[len(levelList) - 1]
31      trend_List[input_parameter] = trendList[len(trendList) - 1]
32      accelerationList[input_parameter] = accelerationCalculationList[len(
    accelerationCalculationList) - 1]
33
34      return foreCastValue
```

Listing 5.1: Aufbau des Vorhersagemodells

Der Parameter *trendValues* aus Zeile 7 wird vor dem Aufbau für diesen Input- bzw. Outputparameter wieder auf 1 zurücksetzt. Dieser besagt, der wievielte zukünftige Zeitschritt berechnet werden soll, was beim Aufbau des Vorhersagemodells der erste ist. Der Parameter wird in dieser Methode selbst noch nicht gebraucht, aber in der nächsten betrachteten Vorhersagemethode in *Unterabschnitt 5.1.2 Rekonstruktion des Vorhersagemodells*, die das Vorhersagemodell rekonstruiert bzw. den zweiten bis n-ten zukünftigen Zeitschritt, sowohl auf Sender- und Empfängerseite berechnet.

Anschließend werden in den Zeilen 9-18 die lokalen für die Berechnungen benötigten Datenstrukturen initialisiert und mit ihren Startwerten befüllt. Die Levelliste bekommt als ältesten Eintrag, den ältesten Historienwert übergeben. Der Trend und die Beschleunigung sind an der Stelle 0, da keine weiter zurückliegenden Einträge als dem ältesten Historienwert bekannt sind. Von dieser Grundlage aus, erfolgt die weitere Berechnung. (Die Datenstruktur *foreCastList* enthält die vorhergesagten Werte, die rückwirkend für die vergangenen Historienschritte berechnet werden und kann vollständig ignoriert werden.)

Der eigentliche Algorithmus läuft von Zeile 20 bis 24 ab. Hier werden nacheinander vom zweitältesten bis neuesten Historienwert der Level, Trend und die Beschleunigung für den jeweiligen Zeitschritt berechnet. Die Implementierung stellt eine analoge Umsetzung der in *Formel 4.4* erläuterten Vorhersagemethode dar und wird hier deshalb nicht wiederholt. Der dt Parameter wurde hardgecoded mit 1 bezüglich einer eventuellen zukünftigen Erweiterung der Implementierung in der Formel belassen.

In Zeile 28 wird der erste zukünftige Wert berechnet. Dieser setzt sich aus der Summe von Level, Trend und Beschleunigung zusammen, wobei die Beschleunigung noch mit dem ϕ Parameter gedämpft wird. Die Rückgabe des ersten zukünftigen Wertes erfolgt in Zeile 34.

Abschließend werden in den Zeilen 30-32 der aktuellste Level als Stützwert, der aktuellste Trend und die aktuellste Beschleunigung als Modelldaten für diesen Parameter zugewiesen aus denen das Vorhersagemodell wieder rekonstruiert werden kann.

5.1.2. Rekonstruktion des Vorhersagemodells

Die Rekonstruktion des in *Listing 5.1* gebildeten Vorhersagemodells auf Empfängerseite, sowie die Berechnung der zweiten bis *n*-ten zukünftigen Werte, sowohl auf Sender- und Empfängerseite, erfolgt mittels der dort erstellten Modelldaten des Vorhersagemodells.

```
1  def extendedDampedHolts_NextValue(value, trend, acceleration, input_parameter):
2      global trendValues
3
4      trendValues[input_parameter] += 1
5
6      phiValue = 0.0
7
8      for x in range(1, (trendValues[input_parameter] + 1)):
9          phiValue += math.pow(phi, x)
10
11     foreCastValue = value + (trendValues[input_parameter] * trend) + (phiValue * acceleration)
12
13     return foreCastValue
```

Listing 5.2: Rekonstruktion des Vorhersagemodells

In *Listing 5.2* wird die Rekonstruktion und die Verwendung des Vorhersagemodells über den ersten zukünftigen Zeitschritt hinaus dargestellt. Die Methode bekommt als Parameter die Modelldaten (Stützwert/Level, Trend und Beschleunigung), sowie einen Integer übergeben, der aussagt auf welchen Input- bzw. Outputparameter sich das Vorhersagemodell bezieht.

Der Parameter *trendValues* aus Zeile 4 besagt, der wievielte zukünftige Wert dieses Vorhersagemodells berechnet werden soll. In der Aufbaumethode in *extendedDampedHolts_BuildModel* aus *Listing 5.1* wurde dieser mit 1 initialisiert (, da dort der erste zukünftige Wert berechnet wurde). Für den Zweiten, wie auch jeden weiteren zukünftigen Zeitschritt, deren Berechnung in *extendedDampedHolts_NextValue* aus *Listing 5.2* erfolgt, wird dieser Parameter vor der Berechnung um 1 inkrementiert.

Überdies muss auf Empfängerseite *trendValues* mit 0 initialisiert werden, statt mit 1 auf Senderseite in *extendedDampedHolts_BuildModel* aus *Listing 5.1*, damit zeitgleich dieselben Vorhersagen auf Empfänger- und Senderseite gespiegelt laufen und nicht um einen Zeitschritt versetzt. Eine Initialisierung mit 1 auf Empfängerseite, würde bei erstmaliger Benutzung von *extendedDampedHolts_NextValue* durch die Inkrementierung um 1 in Zeile 4 also den zweiten Zeitschritt berechnen, weshalb es mit 0 initialisiert werden muss, um durch die Inkrementierung um 1 in Zeile 4 den ersten Zeitschritt zu berechnen. Die Methode *extendedDampedHolts_BuildModel* aus *Listing 5.1* wird auf Empfängerseite logischerweise nie aufgerufen, da dort keine Vorhersagemodelle gebildet werden, sondern ausschließlich von Senderseite erhaltene, verwendet.

Der vorhergesagte Wert für den Zeitschritt *trendValues* wird in Zeile 11 berechnet und in Zeile 13 zurückgegeben. Die Berechnung des vorhergesagten Wertes setzt sich wie auch in *Listing 5.1* aus der Summe der drei Modelldaten (Stützwert/Level, Trend und Beschleunigung) zusammen. Der

Trend wird um die Anzahl der in der Zukunft liegenden Schritte multipliziert. Dieser konnte bei der Aufbaumethode in *extendedDampedHolts_BuildModel* aus *Listing 5.1* weggelassen werden, da dieser für den ersten zukünftigen Zeitschritt 1 ist. Darüber hinaus unterscheidet sich die Berechnung von ϕ (Zeile 6-9). Der Dämpfungsparameter ϕ wird über die Anzahl der zukünftigen Zeitschritte hinweg gedämpft. Da bei der Aufbaumethode nur der erste zukünftige Zeitschritt vorhergesagt wird, konnte dort einfach der Wert für ϕ für die Berechnung herangezogen werden. Bei einer Vorhersage über mehrere Zeitschritte in die Zukunft, muss ϕ jedoch aufsummiert werden, wobei der Exponent um den ϕ exponiert wird mit jedem Summanden um 1 inkrementiert wird.

Es existiert überdies auch eine Vorhersagemethode *extendedDampedHolts_ListNextNValues*, die aus den Modelldaten statt immer nur den nächstweiteren zukünftigen Zeitschritt, eine Liste mit den n zukünftigen Werten zurückgibt. Da diese Methode jedoch in der Arbeit nie aufgerufen wurde, wird diese hier nicht vorgestellt.

5.2. Parameterbestimmungen

Nachdem das Vorhersagemodell implementiert wurde, wird in diesem Abschnitt die Implementierung der Bestimmung der noch fehlenden Parameter γ, m und *Schwellenwert* aufgezeigt. Die anderen Parameter wurden mit $\alpha = 0{,}93$, $\beta = 0{,}86$ und $\phi = 0{,}92$ aus [BM22] übernommen.

5.2.1. Berechnung von γ und m

Zunächst werden die Berechnungen von γ und m alleine für sich behandelt und anschließend wird der iterative Ansatz der gemeinsamen Berechnung dieser aufgezeigt.

Berechnung von γ

In *Listing 5.3* wird die Berechnung von γ gezeigt. Die Methode bekommt ein Array mit Testdatensatzpfaden, dem m zu dem das beste γ ermittelt werden soll und einen Integer, um welchen Input- bzw. Outputparameter es sich handelt übergeben.

```
1  def calculatingGamma(testDataPaths, m, input_parameter):
2      ...
3      for x in range(0, len(testDataPaths)):
4          data_set = testDataPaths.__getitem__(x)
5          readTestData(testDataPaths.__getitem__(x))
6
7          gamma = 0.00
8
9          gamma_best = 0.0
10         average_mmse_best = sys.float_info.max
11
12         for y in range(0, 99):
13             mmse_all_CurrentDataStructure = []
14
15             gamma += 0.01
16
```

```
17            for z in range(m, (len(dataList[input_parameter]) - n)):
18                extendedDampedHolts_BuildModel(dataList[input_parameter][(z - m):z], z, gamma,
    input_parameter)
19
20            average_mmse = 0.0
21            for z in range(0, len(mmse_all_CurrentDataStructure)):
22                average_mmse += mmse_all_CurrentDataStructure[z]
23            average_mmse = average_mmse / (len(mmse_all_CurrentDataStructure))
24
25            if (average_mmse < average_mmse_best):
26                gamma_best = gamma
27                average_mmse_best = average_mmse
28    ...
```

Listing 5.3: Berechnung von γ

Die nachfolgende Berechnung für γ wird für jeden Testdatensatz individuell durchgeführt (Zeile 3), aus denen dann anschließend ein Durchschnitt, gewichtet nach der Anzahl der Iterationsschritte in diesen berechnet wird (, um verhältnismäßig kleine Testdatensätze nicht stärker zu gewichten als größere). Falls nur ein Testdatensatz verwendet werden soll, ist dessen Gewicht entsprechend 1.

Der Parameter γ wird mit 0,00 initialisiert (Zeile 7). Anschließend wird für jedes mögliche γ (Zeile 12) mit einer Genauigkeit von zwei Nachkommastellen (Zeile 15) über den gesamten Testdatensatz, abzüglich der Historiengröße m und dem Vorhersagehorizont n=4, während derer keine Vorhersagen statt finden iteriert (Zeile 17). Für die Vorhersagen werden immer die vorangegangenen m Inputwerte vom aktuellen Inputwert aus verwendet (Zeile 18).

Nach dem Durchlauf für ein bestimmtes γ wird der Durchschnitt der mean modified squared error (mmse) für alle Vorhersagen gebildet (Zeile 20-23). Im Anschluss daran findet eine Abfrage statt, ob der durchschnittliche Fehler, der mit diesem γ ermittelt wurde, kleiner ist als der bisher kleinste durchschnittliche Fehler für diesen Testdatensatz (Zeile 25-27). Falls dies der Fall ist, wird das beste γ durch das aktuelle γ ersetzt und ebenso der durchschnittliche kleinste Fehler für diesen Testdatensatz (Zeile 25-27).

Zusammengefasst geht es bei der Berechnung von γ, um die größtmögliche Minimierung des Fehlers mmse, bei ansonsten fixierten Parametern, die die Anforderungen aus *Unterabschnitt 4.1.1 Systemarchitektur* erfüllen. Es wird also versucht gegensätzliche Ziele zu erreichen, nämlich durchschnittliche Vorhersagedauern und Fehlertoleranz. Die Vorhersagedauern werden auf den anzustrebenden Wert von 4 Zeitschritten fixiert und dafür versucht jene Parameter zu finden, die den kleinsten Fehler verursachen. Je länger die Vorhersagen im Durchschnitt halten sollen, desto großzügiger müssten die Parameter angepasst werden. Damit würde später auch insgesamt eine Verschlechterung der Simulationsergebnisse des verteilten Systems einhergehen.

Berechnung von m

Die Ermittlung von m wird in *Listing 5.4* aufgezeigt. Die Methode erhält als Übergabeparameter ein Array mit Testdatensatzpfaden, das γ zu dem das beste m berechnet werden soll, sowie auf welchen Input- bzw. Outputparameter sich die Bestimmung von m bezieht.

```
def calculatingM(testDataPaths, gamma, input_parameter):
    ...
    for m in range(m_start, 1, -1):
        mmse_m_best = [sys.float_info.max] * len(testDataPaths)

        weightsDataSets = [0 for x in range(len(testDataPaths))]

        for x in range(0, len(testDataPaths)):
            data_set = testDataPaths.__getitem__(x)
            readTestData(testDataPaths.__getitem__(x))

            mmse_all_CurrentDataStructure = []

            for z in range(m, (len(dataList[input_parameter]) - n)):
                extendedDampedHolts_BuildModel(dataList[input_parameter][(z - m):z], z, gamma,
 input_parameter)

            average_mmse = 0.0
            for z in range(0, len(mmse_all_CurrentDataStructure)):
                average_mmse += mmse_all_CurrentDataStructure[z]
            average_mmse = (average_mmse / len(mmse_all_CurrentDataStructure))

            mmse_m_best[x] = average_mmse
    ...
```

Listing 5.4: Berechnung von m

Die Berechnung erfolgt für alle Werte von m=10 bis m=2 (Zeile 3). Die Werte 10 und 2 wurden als Ober- und Unterschranke für m festgelegt. Der Parameter für m wurde mit 10 gedeckelt, da die Ausführung bereits zeigte, dass ein so hohes m sich nie einstellt. Dies wird später in der Evaluation auch in *Unterabschnitt 6.3.1 Parameter* nochmal aufgegriffen und bestätigt. Die untere Schranke wurde bei m=2 festgelegt, da durch eine Unterschranke von m=1, die Vorhersagemethode aus *Formel 4.4* nicht richtig verwendet werden würde. Falls für einen Input- bzw. Outputparameter m=1 ermittelt wird, würde bei einer Vorhersage dessen, immer nur der letzte Historienwert selbst aus dem er erstellt wurde zurück gegeben werden, ohne die Berücksichtigung irgendeines Trends oder einer Beschleunigung.

Anschließend wird der durchschnittliche mmse für jeden Testdatensatz (Zeile 8) mit diesem m berechnet, bei der ganz zum Schluss gewichtet über die Größe der Testdatensätze ein Durchschnitt über alle Testdatensätze hinweg gebildet wird. Es wird über den gesamten Testdatensatz abzüglich der Historiengröße m und dem Vorhersagehorizont n=4 iteriert (Zeile 14). Als Historie für die Vorhersagen werden die letzten m Inputwerte vor dem aktuellen Inputwert herangezogen (Zeile 15).

Zum Schluss wird der durchschnittliche mmse für diesen Testdatensatz und m berechnet (Zeile 17-20) und als Fehler diesem Testdatensatz und m zugewiesen (Zeile 22).

Wie auch zuvor bei der Berechnung von γ kann abschließend festgehalten werden, dass das Ziel darin besteht unter Einhaltung der geforderten Anforderungen aus *Unterabschnitt 4.1.1 Systemarchitektur* den Fehler soweit wie möglich zu minimieren.

Iterative Berechnung von γ und m

Die Parameterbestimmungen für γ und m erfolgen gemeinsam in einem iterativen Ansatz. Die zugrundeliegende Idee ist es, immer einen von beiden Parametern zu fixieren und zu prüfen, was der dazugehörige beste Wert für den nicht fixierten Parameter ist und dies solange zu betreiben bis sich keine Verbesserung mehr einstellt bzw. überhaupt einstellen kann.

In *Listing 5.5* wird die iterative Berechnung von γ und m gezeigt. Die Methoden zur eigentlichen Berechnung von γ und m, wurden im Vorfeld gezeigt.

```
def calculatingParametersDataSetPaths(testDataPaths, prediction):
    ...
        m_values = [m_start, ]

        gamma_values = [gamma_start, ]

        for x in range(0, numberParameters):
            i = 0
            while True:
                gamma_values.append(calculatingGamma(testDataPaths, m_values[i], x))

                m_values.append(calculatingM(testDataPaths, gamma_values[i + 1], x))

                alreadyFoundBestM = False

                for y in range(0, len(m_values) - 1):
                    if ((m_values[y]) == (m_values[len(m_values) - 1])):
                        alreadyFoundBestM = True

                if (alreadyFoundBestM == True):

                    best_mmse = sys.float_info.max
                    best_offset = 0
                    for y in range(0, len(mmse_all_best)):
                        if (mmse_all_best[y] < best_mmse):
                            best_mmse = mmse_all_best[y]
                            best_offset = y

                    parameter_gamma.append(gamma_values[best_offset])
                    parameter_m.append(m_values[best_offset])

                    break
                else:
                    i = i + 1

            m_values = [m_start, ]
            gamma_values = [gamma_start, ]
            mmse_all_best = [sys.float_info.max, ]

            ...
        saveResultsToFile()
```

Listing 5.5: Iterative Berechnung von γ und m

Die Methode bekommt ein Array mit beliebig vielen Pfaden von Testdatensätzen übergeben und mit *prediction*, einen Integer der angibt, ob die Berechnung für die Input- oder Outputvorhersage erfolgt, um zu bestimmen wohin die Ergebnisse gespeichert werden sollen. Eine alternative Methode *calculatingParametersFolder* bekommt statt eines Arrays mit beliebig vielen Testdatensatzpfaden, den Ordner, in dem sich alle Testdatensätze befinden, die für die Berechnung berücksichtigt werden sollen, arbeitet aber ansonsten exakt identisch zur hier vorgestellten.

Die Datenstrukturen aus Zeile 3 und 5 enthalten zu Beginn den Startwert, ab dem gerechnet werden soll. Diese sind m=10 und γ=0,00. Der Parameter für m wurde mit 10 gedeckelt, wie bereits in der vorangegangenen Betrachtung der Berechnung von m erläutert wurde.

Die nachfolgende iterative Berechnung erfolgt für jeden Input- bzw. Outputparameter individuell (Zeile 7).

In Zeile 10 erfolgt die Berechnung für das beste γ, für das das aktuell betrachtete m, welches initial 10 ist, fixiert wird. Anschließend wird in Zeile 12, umgekehrt γ fixiert, um für dieses beste gefundene γ das dazugehörige beste m zu finden. Diese iterative Berechnung geht solange weiter bis ein m mehfach in der Datenstruktur enthalten ist, womit ein Zyklus detektiert wird, es also zu keiner weiteren Verbesserung kommen kann. Mit einem bereits gefundenen m erneut zu rechnen, würde zum exakt selben Ergebnis führen, da die nachfolgenden γ und m Paare sich zyklisch immer wiederholen würden, weshalb dann abgebrochen wird.

Die Überprüfung, ob ein Zyklus vorliegt, erfolgt in den Zeilen 16-18. Für den Fall, dass das beste m noch nicht ermittelt wurde, erfolgt die weitere Berechnung mit dem nächsten bzw. letzten gefundenem m (Zeile 33-34). Für den Fall, dass das beste m gefunden, also ein Zyklus detektiert wurde, wird unter allen ermittelten γ und m-Paaren, jenes ausgewählt, dass den geringsten mmse aufweist, der als Fehlermaß verwendet wird (Zeile 20-30).

$$(5.1) \quad mmse = \frac{1}{4} * \sum_{t=1}^{4} \sqrt{(y_t - \hat{y}_t)^2}$$

Der mmse aus *Formel 5.1* berechnet den Fehler über die ersten 4 zukünftigen Zeitschritte hinweg, der sich aus der Abweichung zwischen y_t, dem aktuellen tatsächlichen Inputwert zum Zeitpunkt t und $\hat{y}_t$, dem vorhergesagten Inputwert zum Zeitpunkt t ergibt. Es wird auf 4 Zeitschritte hin optimiert, da wie in *Unterabschnitt 4.1.1 Systemarchitektur* erläutert wurde, die Vorhersagen im Durchschnitt mindestens über 3 Zeitschritte halten sollen, weshalb der nächstgrößere ganzzahlige Wert genommen wurde.

Es hätte als Fehlermaß genauso der mean squared error (mse) verwendet werden können, bei dem die Wurzel nicht aus den Summanden gezogen wird. Die Entscheidung für den mmse erfolgte ausschließlich aus dem Grund, um während der Implementierung die Ergebnisse bzw. die Korrektheit der implementierten Berechnungen besser nachvollziehen zu können, da die Abweichungen (y_t - $\hat{y}_t$) winzig sind und das ziehen derer Wurzel diese erkennbarer macht. Der mmse kann bedenklos an Stelle des mse verwendet werden, da dieser die Verhältnisse zwischen den Ergebnissen nicht ändert. Das beste bzw. schlechteste Ergebnis gemäß mse, bleibt auch bei alternativer Anwendung des mmse das beste bzw. schlechteste Ergebnis.

Abschließend werden vor dem Durchlauf für den nächsten Input- bzw. Outputparameter alle lokalen Datenstrukturen wieder zurückgesetzt (Zeile 36-38). Nachdem alle Input- bzw. Outputparameter ermittelt wurden, werden die Ergebnisse in einer .txt Datei gespeichert (Zeile 41).

5.2.2. Schwellenwert

Nachdem sämtliche Parameter für die Verwendung der Vorhersagemethode aus *Formel 4.4* bestimmt wurden, müssen abschließend nur noch die Schwellenwerte für die einzelnen Input- bzw. Outputparameter ermittelt werden. Diese geben Auskunft darüber, ab wann (Überschreitung des Schwellenwertes) ein neues Vorhersagemodell erstellt werden muss.

Die grundsätzliche Idee des Verfahrens ist sehr ähnlich zu der zuvor betrachteten Bestimmung von γ. Es werden die Anforderungen aus *Unterabschnitt 5.2.1 Berechnung von γ und m* fixiert und unter Einhaltung dieser versucht den Schwellenwert soweit es geht abzusenken. Auch hier wird versucht den kleinstmöglichen Schwellenwert zu ermitteln, für den die Vorhersagen im Durchschnitt noch 4 Zeitschritte halten. Hierzu muss gesagt werden, dass die Ermittlung des Schwellenwertes, der auf eine Vorhersagedauer von 4 Zeitschritten hin optimiert wird, das Minimum darstellt, ab dem sich der in *Unterabschnitt 4.1.1 Systemarchitektur* erläuterte Ansatz, dass die Modelldaten statt den Inputwerten geschickt werden lohnt. Eine Erhöhung des Schwellenwertes würde für eine Erhöhung der durchschnittlichen Vorhersagedauern und zu einer Absenkung der Qualität führen. Im Rahmen dieses Trade-offs wurde sich für die größtmögliche Qualität entschieden bei dem sich der Ansatz über Modelldaten gerade so noch lohnt.

Startschwellenwert

Hierzu muss zunächst der Startschwellenwert ermittelt werden. Dieser kann zwischen verschiedenen Testdatensätzen, Vorhersagehorizonten und über die verschiedenen Parameter hinweg teilweise sehr stark variieren.

```
1  def thresholdCalculations():
2      ...
3      for k in range(0, len(testDataPaths)):
4          readTestData(testDataPaths[k])
5  
6          upperBoundFound = False
7          thresholdStartFound = False
8          thresholdStart = 0.125
9          thresholdCurrent = thresholdStart
10 
11         while(thresholdStartFound == False):
12             rejected = False
13             tmp = durationCalculation(thresholdCurrent)
14             print("thresholdCurrent: " + str(thresholdCurrent) + ", tmp: " + str(tmp))
15 
16             for m in range(0, len(tmp)):
17                 if (tmp[m] < n):
18                     rejected = True
19 
20             if(upperBoundFound == False):
```

```
21                if(rejected == True):
22                    thresholdCurrent *= 2
23
24                elif(rejected == False):
25                    thresholdCurrent *= 0.5
26                    upperBoundFound = True
27
28            elif (upperBoundFound == True):
29                if (rejected == True):
30                    thresholdCurrent *= 2
31                    thresholdStartFound = True
32
33                elif (rejected == False):
34                    thresholdCurrent *= 0.5
35
36        thresholdStart = thresholdCurrent
37        thresholdDecline = thresholdStart * 0.01
38    ...
```

Listing 5.6: Bestimmung des Startschwellenwertes

In *Listing 5.6* wird zunächst der Startwert für den Schwellenwert bestimmt. Wie bereits in Zeile 3 ersichtlich ist, werden die Schwellenwerte für jeden Parameter individuell, einschließlich innerhalb verschiedener Testdatensätze bestimmt.

Zu Beginn wird ein willkürlicher Wert von 0,125 gewählt (Zeile 8). Die Idee ist es den *upperBound* zu finden, indem man solange diesen ersten Startwert verdoppelt bis alle Vorhersagen im Durchschnitt mindestens den Vorhersagehorizont von 4 Zeitschritten halten. Sollte dies bereits zu Beginn der Fall sein, entsprechend diesen solange zu halbieren bis die Vorhersage zu irgendeinem Parameter im Durchschnitt weniger als den Vorhersagehorizont von 4 Zeitschritten hält und folglich den letzten doppelt so großen zu wählen. Es sollte mit einem so niedrigen Startschwellenwert wie möglich gestartet werden, um die nachfolgende Berechnungszeit minimal zu halten.

Wie lange die Vorhersagen im Durchschnitt halten, wird mittels der Hilfsmethode *durationCalculation* bestimmt (Zeile 13). Da diese sehr ähnlich zur nachfolgenden eigentlichen Schwellenwertbestimmung abläuft, wird hier nicht näher darauf eingegangen, sondern anschließend an der Schwellenwertbestimmung erklärt.

Nachdem der Startschwellenwert gefunden wurde, wird die Absenkungskonstante auf 1% dieses Wertes festgelegt (Zeile 37). Das ist die maximale Genauigkeit des Schwellenwertes (im Verhältnis zum aktuellen Schwellenwert) mit der gearbeitet wird, da immer kleinere Absenkungskonstanten den Rechenaufwand unpraktisch machen.

Schwellenwertbestimmung

Nachdem der Startschwellenwert ermittelt wurde, wird die Bestimmung der Schwellenwerte für die einzelnen Input- bzw. Outputparameter gezeigt.

```
1 def thresholdCalculations():
2     ...
3     while (bestThresholdFound.__contains__(False)):
```

```
4        stepTotal = 0
5        firstPrediction = False
6        historyFull = False
7
8        for x in range(0, len(dataList[0])):
9            for y in range(0, numberParameters):
10               historyValues[y].append(dataList[y].__getitem__(x))
11
12           if (len(historyValues[max_parameter]) == maxLength_m):
13               historyFull = True
14
15           if ((firstPrediction == False) & (historyFull == True)):
16               for z in range(0, numberParameters):
17                   dataListPrediction[z] = extendedDampedHolts_BuildModel(historyValues[z], z
)
18                   dataListPredictionCount[z] += 1
19
20               firstPrediction = True
21
22           elif (firstPrediction == True):
23               for z in range(0, numberParameters):
24                   if (math.sqrt(math.pow(dataList[z].__getitem__(x) - dataListPrediction[z],
 2)) >= (math.sqrt(math.pow(dataList[z].__getitem__(x) * thresholdCurrent, 2)))):
25                           dataListPrediction[z] = extendedDampedHolts_BuildModel(
historyValues[z], z)
26                           dataListPredictionCount[z] += 1
27                     else:
28                       dataListPrediction[z] = extendedDampedHolts_NextValue(valueList[z],
trend_List[z], accelerationList[z], z)
29
30           stepTotal += 1
31    ...
```

Listing 5.7: Bestimmung der Schwellenwerte

In *Listing 5.7* wird die Bestimmung der Schwellenwerte gezeigt. Es handelt sich um eine Dauerschleife, die solange läuft, bis die Schwellenwerte zu allen Input- bzw. Outputparametern gefunden wurden (Zeile 3).

In jedem Iteratonsschritt werden den Historien der Input- bzw. Outputparameter die aktuellen Inputwerte übergeben (Zeile 8-10). Als Datenstruktur für die Historienwerte bieten sich deques an, da für diese eine maximale Länge angegeben werden kann, sodass mit dem Erreichen ihrer maximalen Länge, das Einfügen des nächsten/neuesten Inputwertes, den ältesten Inputwert aus diesem entfernt. Sobald die Historie der längsten Historie vollständig gefüllt ist (Zeile 12-13), werden die ersten Vorhersagemodelle für alle Input- bzw. Outputparameter gleichzeitig erstellt (Zeile 15-20).

Anschließend wird in jedem weiteren Iterationsschritt für jeden Parameter (Zeile 23) überprüft, ob die Abweichung zwischen dem tatsächlichen Inputwert und dem vorhergesagten Inputwert größer ist als der aktuelle Schwellenwert (Zeile 24). Falls dies der Fall ist, wird ein neues Vorhersagemodell zu diesem Parameter erstellt (Zeile 25) und die Anzahl der Vorhersagen für diesen Parameter um

eins erhöht (Zeile 26). Sollte sich die Vorhersage innerhalb des Schwellenwertes befinden, wird lediglich der nächste Inputwert vorhergesagt (Zeile 27-28), ohne ein neues Vorhersagemodell zu bilden. Abschließend erfolgt der nächste Iterationsschritt (Zeile 30).

Die zuvor in der Startschwellenwert angesprochene Hilfsmethode *durationCalculation* bekommt als Parameter den Schwellenwert übergeben, der fix verwendet wird, welcher hier in Zeile 24 variabel läuft.

Abbruchkriterium Schwellenwertbestimmung

Nachdem die Schwellenwertbestimmung für alle Testdatensätze für diesen spezifischen Schwellenwert durchgelaufen sind, muss überprüft werden, ob ein Abbruchkriterium erreicht wurde.

```
def thresholdCalculations():
    ...
    for x in range(0, numberParameters):
        if(bestThresholdFound[x] == False):
            averageDuration = ((stepTotal - maxLength_m) / dataListPredictionCount[x])

            if ((thresholdCurrent == thresholdStart) & (averageDuration < n)):
                bestThreshold[x] = -1
                predictionDuration[x] = averageDuration
                bestThresholdFound[x] = True

            if (averageDuration >= n):
                bestThreshold[x] = thresholdCurrent
                predictionDuration[x] = averageDuration

                if((averageDuration < (n + 0.1)) | (thresholdCurrent <= thresholdDecline)):
                    bestThresholdFound[x] = True

            else:
                    bestThresholdFound[x] = True
    ...
```

Listing 5.8: Abbruchkriterium der Schwellenwerte

In *Listing 5.8* werden die Abbruchkriterien behandelt. Diese prüfen für jeden Parameter (Zeile 3), dessen Schwellenwert noch nicht ermittelt wurde (Zeile 4), ob dieses jetzt erreicht wird. Hierzu wird die durchschnittliche Vorhersagedauer für jeden Parameter aus der Größe des Testdatensatzes abzüglich der Länge des größten m (in diesen ersten m Schritten finden keine Vorhersagen statt) durch die Anzahl der gebildeten Vorhersagemodelle für diesen Parameter bestimmt (Zeile 5).

Zunächst wird geprüft, ob der aktuelle Schwellenwert, dem Startschwellenwert entspricht und ob dabei die durchschnittliche Vorhersagedauer unter dem Vorhersagehorizont von 4 Zeitschritten beträgt (Zeile 7). Falls dies der Fall ist, wird der Variable *bestThreshold[x]* der Wert -1 zugewiesen, was in der späteren Auswertung bedeutet, dass kein Schwellenwert ermittelt werden konnte (Zeile 8). Dies ist eine Bedingung, die niemals auftreten sollte, da der zuvor initiale Schwellenwert so hoch bestimmt wurde, dass jede Vorhersage immer mindestens den Vorhersagehorizont von 4 Zeitschritten halten sollte, egal wie hoch der korrespondierende Schwellenwert dazu auch sein muss.

In einer ursprünglichen Implementierung war der Startschwellenwert hardgecoded und wurde nicht wie in *Listing 5.6* dynamisch zur Laufzeit berechnet. Die Bedingung wurde trotzdem zur etwaigen Identifizierung von Implementierungsfehlern beibehalten.

Als nächstes wird überprüft, ob die durchschnittliche Vorhersagedauer mindestens den Vorhersagehorizont von 4 Zeitschritten beträgt (Zeile 12). Falls dies der Fall ist, werden der aktuelle Schwellenwert und die durchschnittliche Vorhersagedauer gespeichert (Zeile 13-14), sodass mit Erfüllung der Bedingung, dass die Vorhersage über 4 Zeitschritte hält, immer der niedrigere und damit bessere Schwellenwert als bestes Ergebnis festgehalten wird. Da beim nächsten Durchgang in dem die Bedingung eventuell nicht mehr erfüllt ist (Zeile 19), der Schwellenwert und die dazugehörige durchschnittliche Vorhersagedauer natürlich nicht gespeichert werden. Ansonsten wird auch abgebrochen, falls die durchschittliche Vorhersagedauer unter 4,1 Zeitschritte beträgt, um eine ansonsten eventuell sehr lange Rechendauer eines Ergebnisses zu vermeiden, dessen Genauigkeit praktisch keinen Unterschied mehr machen würde. Abgebrochen wird auch, falls der aktuelle Schwellenwert kleiner ist als die Absenkungskonstante (Zeile 16).

Schwellenwertabsenkung

Mit jeder Absenkung des Schwellenwertes, stehen sowohl eine lineare als auch eine multiplikative Absenkung zur Verfügung.

```
1  def thresholdCalculations():
2      ...
3      declineThreshold = False
4      declineThresholdMore = True
5      decliningParameters = [0.9, 0.75, 0.5, 0.25, 0.1]
6      declineFactor = 0.0
7
8      for x in range(0, len(decliningParameters)):
9          tmp = durationCalculation(decliningParameters[x] * thresholdCurrent)
10
11         for m in range(0, len(tmp)):
12             if ((tmp[m] < n) & (bestThresholdFound[m] == False)):
13                 declineThresholdMore = False
14                 break
15
16         if(declineThresholdMore == False):
17             break
18
19         else:
20             declineThreshold = True
21             declineFactor = decliningParameters[x]
22
23     if (declineThreshold == True):
24         thresholdCurrent *= declineFactor
25         thresholdDecline *= declineFactor
26
27     else:
28         thresholdCurrent -= thresholdDecline
```

```
29      ...
```

Listing 5.9: Absenkung der Schwellenwerte

In *Listing 5.9* wird die Schwellenwertabsenkung veranschaulicht. Vor jeder Absenkung des Schwellenwertes wird überprüft, ob linear abgesenkt werden muss (Zeile 3), was langsam in 1% Schritten vom aktuellen Schwellenwert aus erfolgt oder multiplikativ um 10%, 25%, 50%, 75% oder 90% abgesenkt werden kann (Zeile 4-6). Hierzu wird sich wieder der Hilfsmethode *durationCalculation* aus der Startschwellenwertbestimmung bedient (Zeile 9). Es wird überprüft, ob durch eine geringstmögliche der multiplikativen Absenkungen die durchschnittlichen Vorhersagedauern für alle noch nicht bestimmten Parameter, immer noch mindestens den Vorhersagehorizont von 4 Zeitschritten betragen würden (Zeile 11-14). Falls dies der Fall ist, wird mutiplikativ abgesenkt (Zeile 23), ansonsten linear (Zeile 27). Falls multiplikativ abgesenkt werden kann, wird dies für immer größere multiplikative Absenkungen überprüft bis einschließlich 90% (Zeile 8). Falls bei einer der weiteren multiplikativen Absenkungen, die durchschnittliche Vorhersagedauer eines der noch nicht gefundenen Parameter unter 4 Zeitschritte fällt, wird der multiplikative Faktor davor verwendet (Zeile 19-21).

Mit dieser Strategie aus multiplikativer und linearer Absenkung ist es möglich sehr genaue Schwellenwerte (auf 1% des aktuellen Schwellenwertes genau), in einer akzeptablen Rechenzeit zu ermitteln.

Es wurde sich nur für 10%, 25%, 50%, 75% und 90% als multiplikative Absenkungsfaktoren entschieden, da die Hilfsmethode *durationCalculation* für jede Ausführung einiges an Rechenzeit benötigt. Bei noch weniger möglichen Faktoren, würden deutlich mehr Schwellenwertabsenkungen linear durchgeführt werden müssen, was auch in längerer Rechenzeit resultiert. Es stellt den subjektiv bestmöglich ermittelten Kompromiss dar.

5.3. Verteiltes System

Im letzten Abschnitt dieses Kapitels, wird nun die Implementierung des verteilten Systems aufgezeigt. Die Implementierungen aus den vorangegangenen Abschnitten, die sich in diesem wiederfinden, werden kurz ausgeführt, aber ansonsten nicht erneut veranschaulicht und ausführlich erläutert.

5.3.1. Implementierungseinstellungen

Für die Implementierung liegen zwei verschiedene Einstellungen vor, nämlich *mit Löschen* und *ohne Löschen*. Diese stellen die Umsetzung der in *Unterabschnitt 4.1.3 Zusätzliche Anforderungen* beschriebenen Löschstrategien dar.

Gemäß der Löschstrategien werden bei der Einstellung *mit Löschen*, im Falle einer Bewegungsrichtungsänderung des Armes sämtliche Servermodelle, die beim Client vorliegen, gelöscht. Überdies wird von allen neu ankommenden Updates des Servers verlangt, dass diese auf Grundlage eines Clientupdates berechnet wurden, welches diese nach der Bewegungsrichtungsänderung erhalten haben. Ansonsten werden diese verworfen, da diese nicht auf Grundlage einer gültigen Armbewegung berechnet wurden.

Im Falle der Einstellung *ohne Löschen* werden bei einer Bewegungsrichtungsänderung, die beim Client vorliegenden Servermodelle nicht gelöscht und die Updates des Servers werden in jedem Fall nicht deshalb verworfen.

5.3.2. Ablauf

Der Client erhält periodisch mit jedem Iterationsschritt die neuesten Inputwerte aus einer aufgezeichneten Armbewegung, die vor der Simulation aus einer .txt-Datei geparsed wurde. Die angestrebten 30 Frames Pro Sekunde (FPS), in dem alle 33 Millisekunden (ms) der nächste Inputwert dem Client periodisch übergeben werden würde, konnte leider nicht erreicht werden. Die Ausführungszeit der NN ist zu groß, weshalb die Simulation mit 0,25 FPS durchgeführt werden muss, um sicherzustellen, dass die NN mit der Berechnung hinterherkommen und alle Eingaben verarbeitet werden. Die meiste Zeit benötigen die NN mit ca. 0,3 Sekunden unter 0,5 Sekunden für die Verarbeitung was jedoch immer noch viel zu lange dauert, und in einigen Fällen sogar mehrere Sekunden. Deshalb wurde die Simulation auf 0,25 FPS reduziert.

Inputvorhersage des Clients

Mit dem Erhalt der neuesten Inputwerte werden die Parameterhistorien des Clients gefüllt. Sobald die längste Historie vollständig gefüllt ist, werden die ersten Vorhersagemodelle für alle Inputparameter gleichzeitig erstellt und an den Server geschickt.

In jedem weiteren Iterationsschritt wird überprüft, ob der vorhergesagte Inputwert und der tatsächliche Inputwert über den jeweiligen Schwellenwert des jeweiligen Inputparameters hinaus abweichen. Falls dies der Fall ist, wird ein neues Vorhersagemodell für diesen Inputparameter gebildet und eventuell, in Abhängigkeit dieses Inputparameters und der Updatestrategie des Clients auch an den Server geschickt. Falls der vorhergesagte Inputwert weniger als den Schwellenwert vom tatsächlichen Inputwert abweicht, wird einfach nur der Inputwert für den nächsten Zeitschritt vorhergesagt.

Im Falle des Inputparameters *Geschwindigkeit* bzw. *Velocity* wird im Anschluss hieran überprüft, ob sich die Bewegungsrichtung des Armes geändert hat und in diesem Zusammenhang, ob der Arm still steht.

```
1  def inputProcessing(latestInputValue):
2      ...
3          if (z == 1):
4              if ((((historyValues[1][len(historyValues[1]) - 1]) < 0.0) & (
   lastKnownPositiveOrNegativeVelocityValue > 0.0)) | (((historyValues[1][len(historyValues[1]) -
    1]) > 0.0) & (lastKnownPositiveOrNegativeVelocityValue < 0.0))):
5                  if (changedByDirection == False):
6                      print("step: " + str(stepCurrent) + ", changedByDirection = True")
7                      changedByDirection = True
8                      armStandsStill = False
9                      angleLastPosition = historyValues[0][len(historyValues[0]) - 1]
10                     lastKnownPositiveOrNegativeVelocityValue = historyValues[1][len(
   historyValues[1]) - 1]
11
12                     if(deletionStrategy == True):
```

```
13                          client_outputProcessing_Simulation.stepDirectionChanged = stepCurrent
14
15                  else:
16                      armStandsStill = True
17
18              if ((changedByDirection == True) & ((math.sqrt(math.pow(angleLastPosition -
historyValues[0][len(historyValues[0]) - 1], 2)) >= 2.0))):
19                  print("step: " + str(stepCurrent) + ", changedByDirection = False")
20                  changedByDirection = False
21                  armStandsStill = False
22                  angleLastPosition = historyValues[0][len(historyValues[0]) - 1]
23                  lastKnownPositiveOrNegativeVelocityValue = historyValues[1][len(historyValues
[1]) - 1]
24      ...
```

Listing 5.10: Überprüfung Bewegungsrichtungsänderung

Die Implementierung der Bewegungsrichtungsüberprüfung wird in *Listing 5.10* aufgezeigt. Diese Überprüfung findet nur im Fall des Geschwindigkeitsparameters statt (Zeile 3). Der Geschwindigkeitsparameter gibt nicht nur über die eigentliche Geschwindigkeit der Armbewegung Aufschluss, sondern auch über die Richtung dieser. Geht der Arm nach unten ist die Geschwindigkeit positiv. Geht der Arm dagegen runter ist die Geschwindigkeit negativ. Es wird überprüft, ob sich das Vorzeichen der aktuellen Geschwindigkeit und der letzten gemerkten Geschwindigkeit voneinander unterscheiden (Zeile 4).

Ist dies der Fall erfolgt eine nachfolgende Abfrage, ob es die erste Richtungsänderung der aktuellen Armbewegung ist oder nicht. Falls dies die erste Bewegungsrichtungsänderung sein sollte (Zeile 5) wird sich gemerkt, dass sich die Bewegungsrichtung geändert hat, kein Steillstand des Armes detektiert, sowie der aktuelle Winkel und die aktuelle Geschwindigkeit gemerkt (Zeile 7-10). Es werden später 2 verschiedene Implementierungseinstellungen getestet, nämlich eines bei der sämtliche Servermodelle des Clients gelöscht werden, falls sich die Bewegungsrichtung ändert oder nicht. Ob diese Löschstrategie aktiv ist, wird in Zeile 12 überprüft.

In Abhängigkeit von der gewählten Löschstrategie (Zeile 12) werden nachfolgend alle Servermodelle des Clients gelöscht und alle Serverupdates verworfen, deren Berechnung auf Grundlage eines früheren Zeitschrittes erfolgte.

Nachdem überprüft wurde, ob sich die Bewegungsrichtung geändert hat oder nicht, wird im Anschluss überprüft, ob das Merken der Bewegungsrichtungsänderung wieder zurückgesetzt werden kann (Zeile 18). Dies ist dann der Fall, wenn sich die Bewegungsrichtung geändert hat und der aktuelle Winkel sich mehr als eine bestimmte Gradzahl (hier 2 Grad) von jener der ersten Bewegungsrichtungsänderung unterscheidet. Für den Fall das dies zutrifft, wird die Bewegungsrichtungsänderung wieder zurückgesetzt, das Still stehen des Armes wieder gelöst, sowie der aktuelle Winkel und die aktuelle Geschwindigkeit gemerkt (Zeile 20-23).

Clientupdate

Sofern kein Stillstand des Armes detektiert wurde, wird gemäß der Updatestrategie des Clients der Server geupdated. Falls ein Update an den Server erfolgt, wird in jedem Fall der aktuelle Zeitschritt zur Synchronisierung des Servers mitgeschickt. Das ist der fixe Anteil des Updates,

der immer mitgeschickt werden muss. Ansonsten werden die Modelldaten (Stützwert/Level, Trend und Beschleunigung) zu jedem Inputparameter, dessen Update an den Server gesendet werden soll, geschickt. Im Falle der *Individual*-Strategie werden eigentlich gemäß dieser nur die Modelldaten zu allen Inputparametern, die sich geändert haben geschickt. Jedoch werden auch bei der *Individual*-Strategie die Vorhersagemodelle zu allen Inputparametern an den Server geschickt, falls dadurch Bandbreite eingespart werden kann. Eine Bandbreitenersparnis wird in einigen Fällen durch das Verschicken aller Inputparameter deshalb erzielt, weil wenn weniger als alle Inputparameter geschickt werden, hierzu eine Indizesliste mitgeschickt werden muss, auf welchen Inputparameter sich welches Vorhersagemodell bezieht. D.h. falls ((4 * Länge der Indizesliste) ≥ (3 * Anzahl aller Inputparameter)) ist, wird alles an den Server geschickt. Der Ausdruck (3 * Anzahl aller Inputparameter) meint dabei, wenn alle Parameter verschickt werden müssten, da sich jede derer Vorhersagemodelle aus drei Komponenten (Stützwert/Level, Trend und Beschleunigung) zusammensetzt. Falls nur eine Teilmenge, gemäß der *Individual*-Strategie geschickt wird, muss zusätzlich zu jedem Vorhersagemodell noch ein Index mitgeliefert werden, was folglich in Index, Stützwert/Level, Trend und Beschleunigung resultiert.

In diesem Zusammenhang wird noch angemerkt, dass im Falle der *Individual*-Strategie in denen nicht alle Inputparameter bzw. dann später beim Server, alle Outputparameter mitgeschickt werden, es sich also genau genommen nicht um (Vorhersage-)Modelle, sondern nur um (Vorhersage-)Teilmodelle handelt. Dies gilt es zu beachten, da diese ansonsten auch immer als Modelle bzw. Vorhersagemodelle bezeichnet werden.

Für Inputparameter, für die in diesem Iterationsschritt kein neues Vorhersagemodell gebildet wurde, diese aber trotzdem mitgeschickt werden sollen (Updatestrategie (*Master*, *All*), Updatestrategie *Individual* mit allen Inputparametern zur Bandbreitenersparnis) wird vor dem verschicken entsprechend für diese ein neues Vorhersagemodell erstellt.

Server

Nachdem der Server das Update erhalten hat, merkt er sich, zu welchem Zeitschritt dieser das Update erhalten hat. Dies ist deshalb wichtig, da er später beim Update des Clients in jedem Fall zwei Synchronisationsvariablen als fixen Anteil eines jeden Updates immer mitschickt. Dies sind zum einen, zu welchem Zeitpunkt dieser das letzte Update des Clients erhalten hat, womit der Client (für den Fall einer aktiven Löschstrategie) nachvollziehen kann, ob das angekommene Update auf Grundlage einer gültigen Armbewegung berechnet wurde, wie auch dem eventuellen Ersetzen älterer Modelle und zum anderen den Zeitschritt, ab dem das Update verwendet werden darf, der aus dem Ankunftsschritt des Updates beim Server berechnet wird.

Es gilt immer, dass wenn ein Vorhersagemodell beim Client zum Zeitpunkt t gebildet und geschickt wurde, der Server nach Erhalt dessen, bei erstmaliger Verwendung den Output für den Zeitpunkt $t+1$ berechnet, selbst wenn dieses Vorhersagemodell noch zum Zeitpunkt t ankommt. Der Vorhersagewert für den ersten zukünftigen Zeitschritt eines Vorhersagemodells, dass zum Zeitpunkt t gebildet wird, ist der Vorhersagewert für den Zeitpunkt $t + 1$.

Umgekehrt darf ein Vorhersagemodell des Servers, dass bspw. zum Zeitpunkt $t + 1$ gebildet wurde, der Client nach Erhalt dessen, dieses erst zum Zeitpunkt $t + 2$ verwenden, da die Vorhersage des ersten zukünftigen Wertes eines Vorhersagemodells, dass zum Zeitpunkt $t + 1$ gebildet wurde, der Vorhersagewert für den Zeitpunkt $t + 2$ ist. Falls ein Vorhersagemodell, dass zum Zeitpunkt

$t+1$ gebildet, geschickt und auch angekommen ist, bereits zum Zeitpunkt $t+1$ verwendet werden würde, dann würde der Vorhersagewert, der für den Zeitpunkt $t+2$ gilt, bereits zum Zeitpunkt $t+1$ verwendet werden und auch alle folgenden Vorhersagewerte um einen Zeitschritt zu früh verwendet werden und wären damit falsch.

Der Server fängt nach dem Erhalt des allerersten Updates an periodisch (in derselben Geschwindigkeit wie der Client mit 0,25 FPS oder theoretisch möglich, falls die Ausführungszeit der NN es erlauben würde, schneller als der Client) auf Grundlage des aktuellen Updates den Output zu berechnen und periodisch den nächsten Input vorherzusagen und nun für diesen den Output zu bestimmen.

Von den berechneten 2.809 Punkten des Outputergebnisses verwendet dieser jedoch nur 30 indizierte Punkte mit je 3 Koordinaten, also ingesamt 90 Koordinaten, die er als Outputparameter führt und an den Client, gemäß seiner beiden Updatestrategien (*Individual*, *All*) schickt.

Outputvorhersage des Clients

Periodisch mit dem Eintreffen der neuen Inputwerte, gibt der Client den Output aus. Falls kein Servermodell vorhanden ist, berechnet der Client mittels des Backup NN über die Inputwerte ein etwas weniger genaues Ergebnis als das High Quality NN, dass auf dem Server verwendet wird.

Für den Fall, dass der Server ein Update schickt, wird nachfolgend betrachtet, wie der Client diesen verarbeitet.

```
1  def inputReceive(numberParameters_inputPrediction, numberParameters_outputPrediction,
   parameter_gamma_outputPrediction, parameter_m_outputPrediction,
   parameter_threshold_outputPrediction):
2  ...
3  if ((client_receive_Simulation.update == True) & (client_receive_Simulation.
   inputPredictionModelsNumber >= stepDirectionChanged) & (armStandsStill == False)):
4      if((client_receive_Simulation.inputPredictionModelsNumber == inputPredictionModelsNumber)
   | ((client_receive_Simulation.inputPredictionModelsNumber < inputPredictionModelsNumber) &(
   client_receive_Simulation.inputPredictionModelsNumber >= validModelsLastInputPrediction))):
5          if ((client_receive_Simulation.inputPredictionModelsNumber >
   validModelsLastInputPrediction) & (client_receive_Simulation.step <= stepCurrent)):
6              validModels = dict()
7              usingModel = -1
8          stepLatest = client_receive_Simulation.step
9          validModels[stepLatest] = [client_receive_Simulation.indices,
   client_receive_Simulation.values, client_receive_Simulation.trends, client_receive_Simulation.
   accelerations]
10         validModelsLastInputPrediction = client_receive_Simulation.inputPredictionModelsNumber
11
12     client_receive_Simulation.initializeSettings(numberOutputParameters)
13 ...
```

Listing 5.11: Ankunft eines Serverupdates

Die Ankunft eines Serverupdates wird in *Listing 5.11* betrachtet. Zunächst wird in Zeile 3 überprüft, ob ein neues Update des Servers eingetroffen ist. Falls dies der Fall ist, wird direkt mitüberprüft, ob das erhaltene Updatemodell auf Grundlage einer gültigen Armbewegung erfolgt ist, also nachdem eine etwaige Bewegungsrichtungsänderung festgestellt worden ist (inputPredictionModelsNumber

>= stepDirectionChanged). Diese Bedingung ist nur für die Einstellung *mit Löschen* relevant, da *ohne Löschen* die Variable *stepDirectionChanged* immer 0 ist und damit immer zu *wahr* evaluiert. Die letzte Bedingung, die mit erfüllt sein muss, ist dass der Arm nicht still steht. Falls der Arm still steht, wird das letzte verwendete Modell, bevor der Arm still stand (ohne zukünftige Schritte vorherzusagen) ausgegeben, weshalb keine neuen Modelle akzeptiert werden. Außerdem bekommt der Server vom Armstillstand nichts mit und rechnet immer weiter, weshalb die Ergebnisse vermutlich sowieso in jedem Fall unbrauchbar wären.

Falls also diese drei Bedingungen, *Ankunft Update*, *auf gültiger Grundlage berechnet* und *Arm steht nicht still* erfüllt sind, wird das Update nicht sofort verworfen, d.h. aber noch nicht das es auch akzeptiert wird. Ob das Update auch akzeptiert wird, erfolgt in der nächsten Abfrage in (Zeile 4). Hierzu werden zwei Bedingungen überprüft, nämlich ob das Update auf Grundlage des aktuellsten Clientupdates berechnet wurde oder es auf Grundlage eines älteren Clientupdates berechnet wurde, aber immer noch neuer als alle bisher beim Client verwalteten ist. (Die Verwaltung der Modelle wird im Anschluss betrachtet).

Falls diese Abfrage aus Zeile 4 gültig ist, wird das Update verwendet. Vorher wird noch in Zeile 5 überprüft, ob alle bisherigen Modelle, die der Client verwaltet, gelöscht werden sollen (Zeile 6-7). Die Löschung erfolgt, falls das Serverupdate auf Grundlage eines neueren Clientupdates als alle bisherigen erfolgt ist, womit alle bisherigen als veraltet identifiziert werden können (Zeile 5). Überdies muss noch die Bedingung erfüllt sein, dass der Schritt ab dem das Serverupdate verwendet werden darf entweder der aktuelle Zeitschritt ist oder ein bereits vergangener (Zeile 5), sodass es sofort verwendet werden darf. Ein Löschen von allen bisherigen Modellen hätte sonst zur Folge, dass kein aktuell gültiges Servermodell vorhanden wäre und damit auf die Backup-Lösung zurückgegriffen werden müsste.

Es gilt immer, dass ein Vorhersagemodell des Servers, dass bspw. zum Zeitschritt t gebildet wurde, der Client nach Erhalt dessen, dieses erst zum Zeitschritt $t + 1$ verwenden darf. Der Vorhersagewert des ersten zukünftigen Zeitschrittes eines Vorhersagemodells, dass zum Zeitschritt t gebildet wurde, ist der Vorhersagewert für den Zeitschritt $t + 1$. Falls ein Vorhersagemodell, dass zum Zeitschritt t gebildet und geschickt wurde und auch angekommen ist, bereits zum Zeitschritt t verwendet werden würde, dann würde der erste Vorhersagewert, der für den Zeitschritt $t + 1$ gilt, bereits zum Zeitschritt t verwendet werden und auch alle nachfolgenden Vorhersagewerte (2. bis n.) um einen Zeitschritt zu früh verwendet werden und wären damit allesamt falsch.

Das zuweisen des Serverupdates an die Client-Datenstrukturen erfolgt in den Zeilen 8-10. In Zeile 8 wird der Ankunftsschritt dieses letzten Updates gemerkt. In Zeile 9 werden die Modelldaten des Updates an eine *Dictionary* übergeben. Bei einer *Dictionary* handelt es sich um das Python äquivalent einer Java-Hashmap, die also Schlüssel-Wert Paare verwaltet. Der Ankunftszeitschritt stellt den Schlüssel und die Indizes und Modelldaten (Stützwert/Level, Trend und Beschleunigung) den Wert dar. Anschließend wird in Zeile 10 der Zeitschritt, wann der Server sein Update erhalten hat, als das aktuellste verwendete gemerkt.

Löschen von Servermodellen beim Client

Zum Abschluss des Ablaufs wird das Löschen veralteter Servermodelle und die Verwaltung gültiger Servermodelle beim Client behandelt.

```
def inputReceive(numberParameters_inputPrediction, numberParameters_outputPrediction,
parameter_gamma_outputPrediction, parameter_m_outputPrediction,
parameter_threshold_outputPrediction):
    ...
    keyList = sorted(validModels.keys())
    deleteKeyList = []
    for x in range(0, len(keyList)):
        if(armStandsStill == True):
            if(x < (len(keyList) - 1)):
                deleteKeyList.append(keyList[x+1])
        else:
            if(keyList[x] < stepDirectionChanged):
                deleteKeyList.append(keyList[x])
            elif ((keyList[x] < stepCurrent) & (x < (len(keyList) - 1))):
                if(keyList[x+1] <= stepCurrent):
                    deleteKeyList.append(keyList[x])

    for x in range(0, len(deleteKeyList)):
        if(usingModel == deleteKeyList[x]):
            usingModel = -1

         del validModels[deleteKeyList[x]]
    ...
```

Listing 5.12: Löschen von Servermodellen

Die Verwaltung der Servermodelle findet nur für gültige Modelle statt. Um sicherzustellen, dass nur gültige Servermodelle verwaltet und damit benutzt werden, findet zuvor die Löschung ungültiger Servermodelle statt, die in *Listing 5.12* betrachtet wird.

Zunächst einmal werden alle vorhandenen Modelle, gemäß des Zeitschrittes ab dem diese gültig sind (Schlüssel) sortiert (Zeile 3). Anschließend wird über alle Schlüssel iteriert (Zeile 5). Hier erfolgt eine differenzierte Betrachtung, ob der Arm still steht oder nicht.

Zunächst für den Fall, dass der Arm still steht, werden alle Modelle, abgesehen vom ersten in der Modellliste in die Löschliste eingetragen (Zeile 6-8). Dies liegt daran, dass das zuletzt verwendete, also das erste Modell in der Liste, auch weiterhin ausgegeben werden soll und alle nachfolgenden für ungültig erklärt werden, da sich der Arm nicht mehr bewegt. Der Server weiß um diesen eventuellen Stillstand des Armes nicht Bescheid und berechnet weiterhin Outputergebnisse einer fließenden Armbewegung, da dieser für die Dauer des Stillstandes keine Clientupdates mehr erhält.

Für den Fall, dass der Arm nicht still steht, sich also bewegt (Zeile 9), wird jedes Modell, dass vor dem Zeitpunkt der letzten Richtungsänderung gültig ist, in die Löschliste eingetragen (Zeile 10-11). Dies wird nur im Falle der aktiven Löschstrategie betrieben, da ohne Löschen *stepDirectionChanged* immer 0 ist und folglich die Abfrage nie zu *wahr* evaluiert. Ansonsten werden alle Modelle in die Löschliste eingetragen, die vor dem aktuellen Zeitschritt gültig wurden und mindestens ein weiteres Modell existiert, dass nach diesem Modell gültig wurde, aber noch nicht in der Zukunft liegt, also das jetzige betrachtete damit ersetzt, da es verwendet werden darf (Zeile 12-14).

Somit ist das aktuelle und gültige zu verwendende Modell immer das erste in der *Dictionary*.

Die eigentliche Löschung all dieser veraltet identifizierten Schlüssel, findet im Anschluss statt (Zeile 16-20).

Falls unter den gelöschten Schlüsseln, jenes darunter war, dass noch bis zum letzten Iterationsschritt verwendet wurde, wird die Varibale *usingModel* = -1 gesetzt, das aussagt, dass gerade kein Modell verwendet wird. Diese wird in der nachfolgenden Verwaltung der Servermodelle relevant.

Verwalten von Servermodellen beim Client

Nachdem alle veralteten Modelle gelöscht wurden und somit die Modellliste nur noch gültige Modelle enthält, wird nachfolgend deren Verwaltung behandelt.

```
1  def inputReceive(numberParameters_inputPrediction, numberParameters_outputPrediction,
   parameter_gamma_outputPrediction, parameter_m_outputPrediction,
   parameter_threshold_outputPrediction):
2      ...
3      if(len(validModels) == 0):
4          usingModel = -1
5          remote_Solution = False
6
7      else:
8          firstKey = list(validModels.keys())[0]
9          firstModel = validModels[firstKey]
10
11         if(firstKey > stepCurrent):
12             usingModel = -1
13             remote_Solution = False
14
15         elif(firstKey <= stepCurrent):
16             if(usingModel == -1):
17                 for indices in firstModel[0]:
18                     valueList[indices] = firstModel[1].pop(0)
19                     trend_List[indices] = firstModel[2].pop(0)
20                     accelerationList[indices] = firstModel[3].pop(0)
21                     trendValues[indices] = 0
22
23                     for y in range(0, ((stepCurrent-firstKey))):
24                         dataListPrediction[indices] = extendedDampedHolts3(valueList[indices],
    trend_List[indices], accelerationList[indices], indices)
25                     dataListPredictionCount[indices] += 1
26
27                 for y in range(0, numberOutputParameters):
28                     dataListPrediction[y] = extendedDampedHolts3(valueList[y], trend_List[y],
   accelerationList[y], y)
29
30                 usingModel = firstKey
31                 remote_Solution = True
32
33             elif (usingModel == firstKey):
34                 if(armStandsStill == False):
35                     for x in range(0, numberOutputParameters):
36                         dataListPrediction[x] = extendedDampedHolts3(valueList[x], trend_List[
    x], accelerationList[x], x)
37
38                 usingModel = firstKey
```

```
39                  remote_Solution = True
40
41              else:
42                  print("Something went wrong.")
43                  usingModel = -1
44                  remote_Solution = False
45
46      outputProcessing(tracker_Simulation.latestInputValue)
```

Listing 5.13: Verwalten von Servermodellen

Die Verwaltung der Servermodelle wird in *Listing 5.13* betrachtet. Zunächst wird überprüft, ob es überhaupt ein gültiges Modell gibt (Zeile 3). Falls dies nicht der Fall ist, wird die Variable *remote_solution* = False gesetzt, sodass automatisch die Backup-Lösung verwendet wird.

Ansonsten wird zunächst danach unterschieden, ob das erste Modell bereits gültig ist oder noch gar nicht verwendet werden darf.

Falls das erste Modell noch nicht gültig ist, wird ebenfalls die Backup-Lösung verwendet (Zeile 11-13). Ansonsten, wenn das erste Modell gültig ist, wird überprüft, ob es bereits verwendet wird (Zeile 33) oder erst ab diesem Zeitpunkt verwendet wird (Zeile 16).

Sofern das Modell erst jetzt zum ersten Mal verwendet wird, werden die Modelldaten dieses Modells den Datenstrukturen zugewiesen (Zeile 17-21) und anschließend, um eventuell verspätete Updates zu behandeln, dieses neue Modell so oft vorhergesagt wie es der Differenz zwischen dem Zeitpunkt ab dem das Modell gültig ist und dem aktuellen Zeitschritt entspricht (Zeile 23-25). Zum Abschluss wird für alle Outputparameter der nächste Wert vorhergesagt. (Zeile 27-28) und die Verwendung dieses Modells gekennzeichnet (Zeile 30) und das Serverergebnis verwendet (Zeile 31).

Falls das Modell bereits in Gebrauch ist (Zeile 33), wird im Falle eines Armstillstands ohne die Outputparameterwerte für den nächsten Zeitschritt vorherzusagen, dasselbe Ergebnis wieder ausgegeben (Zeile 38-39). Falls der Arm dagegen nicht still steht, werden entsprechend vorher für alle Outputparameterwerte der nächste Zeitschritt vorhergesagt (Zeile 34-36), bevor das Servermodell verwendet wird.

Der Fall, dass das erste Modell gültig ist (Zeile 15) aber weder ein bisheriges Modell verwendet wird (Zeile 16), noch dem ersten Modell entspricht (Zeile 33), stellt einen Fehlerfall dar (Zeile 41), der niemals auftreten sollte und falls doch, dem Serverergebnis damit nicht zu trauen wäre. Deshalb würde in einem solchen Fall die Backup-Lösung verwendet werden (Zeile 44).

Zusammenfassung

Zusammengefasst lässt sich sagen, dass immer das Servermodell verwendet wird, sofern eines vorliegt. In den ersten $\max(m_0) + \max(m_1)$ Zeitschritten muss in jedem Fall die Backup-Lösung verwendet werden, da hier noch kein Servermodell vorhanden sein kann. Ansonsten dauert es bei aktiver Löschstrategie mindestens zwei Zeitschritte, ab dem Zeitpunkt der Löschung bis ein Servermodell wieder vorliegen kann, sodass währenddessen die Backup-Lösung verwendet werden muss.

Das Still stehen des Armes wird damit gewährleistet, dass das letzte benutzte Servermodell, ohne Vorhersage des nächsten Schrittes solange dasselbe Outputergebnis liefert bis der Arm nicht mehr still steht. In der Zeit werden keine Updates des Clients an den Server geschickt und umgekehrt sämtliche Updates des Servers verworfen.

Ein Vorteil der vom Client betriebenen Modellverwaltung ist das Vorhalten von Outputergebnissen des Servers, die bereits vorausberechnet wurden und fertig vorliegen. Auf diese Art ist es möglich, den Server schneller laufen zu lassen als den Client, der die Outputergebnisse im Voraus berechnet. Diese im Voraus berechneten Outputergebnisse wären zumindest solange gültig bis ein Armstillstand detektiert werden würde oder der Client ein neues Update an den Server schickt und dieses auf dessen Grundlagen ein neues Update an den Client zurückschickt. In beiden genannten Fällen, würden damit die auf Clientseite vorliegenden Modelle gelöscht werden. Im Falle der Einstellung *mit Löschen* würde auch jede Bewegungsrichtungsänderung ein sofortiges löschen auslösen. Falls all diese Fälle nicht auftreten würden, könnten die Outputergebnisse theoretisch bis ins Unendliche im Voraus vom Server berechnet werden und diese vom Client bereitgehalten werden, bis diese verwendet werden dürften.

6. Evaluation

In diesem Kapitel erfolgt die Auswertung der in *Kapitel 5 Implementierung* ausgeführten Implementierung. Hierzu wird im ersten Abschnitt die Konfiguration des Systems betrachtet. Anschließend wird auf die verwendeten Testdatensätze und die Fehlermaße der Auswertung eingegangen. Diesem schließen sich die empirischen Ergebnisse als Hauptteil an, in dem die eigentliche Auswertung statt findet. Danach erfolgt ein Abschnitt mit einer weiteren Auswertung mit Bandbreitenoptimierung der Outputparameter. Zum Abschluss des Kapitels findet im letzten Abschnitt eine Diskussion und Interpretation der Ergebnisse statt.

6.1. Evaluationssetup

In diesem Abschnitt wird die Konfiguration aufgezeigt mit dem die Evaluation vorgenommen wurde.

Der Server wurde in Python 3.9 geschrieben und mit tensorflow-cpu 2.10 für die Verwendung der NN auf einem privaten Laptop ausgeführt. Das gleiche gilt für den Client, der durch Pythonklassen simuliert wurde und sich in derselben Projektumgebung befindet, wie der Server.

Nachfolgend wird die genaue Konfiguration des oben erwähnten, privaten Laptops aufgeführt.
Prozessor: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
RAM: 16,0 GB (15,8 GB verwendbar)
Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
Betriebssystem: Windows 10 Home

6.2. Testdatensätze und Fehlermaße

In diesem Abschnitt wird auf die verwendeten Testdatensätze und die Fehlermaße eingegangen.

6.2.1. Testdatensätze

Es wurden zwei Testdatensätze verwendet, wobei das eine für die Bestimmung der Parameter und das andere für die eigentlichen Simulationen des verteilten Systems verwendet wurden und befinden sich beide in der Projektumgebung der Implementierung, die dieser Ausarbeitung beiliegt. Die beiden Testdatensätze heißen *mid_50.txt* und *mid_50_2.txt* und wurden mittels *Vicon* mit motion Tracking Punkten mit 100 FPS aufgenommen.

Der Testdatensatz *mid_50.txt* enthält die Inputwerte für 2.153 Iterationsschritte (d.h. eine 21,53 Sekunden lange Aufnahme) einer aufgezeichneten Armbewegung und im Falle von *mid_50_2.txt* die Inputwerte für 2.141 Iterationsschritte (d.h. eine 21,41 Sekunden lange Aufnahme) einer aufgezeichneten Armbewegung. Beide Testdatensätze enthalten zu jedem Iterationsschritt, den Winkel, die Geschwindigkeit, die Beschleunigung und den Gewichtsparameter (immer 0,0) als korrespondierende Inputwerte.

Der Testdatensatz *mid_50.txt* wurde für die Parameterbestimmungen von γ, m und *Schwellenwert* verwendet. Der Testdatensatz wird direkt im Falle der Parameterbestimmungen der Inputvorhersage gebraucht. Im Falle der Parameter der Outputvorhersage, wofür die sehr genauen indizierten (30 Punkte mit je 3 Koordinaten) Outputergebnisse als Referenz nötig sind, wurde auf Eingabe der Inputwerte von *mid_50.txt*, dessen produzierte sehr genaue indizierte (30 Punkte mit je 3 Koordinaten) Outputwerte als Vergleichswerte für die Parameterbestimmungen der Outputvorhersage verwendet.

Der Testdatensatz *mid_50_2.txt* wird für die eigentliche Simulation des verteilten Systems benötigt. Hiervon wurden die ersten 1.000 Iterationsschritte für die Simulation verwendet und die restlichen ignoriert. Diese werden für alle 12 Simulationsdurchgänge (6 verschiedene (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1) mal 2 verschiedene Implementierungseinstellungen (*mit Löschen*, *ohne Löschen*) verwendet, um die Ergebnisse miteinander vergleichen zu können.

6.2.2. Fehlermaße

Im Rahmen dieser Arbeit wurden zwei Fehlermaße verwendet. Zum einen handelt es sich um den, wie in *Unterabschnitt 5.2.1 Berechnung von γ und m* bereits erläuterten mmse, der als Fehlermaß für die Parameterbestimmungen von γ und m verwendet wird. Zum anderen wird der mean absolute error (mae) in der nachfolgenden Auswertung der Qualität der Simulationsergebnisse in *Unterabschnitt 6.4.4 Qualität* verwendet.

$$(6.1)\quad mae = \frac{1}{x} * \sum_{i=1}^{x} |(y_i - \hat{y}_i)|$$

Der mae aus *Formel 6.1* berechnet den absoluten Fehler zwischen den vorausgesagten Koordinatenwerten der indizierten Outputergebnisse und den tatsächlichen Koordinatenwerten der indizierten Outputergebnisse und bildet einen Durchschnitt über alle 90 Koordinatenwerte (30 Punkte mit je 3 Koordinaten). Dieser wird später den durchschnittlichen Fehler als Maßstab für die Qualität der Simulationsergebnisse für jeden Iterationsschritt darstellen. Aus diesen wird später wiederum der Durchschnitt gebildet, sodass dies den durchschnittlichen Gesamtfehler für die gesamte Simulation darstellt, um auf diese Weise, die verschiedenen Simulationsergebnisse miteinander vergleichen zu können.

6.3. Empirische Ergebnisse

Nachdem in *Abschnitt 6.1 Evaluationssetup*, die Systemkonfiguration betrachtet und in *Abschnitt 6.2 Testdatensätze und Fehlermaße*, die Frage beantwortet wurde, was und wie ausgewertet wird, erfolgen in diesem Abschnitt die Ergebnisse der Parameterbestimmungen und die eigentliche Auswertung der Simulationsergebnisse.

6.3.1. Parameter

Nachfolgend werden die Ergebnisse für die Bestimmung der Parameter γ, m und *Schwellenwert* für die Inputvorhersage ausgeführt. In *Tabelle 6.1* befinden sich die ermittelten Werte für die Parameter der Inputvorhersage.

Parameter	γ	m	*Schwellenwert*
Winkel	0,51	3	0,0202753125
Geschwindigkeit	0,01	4	0,138
Beschleunigung	0,01	2	1,88

Tabelle 6.1.: Parameterwerte der Inputvorhersage (4 Zeitschritte)

Das γ für den Winkelparameter ist mit 0,51 relativ hoch und drückt aus, dass die Veränderungen der Trends und die vorangegangenen Beschleunigungen in etwa gleichen Teilen für die Berechnung der aktuellen Beschleunigung berücksichtigt werden. Daraus lässt sich schließen, dass der Winkel der Armbewegung ziemlich gleichmäßig verändert wurde, also die Bewegung selbst sehr gleichmäßig verlief und man dadurch (anteilig) aus dem Trend heraus auf die Geschwindigkeit schließen kann. Sehr auffällig dagegen ist γ mit 0,01 für den Geschwindigkeits- und Beschleunigungsparameter. Damit wird ersichtlich, dass man aus dem Trend heraus nicht zuverlässig auf die Beschleunigung des Geschwindigkeits- und Beschleunigungsparameters schließen kann, da dieser sich nicht zuverlässig linear verändert. Man muss praktisch ausschließlich die vorangegangene Beschleunigung für die Ermittlung der aktuellen Beschleunigung heranziehen.

Alle Parameter verwenden eine relativ kurze Historie m von 2-4 vorangegangenen Inputwerten. Die vorab eingestellte Oberschranke von m=10 wird nicht einmal annähernd erreicht.

Die Schwellenwerte verhalten sich sehr logarithmisch zueinander. Beim Winkel erfordert bereits eine vergleichsweise kleine Änderung des 0,0202753125-fachen, dass das Vorhersagemodell neu gebildet werden muss. Andererseits drückt es auch aus, dass die Änderungen des Winkels vergleichsweise klein ausfallen. Bereits ab einem so kleinen *Schwellenwert* hält das Vorhersagemodell durchschnittlich 4 Zeitschritte lang. Anders sieht es dagegen bei den Schwellenwerten für den Geschwindigkeits- und Beschleunigungsparameter aus. Die Inputwerte des Geschwindigkeitsparameters rangieren in einem ähnlichen Bereich ± 200, wie der Winkel (ca. 10-140). Jedoch ist der *Schwellenwert* für den Geschwindigkeitsparameter ca. 10mal so hoch, da dieser sich sehr viel schneller ändert. Wie auch schon aus deren γ ersichtlich und bereits erklärt, können der Geschwindigkeits- und Beschleunigungsparameter praktisch nicht aus dem Trend heraus ermittelt werden, weil deren Änderungen im Verlauf zu groß sind. Die Änderung ist beispielsweise so groß, dass diese im Falle des Beschleunigungsparameters fast das doppelte des Beschleunigungsparameterwertes selbst ausmacht, damit die Vorhersagen im Durchschnitt immer noch 4 Zeitschritte lang halten.

Dieselben Ergebnisse für γ, m und *Schwellenwert* für die Outputvorhersage werden hier nicht tabellarisch aufgeführt, da es mit 90 Koordinatenwerten, zu viele Parameter zur Veranschaulichung wären. Diese können in den Dateien mit Namen *results_outputPrediction_gammaAndM.txt* und *results_outputPrediction_thresholds.txt* in der Projektumgebung der Implementierung, die dieser Arbeit beiliegt, genau eingesehen werden. Jedoch sollen diese nachfolgend kurz zusammengefasst werden. In *Tabelle 6.2* wird die Spannweite angegeben in denen sich die Parameter für γ, m und die *Schwellenwerte* befinden.

Parameter	γ	m	*Schwellenwert*
Outputwerte	0,01 - 0,22	2 - 6	0,00005294556081915217 - 0,0010986328125

Tabelle 6.2.: Spannweite der Parameterwerte der Outputvorhersage (4 Zeitschritte)

Die allermeisten γ sind 0,01 für die Parameter der Outputvorhersage und einige wenige reichen bis 0,22, was heißt, dass deren Veränderungen nicht zuverlässig aus dem Trend heraus ermittelt werden können, da sich diese nicht linear im Zeitverlauf ändern.

Die m sind ebenfalls relativ kurz mit Werten von 2-4 und in wenigen Fällen bis zu 6.

Der mit Abstand höchste *Schwellenwert* ist mit 0,0010986328125 immer noch sehr niedrig, was die hohe Sensitivität der Koordinatenwerte verdeutlicht. Die Koordinatenwerte/Outputparameter rangieren in einem sehr kleinen Intervall, weshalb der Schwellenwert so weit abgesenkt werden kann, sodass die Vorhersagen im Durchschnitt immer noch 4 Zeitschritte lang halten.

Da das verteilte System auf bestmögliche Qualität ausgerichtet ist, wurden die Parameterbestimmungen für den Vorhersagehorizont von 4 Zeitschritten optimiert. Für den Fall alternativer Ansätze, sind in *Abschnitt A.1 Parameterbestimmungen für alternative Vorhersagehorizonte* die Parameterbestimmungen für die Vorhersagehorizonte 5, 10, 20 und 30 Zeitschritte aufgeführt.

Allgemein kann zu den verschiedenen Vorhersagehorizonten angemerkt werden, dass längere Vorhersagehorizonte zu größeren Schwellenwerten führen, wobei sensitive Parameter weniger stark ansteigen als nicht-sensitive.

Nachfolgend erfolgt die Auswertung der Simulationsergebnisse des verteilten Systems. Die Auswertungen beziehen sich immer auf die Implementierungseinstellung *ohne Löschen*. Für die Ergebnisse der Implementierungseinstellung *mit Löschen* wird auf den Anhang *Abschnitt A.2 Implementierungseinstellung* mit Löschen verwiesen.

6.3.2. Kommunikation

In *Abbildung 6.1* wird die Kommunikation der Simulationsergebnisse veranschaulicht.

Auf der x-Achse befinden sich die sechs verschiedenen (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1). Die y-Achse enthält die Anzahl der Sendevorgänge für jedes (Input, Output)-Updatestrategiepaar, das sich im Laufe der jeweiligen Simulation ergeben hat, getrennt nach den Inputvorhersagen in Blau (Client zu Server) und Outputvorhersagen in Orange (Server zu Client).

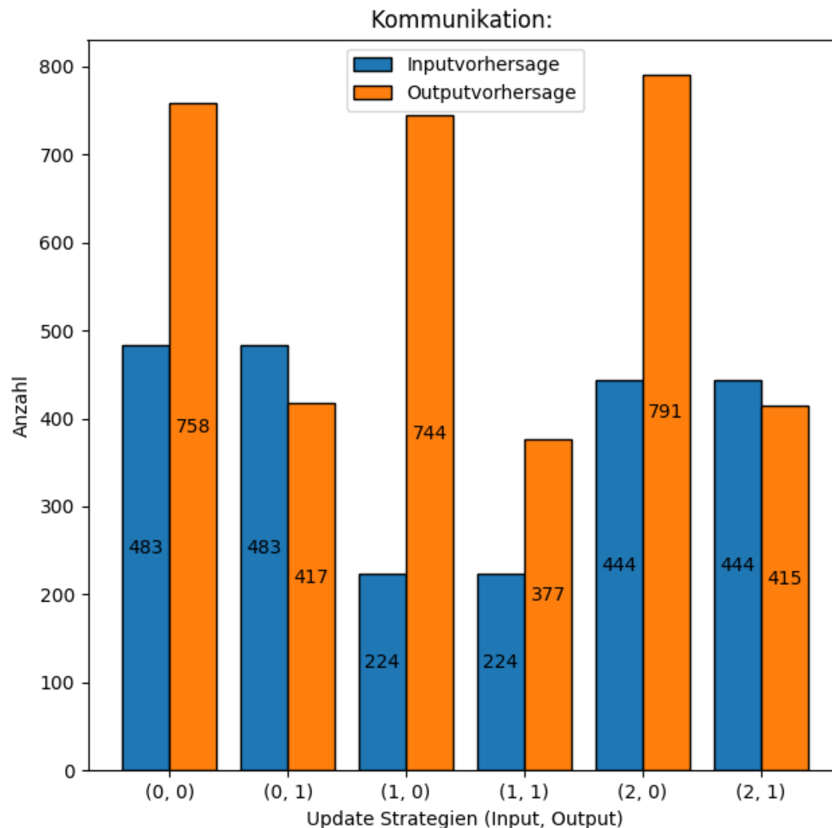

Abbildung 6.1.: Kommunikation (ohne Löschen)

Für die Inputvorhersagen fanden im Rahmen der *Individual*-Updatestrategien 483 Kommunikationsvorgänge und im Rahmen der *All*-Updatestrategien 444 Kommunikationsvorgänge statt. Die *Master*-Updatestrategien weißen mit 224 Kommunikationsvorgängen einen deutlich geringeren Wert auf, der sich jedoch wie erwartet verhält, da ein Sendevorgang bei dieser Updatestrategie nur anfällt, falls für den Masterparameter *Winkel* ein neues Vorhersagemodell gebildet werden muss.

Bei einer Betrachtung der Outputvorhersagen fällt auf, dass die Anzahl der Kommunikationsvorgänge für die *Individual*-Updatestrategien immer etwas unter 800 Sendevorgänge enthält und für die *All*-Updatestrategien um die 400 Sendevorgänge. Dieser Unterschied zwischen den beiden Updatestrategien lässt sich damit erklären, dass bei der *Individual*-Updatestrategie ein Sendevorgang anfällt, falls für irgendeinen oder mehrere Parameter das Vorhersagemodell neu erstellt werden muss. Die anderen Parameter werden dabei nicht mitgeschickt, sodass deren Vorhersagemodelle zu einem späteren Zeitpunkt nicht mehr halten und diese wieder individuell verschickt werden. Dadurch steigt die Gesamtanzahl der Sendevorgänge im Vergleich zur *All*-Updatestrategie an.

Dieser Unterschied zwischen der *Individual*- und *All*-Updatestrategie ist für die Outputvorhersage sehr viel größer, da es 30 mal mehr Outputparameter als Inputparameter gibt, weshalb die Inputvorhersagen mit 483 Sendevorgängen für die *Individual*- und 444 Sendevorgängen für die *All*-Updatestrategien, eine deutlich geringere Differenz zwischen diesen beiden Updatestrategien aufweist.

Von Interesse ist auch der Zusammenhang zwischen der Anzahl der Inputvorhersagen und deren jeweiligen Outputvorhersagen. Unabhängig von der Updatestrategie der Inputvorhersage, weist die darauf folgende Outputvorhersage annähernd dieselbe Anzahl an Sendevorgängen auf bzw. ist sogar mit 744 bzw. 377 Sendevorgängen bei der *Master*-Updatestrategie der Inputvorhersage am geringsten. Hieraus wird der maßgebliche Einfluss des Parameters *Winkel* auf die NN verdeutlicht.

Die entsprechende Veranschaulichung der Kommunikation *mit Löschen* kann in *Abbildung A.1* eingesehen werden.

6.3.3. Vorhersagedauer

Nachdem die Kommunikation betrachtet wurde, werden nachfolgend die Vorhersagedauern der Simulationen ausgewertet. In *Abbildung 6.2* werden die durchschnittlichen Vorhersagedauern der Simulationsergebnisse veranschaulicht.

Die x-Achse zeigt die Anzahl der Iterationsschritte, die die Vorhersagemodelle im Durchschnitt halten. Die y-Achse enthält die sechs verschiedenen (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1). Diese ist für jedes (Input, Output)-Updatestrategiepaar weiter aufgeteilt in durchschnittlich gesendete Inputvorhersagen in Blau, durchschnittlich totale Inputvorhersagen in Orange, durchschnittlich gesendete und totale Outputvorhersagen in Grün, sowie durchschnittlich bereinigte totale Outputvorhersagen in Rot. Da es 3 verschiedene Inputparameter und 90 verschiedene Outputparameter gibt, wurde ein gemeinsamer Durchschnittswert für die Inputparameter und ein gemeinsamer Durchschnittswert für die Outputparameter gebildet.

Die gesendeten und totalen Durchschnittswerte, unterscheiden sich danach, wann diese gemessen wurden. Gesendet entspricht der durchschnittlichen Anzahl der Vorhersagemodelle, die tatsächlich gesendet wurden, also bei der tatsächlichen Verschickung gezählt wurden. Im Falle der totalen durchschnittlichen Anzahl, wird die Gesamtanzahl der Bildungen der Vorhersagemodelle betrachtet, unabhängig davon, ob diese auch verschickt wurden.

Für die gesendeten und totalen Durchschnittswerte ergeben sich bei der Inputvorhersage noch zwei weitere Einflussfaktoren, die es bei der Outputvorhersage nicht gibt. Die erste Möglichkeit betrifft nur den Fall der *Master*-Updatestrategie, die es nur beim Client gibt, weshalb dieser Fall nicht beim Server und damit nicht der Outputvorhersage auftreten kann. Bei der *Master*-Updatestrategie wird nur an den Server gesendet, falls für den Masterparameter *Winkel* ein neues Vorhersagemodell gebildet wurde, sodass dann die Vorhersagemodelle zu allen Inputparametern geschickt werden. D.h. falls auch nur für einen der beiden Inputparameter ein neues Vorhersagemodell erstellt wurde, aber nicht für den Winkel, wird nichts an den Server geschickt, was zu verschiedenen Durchschnittswerten führt. Damit erklärt sich auch, warum der Unterschied zwischen den gesendeten (3,35) und totalen (4,45) Durchschnittswerten für die *Master*-Updatestrategien mit Abstand am größten ist.

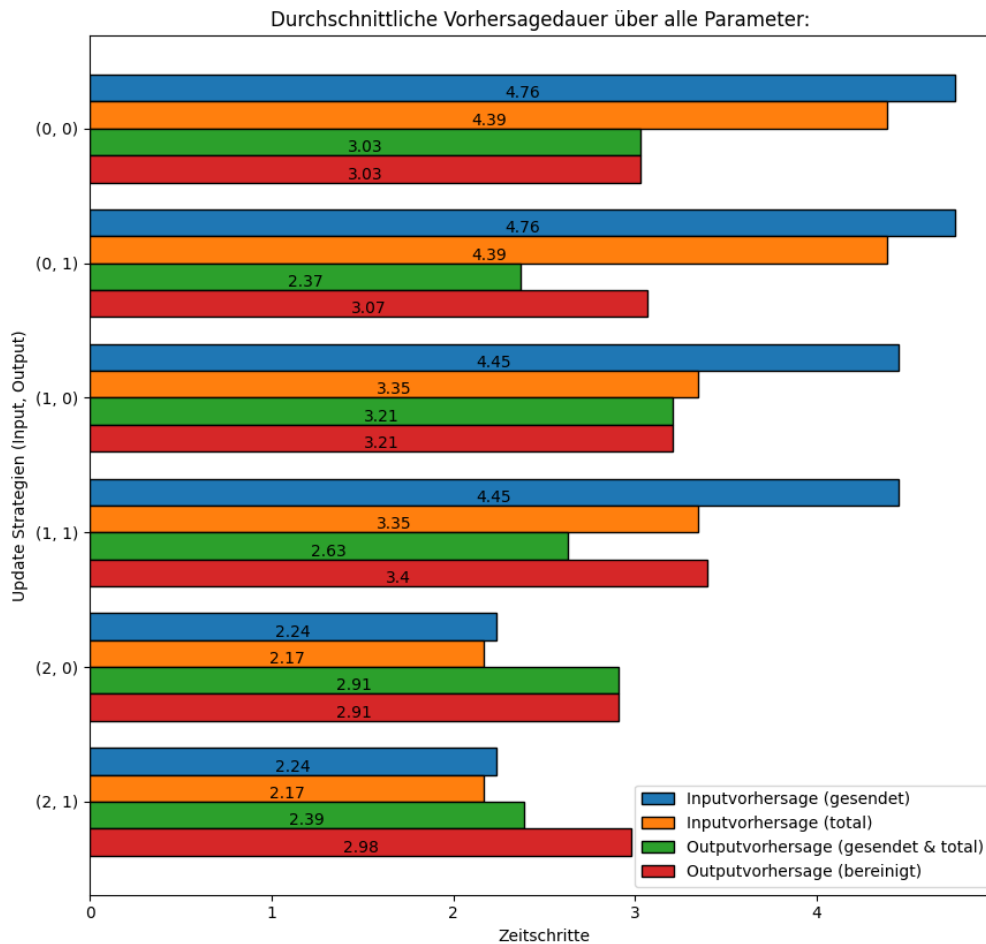

Abbildung 6.2.: Vorhersagedauern (ohne Löschen)

Die zweite Möglichkeit betrifft alle Updatestrategien der Inputvorhersage, nämlich wenn der Arm still steht. Wenn der Arm still steht, werden die Vorhersagemodelle weiterhin gebildet, jedoch für die gesamte Dauer des Stillstandes nicht an den Server gesendet. Deshalb ist der Unterschied zwischen den gesendeten (2,24) und totalen (2,17) Durchschnittswerten für die *All*-Updatestrategien der Inputvorhersage mit Abstand am kleinsten, weil sich diese Differenz, wie auch bei den *Individual*-Updatestrategien, ausschließlich aus dem Stillstand des Armes begründet, aber im Gegensatz zur *Individual*-Updatestrategie durch sehr viel mehr Modellbildungen dieses Verhältnis kleiner ist. Der Umstand des Stillstandes des Armes hat keinerlei Auswirkungen auf den Server, da dieser in jedem Fall für den nächsten Zeitschritt das Ergebnis berechnet und schickt, womit sich für gesendete und totale Messungen, identische Durchschnittswerte für die Outputvorhersage ergeben. Deshalb wurde für die Outputvorhersage noch der bereinigte Durchschnittswert gebildet, um die Vorhersagedauer der Vorhersagemodelle, unabhängig von der Verzerrung, die sich durch die *All*-Updatestrategie der Outputvorhersage ergibt, unverfälscht ermitteln zu können. Entsprechend geben der bereinigte,

sowie die gesendeten und totalen Durchschnittswerte im Falle der *Individual*-Updatestrategien der Outputvorhersage identische Werte aus. Die Abweichungen der bereingten, sowie der gesendeten und totalen Durchschnittswerte ist für die *All*-Updatestrategien der Outputvorhersage sehr viel größer als für die Inputvorhersage. Dies erklärt sich dadurch, dass es 30mal mehr Outputparameter gibt als Inputparameter, wodurch diese Durchschnittswerte für bereinigt zum einen und gesendet und total zum anderen für die Outputvorhersage stärker abweichen.

Der unverfälschte Wert an dem die tatsächliche durchschnittliche Vorhersagedauer der Vorhersagemodelle für die Inputvorhersage bestimmt wird, kann anhand der totalen Durchschnittswerte der *Individual*-Updatestrategie der Inputvorhersage (4,39) abgelesen werden. Dies ist der unverfälschte Wert, wie lange die Vorhersagemodelle über alle Inputparameter hinweg durchschnittlich halten, ohne dass die Inputparameter sich gegenseitig oder das Stillstehen des Armes diesen beeinflussen.

Derselbige unverfälschte Wert, durch den die tatsächliche Vorhersagedauer der Vorhersagemodelle für die Outputvorhersage in etwa bestimmt werden kann, sind die gesendeten und totalen Durchschnittswerte der *Individual*-Updatestrategie der Outputvorhersage (3,03; 3,21; 2,91). Hier ergeben sich drei verschiedene Werte, weil die Outputvorhersagen kausal von den eingegebenen Inputparametern und damit von den drei verschiedenen Updatestrategien der Inputvorhersage abhängen. Deshalb kann dieser tatsächliche, unverfälschte Wert für die Outputvorhersage nur in etwa angegeben werden bzw. als Tendenz, die ausgemacht wird.

In jedem Fall halten alle Input- und Outputvorhersagemodelle für sich genommen im Durchschnitt über 3 Zeitschritte, abgesehen von den Updatestrategien der Outputvorhersage, die gerade so im Durchschnitt 3 Zeitschritte lang halten. Das diese in anderen Fällen abweichen können, ist nicht der Qualität der Vorhersagemodelle selbst geschuldet, sondern den jeweiligen Updatestrategien, dass diese auch neu gebildet und verschickt werden, obwohl die Vorhersagemodelle noch halten würden.

Die entsprechende Veranschaulichung der Vorhersagedauern *mit Löschen* kann in *Abbildung A.2* eingesehen werden.

6.3.4. Bandbreite

Nachfolgend wird die benötigte Bandbreite für die durchgeführten Simulationen ausgewertet. In *Abbildung 6.3* werden die durchschnittlichen Vorhersagedauern der Simulationsergebnisse veranschaulicht.

Gesamtbetrachtung aller Updatestrategien

Auf der x-Achse sind die sechs verschiedenen (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1) und zum Vergleich, die benötigte Bandbreite bei 100% offloading als Säulendiagramme aufgetragen. An der y-Achse kann in Bytes abgelesen werden, wieviel Bandbreite die einzelnen (Input, Output)-Updatestrategiepaare benötigen, differenziert nach Inputvorhersagen in Orange, Outputvorhersagen in Rot und den kumulierten Gesamtbedarfen über den Säulendiagrammen. Die benötigte Bandbreite für die Übertragung des Inputs bei 100% offloading ist in Blau und die Übertragung des Outputs bei 100% offloading ist in hellblau dargestellt.

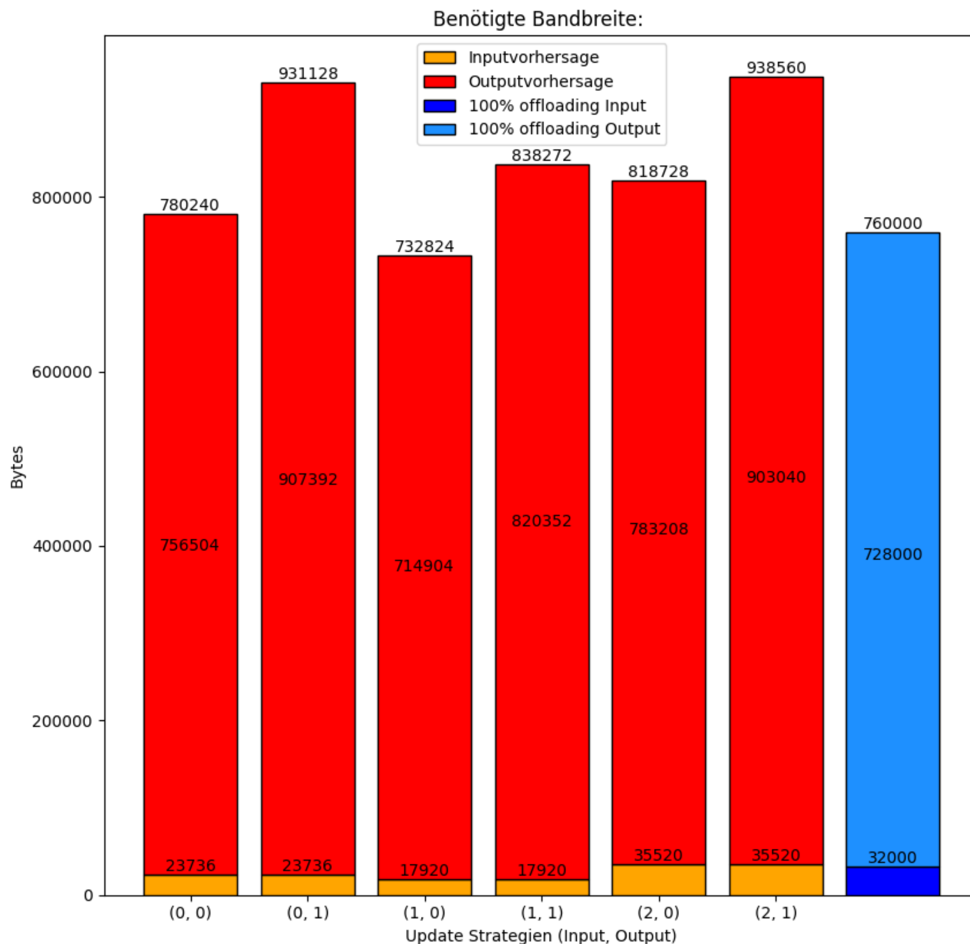

Abbildung 6.3.: Bandbreite (ohne Löschen)

Die Berechnungen der aufgeführten Bandbreiten für die Updatestrategien erfolgte so, dass jedes gesendete Vorhersagemodell 24 Bytes benötigt. Die Vorhersagemodelle, sowohl für die Input- als auch für die Outputvorhersagen setzen sich aus 3 Modelldaten (Stützwert, Trend und Beschleunigung) zusammen, welche als Floats mit 8 Bytes kodiert werden. Zu jeder Kommunikation im Rahmen der Inputvorhersage, schickt der Client neben den zu sendenden Vorhersagemodellen noch als Integer, der mit angenommenen 8 Bytes kodiert wird, den aktuellen Zeitschritt als Synchronisationsvariable, der als fixer Anteil eines jeden Sendevorgangs zur Inputvorhersage immer mitanfällt. Die Sendevorgänge zur Outputkommunikation enthalten neben den Vorhersagemodellen, den aktuellen Zeitschritt und den Zeitschritt, auf Grundlage dessen die Outputvorhersagemodelle berechnet wurden, ebenfalls beide als Integer mit je angenommenen 8 Bytes kodiert. Im Falle der *Individual*-Updatestrategien werden zu den zu sendenden Vorhersagemodellen entweder eine Indizesliste mitgeschickt, die die

Parameter enthalten, auf die sich die Vorhersagemodelle beziehen, wobei jeder Index ein Integer ist, der mit angenommenen 8 Bytes kodiert wird oder es werden alle Vorhersagemodelle geschickt, womit die Indizesliste überflüssig wird.

$$Input = 8Bytes \cdot (|Inputparameter| + 1) \cdot (|Simulationsschritte| - |Armstehtstill|) \tag{6.2}$$

$$Output = 8Bytes \cdot (|Outputparameter| + 1) \cdot |Simulationsschritte| \tag{6.3}$$

Die Berechnung der Inputbandbreite für den Fall eines 100% offloadings kann wie in *Formel 6.2* berechnet werden. Für die dadurch benötigte Inputbandbreite gilt, dass in jedem Zeitschritt alle Inputparameter und dazu der aktuelle Zeitschritt, die alle mit 8 Bytes kodiert werden, an den Server gesendet werden. Dies gilt für alle Simulationsschritte abzüglich jenen, in denen der Arm still steht, wo folglich nichts gesendet wird.

Für die Berechnung der Bandbreiten im Falle eines 100% offloadings des Outputs erfolgt die Berechnung wie in *Formel 6.3*. In jedem Zeitschritt werden alle Outputparameter und der Zeitschritt an den Client zurückgesendet. Dies geschieht für jeden Zeitschritt der Simulation.

Die *Master*-Updatestrategien benötigen mit 17.920 Bytes unter allen Updatestrategien der Inputvorhersage am wenigsten Bandbreite. Dies liegt daran, dass die Sendevorgänge hier nur anfallen, falls für den Masterparameter *Winkel* ein neues Vorhersagemodell gebildet werden muss, wofür dann in der Folge die Vorhersagemodelle für alle Inputparameter an den Server geschickt werden.

Mit 23.736 Bytes benötigen die *Individual*-Updatestrategien der Inputvorhersage etwas mehr Bandbreite. Im Rahmen dieser Updatestrategie werden zwar nur die Vorhersagemodelle geschickt, die tatsächlich neu gebildet wurden, jedoch ist die dafür benötigte Bandbreite leicht höher, da die anderen Inputparameter öfter neu gebildet werden und damit verschickt werden müssen als der zuvor in der *Master*-Updatestrategie, wo der Masterparameter *Winkel* seltener einen Sendevorgang auslöst.

Schließlich weisen die *All*-Updatestrategien der Inputvorhersage mit 35.520 Bytes mit Abstand die größten benötigten Bandbreiten auf. Die Sendevorgänge werden hier durch jeden beliebigen Parameter im Gegensatz zum Masterparameter *Winkel* der *Master*-Updatestrategie verursacht. Jedoch werden beim Auslösen, genauso wie bei der *Master*-Updatestrategie die Vorhersagemodelle zu allen Parametern geschickt.

Die Outputvorhersage benötigt den Hauptteil der Gesamtbandbreite, was jedoch nicht verwundern darf, da es 30mal mehr Outputparameter als Inputparameter gibt. Im Rahmen der Outputvorhersage gibt es nur die beiden Updatestrategien *Individual* und *All*. Die *Individual*-Updatestrategien brauchen mit (756.504; 714.904; 783.208) Bytes weniger Bandbreite als die *All*-Updatestrategien mit (907.392; 820.352; 903.040) Bytes.

Die summierten Ergebnisse von benötigten Input- und Outputbandbreiten betragen (780.240; 931.128; 732.824; 838.272; 818.728; 938.560) Bytes.

Nachfolgend werden in *Tabelle 6.3* die benötigten Bandbreiten der Updatestrategien im Vergleich zum 100% offloading angegeben.

Updatestrategie	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
100% Offloading	102,66%	122,51%	96,42%	110,29%	107,72%	123,49%

Tabelle 6.3.: Bandbreitenbedarf der Updatestrategien im Verhältnis zum 100% offloading

Abgesehen von Updatestrategie (1, 0), welches 96,42% der Bandbreite des 100% offloadings braucht, benötigen sämtliche Updatestrategien mehr Bandbreite als das 100% offloading. Dies darf nicht verwundern, da das verteilte System auf bestmögliche Qualität ausgerichtet ist. Die Schwellenwerte wurden so optimiert, dass der Ansatz die Modelldaten statt den Inputwerten zu schicken, sich gerade noch so lohnt. Falls jedoch der Schwellenwert erhöht werden würde, was zwar die Qualität reduziert, aber umgekehrt die durchschnittlichen Vorhersagedauern erhöhen und in der Konsequenz die Kommunikation zwischen Client und Server verringert, würde entsprechend den Bandbreitenbedarf reduzieren. Damit würde der Vorteil des Ansatzes die Modelldaten zu schicken, sich nicht nur für Updatestrategie (1, 0) erfüllen, sondern in jedem Fall auch mit größeren Schwellenwerten auch deutlicher auffallen. Obwohl die Bandbreitenbedarfe der anderen Updatetrategien über denen des 100% offloading liegen, sind die benötigten Bandbreiten mit 102,66-123,49% trotzdem in einem vergleichbaren Bereich mit jenen des 100% offloadings.

Vergleich zwischen den Updatestrategien

Die *Individual*-Updatestrategien der Outputvorhersage benötigen im Verhältnis zu den *All*-Updatestrategien der Outputvorhersage auffällig mehr Bandbreite. Im Falle der Inputvorhersagen beträgt dieses Verhältnis in etwa $\frac{2}{3}$, wohingegen es für die Outputvorhersagen in etwa 83-87% entspricht. Dies lässt sich damit erklären, dass im Falle der Outputvorhersage die *Individual*-Updatestrategien deutlich öfter alle Vorhersagemodelle schicken als nur jene, für die tatsächlich ein neues Vorhersagemodell erstellt wurde mit einer dazugehörigen Indizesliste. Im Falle der Outputvorhersage werden ab 68 Outputparametern, für die ein neues Outputvorhersagemodell gebildet wurden, alle Vorhersagemodelle an den Client geschickt, da ((4 * Länge der Indizesliste) ≥ (3 * Anzahl aller Parameter)) ist. Gemäß derselbigen Formel sendet der Client im Rahmen der *Individual*-Updatestrategien immer eine Indizesliste, sofern sich weniger als alle (3) Inputparameter geändert haben. Das heißt, weil es 30mal mehr Outputparameter als Inputparameter gibt und die Grenze der geänderten Outputparameter, ab der alle Outputparameter geschickt werden lediglich bei 68 liegt, werden bei der Outputvorhersage im Verhältnis deutlich mehr Vorhersagemodelle mitgeschickt, wodurch mehr Bandbreite benötigt wird. Dies stellt sogar noch eine Ersparnis dar, da ein Verschicken von Indizeslisten ab 68 Outputvorhersagemodellen, sogar noch mehr Bandbreite benötigen lassen würde.

Synchronisationsvariablen

Die Synchronisationsvariable *Zeitschritt* und *Nummer des Inputupdates* machen einen sehr unterschiedlichen Anteil an der benötigten Bandbreite auf. Durch die Anzahl der Sendevorgänge aus *Abbildung 6.1* kann die benötigte Bandbreite für die Synchronisationsvariablen ermittelt werden. Diese können im Falle der Bandbreiten für die Inputvorhersage durch die Anzahl der Sendevorgänge der Inputvorhersage mal 8 Bytes für den Zeitschritt berechnet werden. Im Falle

der Outputvorhersagen muss die Anzahl der Sendevorgänge mit 16 Bytes multipliziert werden, da neben dem *Zeitschritt* noch die *Nummer des Inputupdates* zu jedem Sendevorgang fix mitgeschickt wird.

Daraus ergeben sich die in *Tabelle 6.4* aufgeführten Bandbreiten in Bytes.

Updatestrategie	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
Inputvorhersage	3.864	3.864	1.792	1.792	3.552	3.552
Outputvorhersage	12.128	6.672	11.904	6.032	12.656	6.640

Tabelle 6.4.: Benötigte Bandbreite für die Synchronisationsvariablen (Absolut)

Die Bandbreiten für die Synchronisationsvariablen der Outputvorhersage benötigen fast das Doppelte im Falle der Updatestrategie (0, 1) und sogar mehr als das 6-fache im Falle der Updatestrategie (1, 0). Wenn man jedoch das Verhältnis der benötigten Bandbreiten für die Synchronisationsvariablen, gemessen an den Bandbreiten für die Input- bzw. Outputvorhersagen betrachtet, kehren sich diese Verhältnisse mehr als um. Daraus ergeben sich die in *Tabelle 6.5* aufgeführten Verhältnisse.

Updatestrategie	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
Inputvorhersage	16,27%	16,27%	10,00%	10,00%	10,00%	10,00%
Outputvorhersage	1,60%	0,73%	1,66%	0,73%	1,61%	0,73%

Tabelle 6.5.: Benötigte Bandbreite für die Synchronisationsvariablen (Verhältnis)

Die Synchronisationsvariablen für die Outputvorhersage machen mit 0,73 - 1,66% einen ca. 10mal geringeren Anteil an der Bandbreite der Outputvorhersage als die Synchronisationsvariable der Inputvorhersage mit 10,00 - 16,27% an der Inputvorhersage aus. Dieser Umstand lässt sich dadurch erklären, dass es 30mal mehr Outputparameter als Inputparameter gibt, weshalb dieser fixe Anteil der Synchronisationsvariablen einen weniger großen Anteil bei der Outputvorhersage ausmacht. Allerdings ist dieses Verhältnis auch nicht 30mal kleiner, sondern nur 10mal kleiner bzw. 5 kleiner, wenn man berücksichtigt, dass die Outputvorhersage zwei Synchronisationsvariablen verwendet, was durch eine durchschnittlich häufigere Verschickung von Vorhersagemodellen erklärt werden kann, wie es auch zuvor in *Unterabschnitt 6.4.2 Vorhersagedauer* beobachtet wurde.

Die Bandbreiten, die sich *mit Löschen* ergeben, können in *Abbildung A.3* eingesehen werden.

6.3.5. Qualität

Den Abschluss der empirischen Ergebnisse macht die Auswertung der Qualität aus. Die Qualität wurde im Vergleich zum High Quality NN gemessen. Die Outputwerte wurden vor der Simulation durch Eingabe der Inputwerte der tatsächlichen aufgezeichnten Armbewegung in das High Quality NN generiert. Der Vergleich der nachfolgenden Outputergebnisse der Simulation erfolgt anhand dieser.

Gesamtbetrachtung aller Updatestrategien

Nachfolgend werden die Qualitätsunterschiede zwischen den verschiedenen Updatestrategien behandelt. In *Abbildung 6.4* wird die Gesamtqualität der Simulationsergebnisse veranschaulicht.

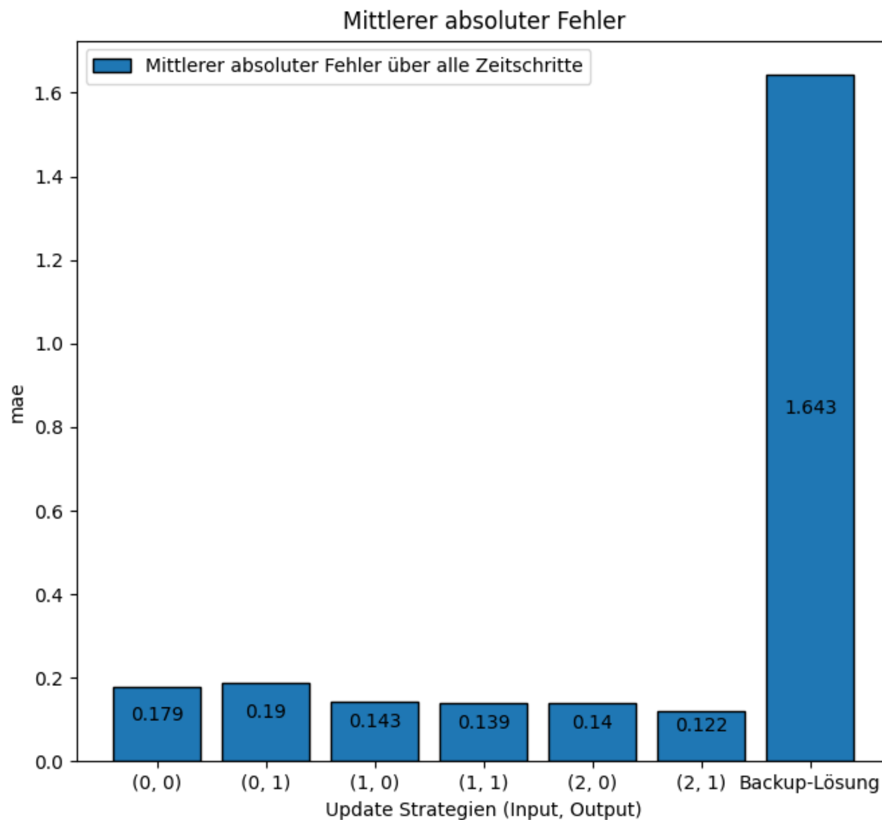

Abbildung 6.4.: Gesamtqualität (ohne Löschen)

Auf der x-Achse sind die sechs verschiedenen (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1) und zum Vergleich dieser, die Backup-Lösung als Säulendiagramme aufgetragen. Die y-Achse weist den mae für jeden der sechs verschiedenen (Input, Output)-Updatestrategiepaare und der Backup-Lösung auf.

Die Updatestrategien (0, 0) und (0, 1) weisen unter den sechs verschiedenen Updatestrategien mit einem Wert von 0,179 bzw. 0,19 die größten mae auf.

Sehr interessant ist, dass die Updatestrategie (1, 0) wiederum einen erkennbar geringeren mae mit einem Wert von 0,143 aufweist als die beiden zuvor. Hieran kann der große Einfluss des Masterparameters *Winkel* auf das Gesamtergebnis erkannt werden. Updatestrategie (1, 0) unterscheidet sich von Updatestrategie (0, 0) nur darin, dass der Auslöser von Sendevorgängen für die Inputvorhersage einzig und alleine der *Winkel* ist und dazu die anderen beiden Inputparameter mitgeschickt werden. Obwohl in der Updatestrategie (1, 0) die Outputvorhersage individuell erfolgt und nicht alle

Outputparameter geschickt werden, wie in der Updatestrategie (0, 1), in der bei jeder beliebigen Erstellung eines Outputparameters, neue Vorhersagemodelle zu allen anderen Outputparametern mitgeschickt werden, ist der mae geringer, obwohl der mae an den Outputparametern gemessen wird.

Noch interessanter wird es bei einer Betrachtung der Updatestrategie (1, 1), der mit einem mae von 0,139 fast denselben Wert wie Updatestrategie (1, 0) aufweist. In dieser ist ebenfalls der *Winkel* der auslösende Sendevorgang der Inputvorhersage, jedoch führt jede Neuerstellung eines Outputparameters zur Neuerstellung aller anderen Outputparameter, die immer mitgeschickt werden. Das diese unterschiedlichen Updatestrategien für die Outputvorhersage, bei ansonsten identischen Updatestrategien für die Inputvorhersage, nicht zu einer deutlicheren Veränderung führt, zeigt sogar noch eindrücklicher die Relevanz des *Winkels* auf das Gesamtergebnis auf.

Eine leichte Verringerung des mae auf 0,14 weist die Updatestrategie (2, 0) auf. In diesen lösen neben dem *Winkel* als Masterparameter der Updatestrategien (1, 0) und (1, 1) auch die anderen beiden Inputparameter Sendevorgänge bei der Inputvorhersage aus. Dieser Einfluss der anderen beiden Inputparameter ist zwar da, wie am geringeren mae im Vergleich zu Updatestrategie (1, 0) erkannt werden kann, jedoch so gering, dass diese fast nicht auffallen.

Für eine nochmals erkennbare Verringerung des mae sorgt die Updatestrategie (2, 1) mit einem Wert von 0,122. Hier sorgen im Gegensatz zur Updatestrategie (2, 0) neben dem Auslösen von allen Inputparametern von Sendevorgängen für die Inputvorhersage, auch alle Outputparameter für das Auslösen von Sendevorgängen für die Outputvorhersage.

Eine rein lokale Ausführung der Simulation mithilfe des auf dem Client laufenden Backup NN sorgt für einen sehr großen mae von 1,643.

Daraus ergeben sich die in *Tabelle 6.6* aufgeführten Qualitätsverhältnisse im Vergleich zur Backup-Lösung, d.h. um wieviel der mae, den die Backup-Lösung erzeugt, mit Verwendung der Updatestrategien im Verhältnis dazu reduziert werden kann.

Updatestrategie	(0, 0)	(0, 1)	(1, 0)	(1, 1)	(2, 0)	(2, 1)
Inputvorhersage	89,10%	88,43%	91,29%	91,53%	91,47%	92,57%

Tabelle 6.6.: Fehlerreduzierung der Updatestrategien im Verhältnis zur Backup-Lösung

Sämtliche Updatestrategien sorgen für eine deutliche Verbesserung der Qualität und sprechen damit für das verteilte System, statt die Berechnungen rein lokal auf dem Client laufen zu lassen. Unter den dafür untersuchten Updatestrategien weist die *Individual*-Updatestrategie der Inputvorhersage den größten mae auf, der jedoch immer noch sehr viel geringer ist als eine rein lokale Ausführung über das Backup NN auf dem Client. Die Verwendung der *Master*-Updatestrategie der Inputvorhersage sorgt für eine erkennbare Verbesserung, während die *All*-Updatestrategie für die größte Verbesserung sorgt. Im Falle der Outputvorhersage stellt die *All*-Updatestrategie in allen Fällen, außer der *Individual*-Updatestrategie der Inputvorhersage, eine Verbesserung dar.

Die Qualität, die sich *mit Löschen* ergibt, kann in *Abbildung A.4* eingesehen werden.

Die Qualität der Updatestrategien *mit Löschen* sind deutlich schlechter als jene die *ohne Löschen* durchgelaufen sind. Die Erklärung dahinter ist, dass sobald eine Bewegungsrichtungsänderung erfolgt ist, werden sämtliche Servermodelle beim Client gelöscht und zusätzlich werden alle

Servermodelle verworfen, die nicht auf Grundlage der neuen Armbewegung berechnet wurden. D.h. damit ein Servermodell vom Client akzeptiert wird, muss zunächst der Client ein Update an den Server schicken, sodass dieser auf Grundlage der neuen Armbewegung die Outputwerte berechnet. Anschließend muss der Server, der nun auf Grundlage dieser neuen Armbewegung Outputergebnisse produziert, ein Update an den Client schicken, nachdem es ein Update vom Client, nach dem Zeitschritt der Bewegungsrichtungsänderung erhalten hat.

Das Problem das bei *mit Löschen* entsteht ist, dass oftmals entweder nicht direkt nach der Bewegungsrichtungsänderung ein Update des Clients an den Server erfolgt oder der Client schickt zwar relativ zeitnah ein Update an den Server, dieser schickt jedoch lange Zeit kein Update an den Client zurück, da für keinen Outputparameter ein neues Vorhersagemodell erstellt wurde. In all dieser Zeit, in der es kein Servermodell beim Client gibt und bis der Server endlich ein Update an den Client schickt, dass auf Grundlage der neuen Armbewegung berechnet wurde, also der Server selbst zuvor ein Update des Clients erhalten hat, wird das Backup NN verwendet, dass wie zuvor schon festgestellt deutlich schlechtere Outputergebnisse erzeugt.

Dieser Umstand, dass lange Zeit (teilweise mehrere Dutzend Zeitschritte) kein Update des Clients an den Server oder vom Server zurück an den Client geschickt wird, entsteht, weil die Vorhersagemodelle so gut funktionieren. Die bloße Änderung der Bewegungsrichtung selbst löst kein Update aus. Diese wird zumeist, wie erwartet vom Vorhersagemodell erkannt. Als Beispiel seien die folgenden Werte: $25 \rightarrow 15 \rightarrow 5$, die die letzten Inputwerte für die Geschwindigkeit darstellen gegeben. Das Vorhersagemodell wird mit großer Wahrscheinlichkeit als nächsten Vorhersagewert eine Geschwindigkeit im Bereich von ca. -5 vorhersagen und sobald dieser Wert (bzw. ein naher Wert, der innerhalb des Schwellenwertes von diesem abweicht) als nächstes eintritt, diesen richtig vorhersagen. Die Bewegungsrichtung ändert sich, alle Servermodelle werden gelöscht, alle in Zukunft ankommenden Servermodelle, die auf der vorigen Armbewegung beruhen werden verworfen, und es wird kein neues Vorhersagemodell gebildet.

In diesem Zusammenhang fällt auch auffällig auf, dass die mae der *All*-Updatestrategien der Outputvorhersage für die Einstellung *mit Löschen* sogar noch deutlich größer sind. Das liegt daran, dass die Vorhersagemodelle so gut funktionieren bzw. im Falle der *All*-Updatestrategien der Outputvorhersage sogar noch viel besser als jene der *Individual*-Updatestrategien bzw. noch länger halten. Deshalb werden im Rahmen der *All*-Updatestrategien der Outputvorhersage sogar noch länger keine Updates geschickt, wodurch noch länger das Backup NN verwendet werden muss, was in der Folge noch größere mae erzeugt.

Betrachtung aller Zeitschritte der Updatestrategien

Für die genauere Untersuchung der Updatestrategien wird in *Abbildung 6.5* der mae für alle Updatestrategien und die Backup-Lösung zu jedem Zeitschritt angezeigt.

Auf der x-Achse sind alle 1.000 Zeitschritte aufgeführt, die die Simulationen umfassen. Die linke y-Achse enthält den zum jeweiligen Zeitschritt dazugehörigen *Winkel* und wird in Schwarz dargestellt. Die rechte y-Achse enthält den mae für jeden Zeitschritt. Dieser wird für die Backup-Lösung mit Blau dargestellt und dient für die Updatestrategien, die mit allen anderen Farben dargestellt werden als Vergleich. Die schraffierte rote Fläche zeigt die Zeitschritte an, während dieser der Arm still steht.

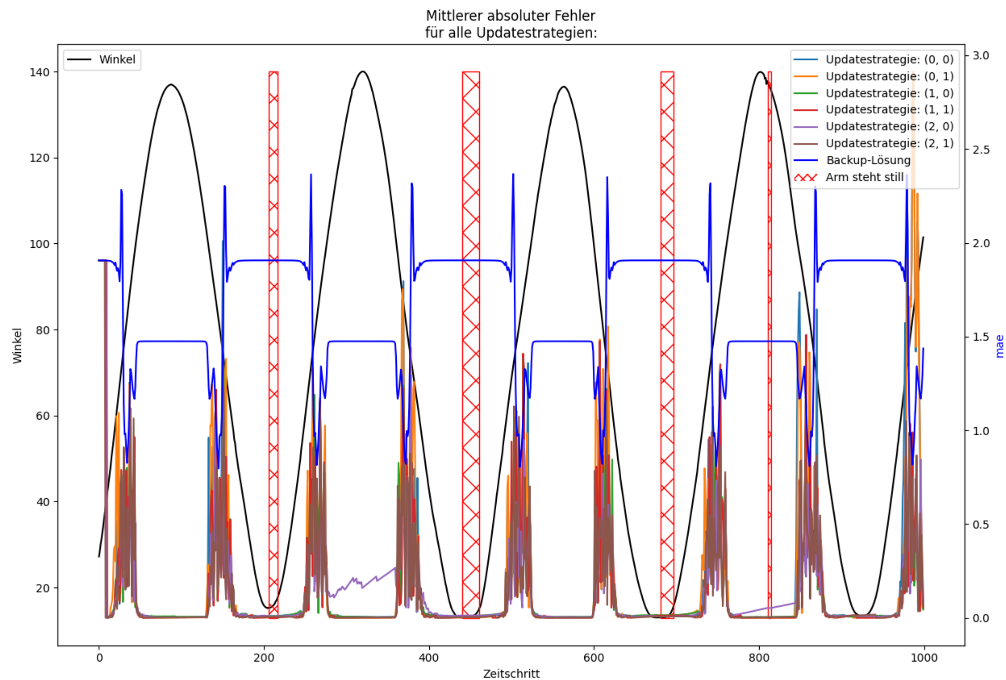

Abbildung 6.5.: Qualität aller Updatestrategien in jedem Zeitschritt (ohne Löschen)

Zunächst weisen alle Updatestrategien für die ersten $\max(m_0) + \max(m_1)$ Zeitschritte denselben mae auf, wie die Backup-Lösung. Dies liegt daran, dass in dieser Zeit, noch keine vom Server berechnete Lösung vorliegt, weshalb auf die Backup-Lösung zurückgegriffen werden muss. Anschließend wird der mae für alle Updatestrategien drastisch besser und geht gegen Null, während die Backup-Lösung eine Lösung erzeugt, deren mae alternierend im Bereich von 1,45 und 1,90 konstant stark von der Referenzlösung abweicht.

Obwohl die Updatestrategien fast immer einen mae aufweisen, der gegen Null geht, wird dieser im Winkelbereich von ca. 80 ± 10 Grad deutlich schlechter. In diesem Bereich verändern sich die Koordinatenwerte des Armes so drastisch, dass die Updatestrategien nicht nur in etwa gleich schlechte mae aufweisen, wie die Backup-Lösung, sondern in einigen Fällen sogar etwas darüber liegen. Die mae für die sechs verschiedenen Updatestrategien, die in *Abbildung 6.4* angegeben sind, ergeben sich fast ausschließlich aus diesem Winkelbereich von ca. 80 ± 10 Grad. Ansonsten würden diese gegen Null gehen. Diese Verschlechterung innerhalb dieses Bereichs würde übrigens auch bei 100% offloading entstehen, falls die Updates verspätet eintreffen bzw. würden sich, falls die Kommunikation eine gewisse Verzögerung aufweist, sogar noch länger anhalten.

Betrachtung aller Zeitschritte der qualitativ besten Updatestrategie

Nach der Gesamtbetrachtung aller Updatestrategien, wird nachfolgend Updatestrategie (2, 1) als qualitativ beste Updatestrategie näher untersucht.

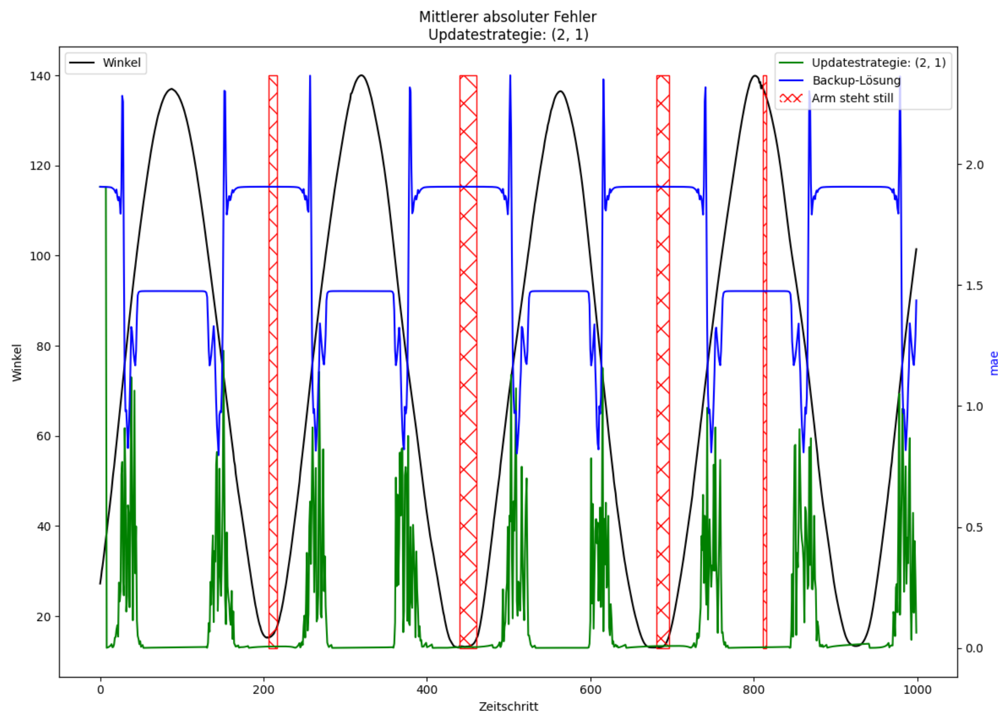

Abbildung 6.6.: Qualität Updatestrategie (2, 1) (ohne Löschen)

In *Abbildung 6.6* wird die Qualität von Updatestrategie (2, 1) über alle 1.000 Zeitschritte der durchgeführten Simulation aufgezeigt.

Die anderen Updatestrategien werden hier nicht näher betrachtet. Die Schaubilder zur Qualität aller anderen Updatestrategien *ohne Löschen* befinden sich im Anhang in *Abschnitt A.3 Implementierungseinstellung* ohne Löschen.

Der einzige Unterschied zu *Abbildung 6.5* liegt darin, dass alle anderen Updatestrategien entfernt wurden. Der mae von Updatestrategie (2, 1) läuft, wie auch allgemein zuvor für alle Updatestrategien beschrieben, fast überall gegen Null, aber im Übergangsbereich von ca. 80 ± 10 Grad befinden sich dessen mae erkennbar unterhalb denen der Backup-Lösung. Überdies weißt Updatestrategie (2, 1) keinerlei Ausreißer auf.

Die Qualität, die sich *mit Löschen* ergibt, kann in *Abbildung A.4* eingesehen werden.

Abschließend werden nachfolgend noch die Updatestrategien (2, 1) in *Abbildung A.11* und (2, 0) in *Abbildung A.10*, beide für die Einstellung *mit Löschen* betrachtet.

In Updatestrategie (2, 0) *mit Löschen* in *Abbildung A.10* erfolgen zu den Zeitschritten 88, 204, 320, 439, 564, 677, 802 und 925 Bewegungsrichtungsänderungen. Dies kann sowohl am *Winkel* in Schwarz abgelesen werden, als auch dass an der Stelle angefangen wird die Backup-Lösung zu verwenden, da der mae ab der Stelle identisch ist mit der Backup-Lösung, da diese ab dann verwendet wird. Die Updatestrategie verwendet nur einige wenige Zeitschritte die Backup-Lösung, wohingegen

die Updatestrategie (2, 1) *mit Löschen* in *Abbildung A.11* jedes Mal mehrere Dutzend Zeitschritte lang die Backup-Lösung verwendet bzw. diese verwendet muss. Weil die Vorhersagemodelle sogut funktionieren bzw. sogar noch besser für die *All*-Updatestrategien, halten diese noch länger und es erfolgen für noch längere Zeitschritte keine Updates, wodurch noch länger die Backup-Lösung verwendet werden muss, die aufgrund der schlechteren Outputergebnisse, den mae noch mehr erhöht.

6.4. Empirische Ergebnisse mit Bandbreitenoptimierung der Outputparameter

In diesem Abschnitt erfolgt eine erneute Auswertung mit Bandbreitenoptimierung der Outputparameter. Hier wurden die Parameter der Simulationen vorab nicht unter dem ausschließlichen Aspekt größtmöglicher Qualität optimiert, sondern einschließlich einer angemessenen Berücksichtigung der benötigten Bandbreiten. Die Parameter der Inputvorhersage wurden auf einer durchschnittlichen Vorhersagedauer von 4 Zeitschritten belassen, jedoch wurden die Parameter der Outputvorhersage auf eine durchschnittliche Vorhersagedauer von 30 Zeitschritten erhöht.

Ein alternativer Ansatz, in dem sowohl die Parameter der Inputvorhersage als auch jene der Outputvorhersage auf eine durchschnittliche Vorhersagedauer von 30 Zeitschritten hin optimiert wurden, verursacht einen zu großen Fehler und wird nicht weiter betrachtet. Dieser kann jedoch in *Abschnitt A.4 30 Zeitschritte bandbreitenoptimierter Input und Output ohne Löschen* eingesehen werden.

Die Einstellung *mit Löschen* wird ebenfalls nicht weiter untersucht, da diese wie bereits zuvor festgestellt und ausführlich erläutert für durchgängig schlechtere Ergebnisse sorgt.

6.4.1. Kommunikation

In *Abbildung A.28* wird die Kommunikation der Simulationsergebnisse veranschaulicht.

Diesen kann entnommen werden, dass alle Inputvorhersagen mit 483, 224 und 444 Sendevorgängen identisch viele Kommunikationsvorgänge wie zuvor ohne Bandbreitenoptimierung verursachen, jedoch weisen die Outputvorhersagen für die *Individual*-Updatestrategien mit 616, 595 und 606 Sendevorgängen verringerte und für die *All*-Updatestrategien mit 139, 118 und 130 Sendevorgängen sehr stark verringerte Kommunikationsvorgänge wie zuvor ohne Bandbreitenoptimierung der Outputparameter auf.

Die Parameter der Outputvorhersage wurden so optimiert, dass diese durchschnittlich 30 Zeitschritte halten sollen, weshalb diese so stark verringert sind. Diese Verringerung wirkt sich aufgrund der Vielzahl der Outputparameter stärker auf die *All*-Updatestrategien als auf die *Individual*-Updatestrategien der Outputvorhersage aus.

6.4.2. Vorhersagedauer

In *Abbildung A.29* werden die durchschnittlichen Vorhersagedauern der Simulationsergebnisse veranschaulicht.

Diese entsprechen in etwa den Werten für die diese zuvor optimiert wurden. Im Falle der Parameter der Inputvorhersage halten diese mit 4,76 Zeitschritten genauso lange wie zuvor ohne Bandbreitenoptimierung. Die Parameter der Outputvorhersage dagegen halten durchschnittlich ca. 25 Zeitschritte und damit etwas unterhalb von 30 Zeitschritten auf die diese zuvor optimiert wurden.

6.4.3. Bandbreite

In *Abbildung A.30* werden die Bandbreiten der Simulationsergebnisse veranschaulicht.

Die Inputvorhersagen benötigen mit 23.736, 17.920 und 35.520 Bytes weiterhin identisch viel Bandbreite wie ohne Bandbreitenoptimierung. Dagegen weisen die Outputvorhersagen stark reduzierte Bandbreitenbedarfe auf. Im Falle der *Individual*-Updatestrategien der Outputvorhersage werden mit 135.696, 127.040 und 129.360 Bytes weniger als 20% des 100% offloadings gebraucht. Die *All*-Updatestrategien der Outputvorhersage weisen mit 302.464, 256.768 und 282.880 Bytes sehr viel höhere Bandbreitenbedarfe auf, jedoch stellt dies eine große Einsparung im Vergleich zu ohne Bandbreitenoptimierung dar und entspricht im Vergleich zum 100% offloading teilweise weniger als 40% des Bedarfes.

Die Gesamtbandbreiten für die *Individual*-Updatestrategien der Outputvorhersage entsprechen mit 159.432, 144.960 und 164.880 Bytes ebenfalls ca. 20% der Bandbreite im Vergleich zum 100% offloading. Die Gesamtbandbreiten für die *All*-Updatestrategien der Outputvorhersage entsprechen mit 326.200, 274.688 und 318.400 Bytes ebenfalls ca. 40% der Bandbreite im Vergleich zum 100% offloading bzw. im Falle von Updatestrategie (1, 1) sogar nur noch 36% der Bandbreite. Mit der Bandbreitenoptimierung wird der große Vorteil die Modelldaten, statt die Historienwerte zu schicken offensichtlich.

6.4.4. Qualität

In *Abbildung A.31* wird der mae, den die Simulationen erzeugen angezeigt. Diese sind mit 0,542, 0,594 und 0,492 für die *Individual*-Updatestrategien der Outputvorhersage relativ hoch. Jedoch sind diese mit 0,295, 0,22 und 0,228 für die *All*-Updatestrategien der Outputvorhersage in einem vergleichsweise akzeptablen Bereich. Die mae von 0,22 und 0,228 sind dabei in etwa doppelt so hoch wie der mae von 0,122 der qualitativ besten Updatestrategie ohne Bandbreitenoptimierung.

Auch hier kann wieder der maßgebliche Einfluss des Winkels auf die Gesamtqualität ausgemacht werden. Die Parameter der Inputvorhersage wurden mit einer Optimierung von 4 Zeitschritten auf größtmöglicher Qualität belassen und jene der Outputvorhersage von 4 auf 30 Zeitschritte erhöht. Dies sorgt nicht einmal für eine Verdopplung des mae zwischen der qualitativ besten Updatestrategie mit und ohne Bandbreitenoptimierung, die im ersteren Fall Updatestrategie (1, 1) und im letzteren

Fall Updatestrategie (2, 1) ist. Sofern man dagegen die Parameter der Inputvorhersage ebenfalls auf 30 Zeitschritte durchschnittlicher Vorhersagedauer erhöhen würde, ergebe dies mit 0,383 wie in *Abbildung A.20* eingesehen werden kann, mehr als eine Verdreifachung des mae.

In jedem Fall stellen alle Updatestrategien mit Bandbreitenoptimierung immer noch eine deutliche Verbesserung dar. Die Fehlerreduzierung würde immer noch ca. 86% gegenüber einer rein lokalen Ausführung des Clients mit Hilfe der Backup-Lösung entsprechen.

Alle weiteren Schaubilder zur Auswertung der Simulationen mit Bandbreitenoptimierung der Outputparameter können in *Abschnitt A.5 30 Zeitschritte bandbreitenoptimierter Output ohne Löschen* eingesehen werden.

6.5. Diskussion und Interpretation

Zum Abschluss des Kapitels findet in diesem Abschnitt eine Zusammenfassung, sowie Diskussion und Interpretation der Ergebnisse statt.

Die Vorhersagemethoden zu den Input- und Outputparametern halten durchschnittlich, mehr als die geforderten drei Zeitschritte. Die Inputvorhersage hält mit durchschnittlich 4,39 Zeitschritten, am längsten für die *Individual*-Updatestrategien. Deutlich kürzer mit 3,35 Zeitschritten halten die *Master*-Updatestrategien der Inputvorhersage. Am kürzesten hält die Inputvorhersage mit durchschnittlich 2,17 Zeitschritten für die *All*-Updatestrategien, da jede Neuerstellung eines Parameters auch die Neuerstellung und Verschickung von allen anderen auslöst. Die Updatestrategien der Outputvorhersage, die auf Grundlage der Inputvorhersage erfolgen, erfüllen mit durchschnittlich ca. 3 Zeitschritten gerade so die geforderte Mindestanzahl von durchschnittlich über drei Zeitschritten bis ein neues Vorhersagemodell gebildet werden muss.

Sämtliche realisierten Updatestrategien stellen eine erhebliche Verbesserung gegenüber einer rein lokalen Ausführung des Clients mit Hilfe der Backup-Lösung dar. Der Fehler wird je nach gewählter Updatestrategie um 88,43% - 92,57% reduziert. Obwohl jede der implementierten Updatestrategien eine sehr starke Qualitätsverbesserung im Vergleich zur Backup-Lösung darstellt, gibt es auffällige Qualitätsunterschiede zwischen den verschiedenen Updatestrategien. Die mit Abstand qualitativ beste Lösung stellt die Updatestrategie (2, 1) bzw. allgemein die *All*-Updatestrategien der Inputvorhersage dar. Diese wird gefolgt von den *Master*-Updatestrategien der Inputvorhersage. Das Schlusslicht bilden die *Individual*-Updatestrategien der Inputvorhersage, deren mae ca. 50% höher ist als jene der Updatestrategie (2, 1), die die qualitativ beste Lösung darstellt. Zusammengefasst hängen die Qualität der Updatestrategien zwar nicht ausschließlich, aber im Wesentlichen von der Updatestrategie der Inputvorhersage ab. Der *Winkel* übt einen überragenden Einfluss auf das Ergebnis aus, was sich in sämtlichen Ergebnissen widerspiegelt.

Die *All*-Updatestrategie der Outputvorhersage sorgt im Regelfall für eine weitere Qualitätssteigerung im Gegensatz zur *Individual*-Updatestrategie der Outputvorhersage.

Die Einstellung *mit Löschen* liefert durchgängig sehr viel schlechtere Ergebnisse, weshalb ausschließlich die Einstellung *ohne Löschen* verwendet werden sollte. Der Grund liegt darin, dass nach einer Bewegungsrichtungsänderung oftmals lange Zeit keine Updates geschickt werden. Je besser und länger die Vorhersagemodelle halten, umso länger dauert es bis die Updates geschickt werden, weshalb solange auf die Backup-Lösung zurückgegriffen werden muss, die ungenauere

Ergebnisse produziert. Aus demselben Grund ist die Qualität *mit Löschen* bei Verwendung der All-Updatestrategie der Outputvorhersage sogar noch schlechter, da diese Vorhersagemodelle noch länger und besser funktionieren, weshalb noch länger die Backup-Lösung verwendet werden muss.

Am wenigsten Bandbreite benötigen diejenigen Updatestrategien, deren Outputvorhersage, die *Individual*-Updatestrategie verwendet, wovon die Updatestrategie (1, 0), die mit Abstand geringste Bandbreite benötigt, da in diesem nur der *Winkel* Sendevorgänge für die Inputvorhersage auslöst.

Die meiste Bandbreite wird von denjenigen Updatestrategien benötigt, deren Outputvorhersage, die *All*-Updatestrategie verwendet. Damit kann zusammengefasst werden, dass die benötigte Bandbreite in etwa mit der Anzahl der Parameter korreliert, wovon es im Falle der Outputvorhersage 30mal mehr gibt als für die Inputvorhersage.

Abgesehen von Updatestrategie (1, 0) wird mehr Bandbreite benötigt als wenn 100% offloading betrieben werden würde. Das verteilte System ist auf bestmögliche Qualität ausgelegt, weshalb die Schwellenwerte möglichst niedrig eingestellt wurden, sodass die Vorhersagemethoden nur durchschnittlich knapp über 3 Zeitschritte halten, was in höherer Kommunikation und größerem Bandbreitenbedarf resultiert.

Abhängig von den zur Verfügung stehenden Ressourcen und den eigenen Anforderungen, ergeben sich unterschiedliche Empfehlungen. Für den Fall, dass die Qualität der Ergebnisse von größter Wichtigkeit ist, empfiehlt sich die Verwendung von Updatestrategie (2, 1), die mit Abstand die qualitativ besten Ergebnisse liefert. Sollte jedoch Bandbreite ein kritischer Faktor sein, kehrt sich dies um, da diese Updatestrategie sehr viel Bandbreite benötigt.

Falls nicht Maximalanforderungen an die Qualität gestellt werden, sondern Bandbreite ein kritischer Faktor ist, der ebenfalls mit berücksichtigt werden soll, empfiehlt es sich statt bloß die Updatestrategie zu wechseln, die Schwellenwerte der Parameter der Outputvorhersage anzuheben, da die Updatestrategien ansonsten alle vergleichsweise sehr viel Bandbreite benötigen. Durch die Anhebung der Schwellenwerte der Parameter der Outputvorhersage kann sehr viel Bandbreite eingespart werden, ohne das die Qualität zu sehr darunter leidet. Die Parameter der Outputvorhersage machen den allergrößten Teil des Bandbreitenbedarfes aus, üben aber nur einen sehr kleinen Einfluss auf die Gesamtqualität des Ergebnisses aus. Es empfiehlt sich nicht die Schwellenwerte der Parameter der Inputvorhersage zu erhöhen, da der Winkel, der einen überragenden Einfluss auf das Gesamtergebnis hat in diesen enthalten ist und überdies die Parameter der Inputvorhersage sowieso nicht nennenswert viel Bandbreite benötigen.

Für den Trade-off zwischen Qualität und Bandbreite sollten die Parameter der Inputvorhersage in jedem Fall bei größtmöglicher Qualität belassen werden und statt dessen nur die Parameter der Outputvorhersage so angehoben werden, dass diese den eigenen Anforderungen bestmöglich entsprechen. Für maximal mögliche Qualität sollte Updatestrategie (2, 1) verwendet werden und die Parameter der Outputvorhersage mit 4 Zeitschritten durchschnittlicher Vorhersagedauer laufen. Je nach Bandbreitenanforderung sollten die Parameter der Outputvorhersage auf bis zu 30 Zeitschritte angehoben werden. Im Falle der Erhöhung der Parameter der Outputvorhersage auf 30 Zeitschritte durchschnittlicher Vorhersagedauer würde die Fehlerreduzierung bei Updatestrategie (1, 1) immer noch ca. 86% gegenüber einer rein lokalen Ausführung des Clients mit Hilfe der Backup-Lösung betragen. Gleichzeitig würde dabei die Bandbreite auf ca. $\frac{1}{3}$ absinken im Vergleich zu vor der Erhöhung und damit nur noch ca. 36% der benötigten Bandbreite eines 100% offloadings entsprechen.

7. Fazit

Große NN ermöglichen sehr genaue Berechnungen, benötigen jedoch im Vergleich zu kleineren NN auch deutlich mehr Rechenleistung, die eventuell auf einem Client/Endgerät nicht zur Verfügung stehen. Ein Ansatz diesem Umstand zu begegnen besteht darin, dieses große NN auf einem entfernten Server zu betreiben, an den der Client seine Berechnungen auslagert. Um die Anzahl der dafür nötigen Kommunikationsvorgänge zu reduzieren, können Vorhersagemethoden verwendet werden.

Im Rahmen dieser Bachelorarbeit wird die gedämpfte Holt'ʻs Methode, die in [BM22] zur *erweiterten gedämpften Holtʻs Methode* weiterentwickelt wurde verwendet, um zukünftige Inputwerte einer aufgezeichneten Armbewegung und zukünftige Outputwerte von Oberarmkoordinatenwerten vorherzusagen.

Vor der Verwendung des verteilten Systems wurden die Parameter zur Anwendung der Vorhersagemethode bestimmt, sowie die Schwellenwerte zu allen Parametern, die aussagen, ab wann ein Vorhersagemodell nicht mehr gut genug ist, sodass dieses neu gebildet und an den Client bzw. den Server zur gespiegelten Verwendung dessen geschickt werden muss.

Die *erweiterte gedämpfte Holtʻs Methode* wird im Kontext des verteilten Systems verwendet, welches gespiegelt auf dem Server und dem Client zur Anwendung kommt, um sowohl die zukünftigen Inputwerte als auch die zukünftigen Outputwerte vorherzusagen. Der Client schickt die Modelldaten seiner Inputparameter an den Server, aus denen der Server exakt dasselbe Vorhersagemodell bildet, welches dann sowohl auf dem Client als auch auf dem Server betrieben wird. Umgekehrt schickt der Server die Modelldaten seiner Outputparameter an den Client, sodass dieser mit dessen Rekonstruktion exakt dieselben Vorhersagemodelle zu den Outputparametern gespiegelt bei sich betreiben kann. Ein großer Vorteil die gespiegelten Vorhersagemodelle mit den Modelldaten aufbauen zu können, besteht darin, dass verspätete Updates, die bereits ab einem vergangenen Zeitschritt gültig wurden, immer noch verwendet werden können, indem diese soviele Zeitschritte weiter in die Zukunft vorhergesagt werden, die der Differenz zwischen dem aktuellen Zeitschritt und dem Zeitschritt, ab dem diese gültig wurden entspricht. Im Falle eines 100% offloadings wären verspätete Updates u.U nicht mehr zu gebrauchen.

Ein kleines NN, welches weniger genaue Koordinatenwerte berechnet, wird auf dem Client als Backup-Lösung betrieben, falls kein Modell des Servers zur Verfügung stehen sollte. Es wurde zudem ein Mechanismus auf dem Client entwickelt, der eventuelle Bewegungsstillstände des Armes detektiert und in der Folge die clientseitige Kommunikation zum Server einstellt. Darüber hinaus wurde mit der Modellverwaltung des Clients ein Verfahren entwickelt, dass Servermodelle, die für zukünftige Zeitschritte schon im Voraus berechnet wurden, beim Client vorgehalten werden können, noch bevor diese gebraucht werden. Diese können beim Client bereitstehen und sofort verwendet werden, sobald der Zeitschritt, ab dem diese gültig werden, schließlich eintritt.

Überdies wurden verschiedene Updatestrategien realisiert, die festlegen wann und welche Vorhersagemodelle an den Client bzw. den Server geschickt werden sollen. Im Rahmen der *Individual*-Updatestrategie werden die Modelldaten aller Vorhersagemodelle, die sich im jeweiligen Zeitschritt geändert haben geschickt. Die *Master*-Updatestrategie sendet bei der Änderung des Masterparameters, die Vorhersagemodelle zu allen Parametern, einschließlich jener, die sich in dem jeweiligen Zeitschritt nicht geändert haben, wofür diese vor Versendung neu gebildet werden. In der *All*-Updatestrategie löst die Veränderung jedes beliebigen Parameters den Sendevorgang aus, der genauso wie die *Master*-Updatestrategie, die Vorhersagemodelle zu allen Parametern schickt.

Auf dem Client können alle drei Updatestrategien betrieben werden mit dem *Winkel* als Masterparameter der *Master*-Updatestrategie, da dieser einen überragenden Einfluss auf das Ergebnis der NN hat. Da für die Outputvorhersage kein Parameter existiert, der einen solch entscheidenden Einfluss ausübt, werden auf dem Server nur die *Individual* und *All*-Updatestrategien betrieben. Daraus ergeben sich die sechs verschiedenen (Input, Output)-Updatestrategiepaare: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1).

Alle Updatestrategien im Rahmen des verteilten Systems sind in der Lage den Fehler der Outputergebnisse verglichen mit einer rein lokalen Ausführung des Backup NN auf dem Client, zwischen 88,43% - 92,57% zu reduzieren. Da die Updatestrategien unterschiedlich gute Ergebnisse liefern und dazu unterschiedlich viel Bandbreite benötigen, ergeben sich in Abhängigkeit von den eigenen Anforderungen und den zur Verfügung stehenden Ressourcen, verschiedene Updatestrategien, die verwendet werden sollten.

Falls die maximale Qualität gefordert ist, bietet sich die Updatestrategie (2, 1) an, die mit Abstand die besten Ergebnisse ermöglicht. Jedoch benötigt diese Updatestrategie auch am zweitmeisten Bandbreite.

Das verteilte System ist dahingehend auf größtmögliche Qualität ausgerichtet, dass alle Schwellenwerte der Parameter, sowohl der Input- und Outputvorhersage soweit abgesenkt wurden, dass die Vorhersagedauern im Durchschnitt ca. 4 Zeitschritte lang halten, bevor ein neues Vorhersagemodell gebildet werden muss. Dies sorgt für qualitativ sehr gute Ergebnisse, sorgt in der Konsequenz jedoch für sehr viel Kommunikation innerhalb des verteilten Systems und damit einhergehend einem sehr großen Bandbreitenbedarf. Falls kein Höchstmaß an Qualität gefordert ist, empfiehlt es sich statt bloß die Updatestrategie zu wechseln, die Schwellenwerte der Parameter der Outputvorhersage zu erhöhen, wodurch die Vorhersagemodelle im Durchschnitt länger halten, was in reduzierter Kommunikation und Bandbreitenbedarf, und nur einem geringen Qualitätsverlust einhergeht. Eine Erhöhung der Parameter der Outputvorhersage von 4 auf 30 Zeitschritte, würde bei Verwendung von Updatestrategie (1, 1) immer noch 86% des Fehlers gegenüber einer rein lokalen Ausführung des Clients mit Hilfe der Backup-Lösung vermeiden. Gleichzeitig könnte damit die Bandbreite auf ca. $\frac{1}{3}$ im Vergleich zu vor der Erhöhung reduziert werden, was nur noch ca. 36% eines 100% offloadings entspricht.

Für die Berechnung der Outputergebnisse werden fast ausschließlich die Servermodelle verwendet. Lediglich in den ersten $\max(m_0) + \max(m_1)$ Zeitschritten muss lokal auf die Backup-Lösung zurückgegriffen werden, da das allererste Update erst geschickt wird, wenn alle Historienwerte vollständig gefüllt sind.

Eine zukünftige Arbeit könnte dieses verteilte System, welches hier simuliert wird, in ein verteiltes System überführen, in welchem ein mobiles Endgerät und ein Server tatsächlich miteinander kommunizieren.

Literaturverzeichnis

[BM22] F. Belz, B. Mehler. „Prediction supported Continuous Simulations using Neural Networks“. In: (2022) (zitiert auf S. 3, 4, 17, 25, 32–34, 38, 81).

[BMH16] C. Barrios, Y. Motai, D. Huston. „Intelligent forecasting using dead reckoning with dynamic errors“. In: *IEEE Transactions on Industrial Informatics* 12.6 (2016), S. 2217–2227 (zitiert auf S. 23).

[CGL+98] J. H. Chim, M. Green, R. W. Lau, H. Va Leong, A. Si. „On caching and prefetching of virtual objects in distributed virtual environments“. In: *Proceedings of the sixth ACM international conference on Multimedia*. 1998, S. 171–180 (zitiert auf S. 22).

[CLN01] A. Chan, R. W. Lau, B. Ng. „A hybrid motion prediction method for caching and prefetching in distributed virtual environments“. In: *Proceedings of the ACM symposium on Virtual reality software and technology*. 2001, S. 135–142 (zitiert auf S. 21).

[CLS+98] J. H. Chim, R. W. Lau, A. Si, H. Va Leong, D. To, M. Green, M. L. Lam. „Multi-resolution model transmission in distributed virtual environments“. In: *Proceedings of the ACM symposium on Virtual reality software and technology*. 1998, S. 25–34 (zitiert auf S. 22).

[Fie14a] G. Fiedler. „Deterministic Lockstep“. In: (2014). URL: https://gafferongames.com/post/deterministic_lockstep/ (zitiert auf S. 19).

[Fie14b] G. Fiedler. „Introduction to Networked Physics“. In: (2014). URL: https://gafferongames.com/post/introduction_to_networked_physics/ (zitiert auf S. 19).

[Fie14c] G. Fiedler. „Snapshot Interpolation“. In: (2014). URL: https://gafferongames.com/post/snapshot_interpolation/ (zitiert auf S. 20).

[Fie15a] G. Fiedler. „Snapshot Compression“. In: (2015). URL: https://gafferongames.com/post/snapshot_compression/ (zitiert auf S. 20).

[Fie15b] G. Fiedler. „State Synchronization“. In: (2015). URL: https://gafferongames.com/post/state_synchronization/ (zitiert auf S. 20).

[FSWB99] M. Faerman, A. Su, R. Wolski, F. Berman. „Adaptive performance prediction for distributed data-intensive applications“. In: *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. 1999, 36–es (zitiert auf S. 22).

[Gam22a] G. Gambetta. „Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation“. In: (2022). URL: https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html (zitiert auf S. 21).

[Gam22b] G. Gambetta. „Fast-Paced Multiplayer (Part III): Entity Interpolation“. In: (2022). URL: https://www.gabrielgambetta.com/entity-interpolation.html (zitiert auf S. 21).

[HA18] R. J. Hyndman, G. Athanasopoulos. „Forecasting: Principles and Practice“. In: (2018). URL: https://otexts.com/fpp2/holt.html (zitiert auf S. 34).

[LZL+16] D. Liu, L. Zeng, C. Li, K. Ma, Y. Chen, Y. Cao. „A distributed short-term load forecasting method based on local weather information“. In: *IEEE Systems journal* 12.1 (2016), S. 208–215 (zitiert auf S. 24).

[Tay03] J. W. Taylor. „Exponential smoothing with a damped multiplicative trend“. In: *International Journal of Forecasting* 19.4 (2003), S. 715–725. ISSN: 0169-2070. DOI: https://doi.org/10.1016/S0169-2070(03)00003-7. URL: https://www.sciencedirect.com/science/article/pii/S0169207003000037 (zitiert auf S. 32, 34).

[WSH99] R. Wolski, N. T. Spring, J. Hayes. „The network weather service: A distributed resource performance forecasting service for metacomputing“. In: *Future Generation Computer Systems* 15.5-6 (1999), S. 757–768 (zitiert auf S. 22).

Alle URLs wurden zuletzt am 01. 11. 2022 geprüft.

A. Anhang

A.1. Parameterbestimmungen für alternative Vorhersagehorizonte

A.1.1. Vorhersagehorizont von 5 Zeitschritten

Parameter	γ	m	*Schwellenwert*
Winkel	0,39	3	0,020958749999999998
Geschwindigkeit	0,01	4	0,20047500000000001
Beschleunigung	0,01	2	2,25

Tabelle A.1.: Parameterwerte der Inputvorhersage (5 Zeitschritte)

Parameter	γ	m	*Schwellenwert*
Outputwerte	0,01 - 0,14	2 - 6	0,0001245902734398924 - 0,002373046875

Tabelle A.2.: Spannweite der Parameterwerte der Outputvorhersage (5 Zeitschritte)

A.1.2. Vorhersagehorizont von 10 Zeitschritten

Parameter	γ	m	*Schwellenwert*
Winkel	0,19	3	0,03207600000000001
Geschwindigkeit	0,01	6	0,552
Beschleunigung	0,01	2	7,4399999999999995

Tabelle A.3.: Parameterwerte der Inputvorhersage (10 Zeitschritte)

Parameter	γ	m	*Schwellenwert*
Outputwerte	0,01 - 0,14	2 - 10	0,0007403906249999949 - 0,007187499999999994

Tabelle A.4.: Spannweite der Parameterwerte der Outputvorhersage (10 Zeitschritte)

A.1.3. Vorhersagehorizont von 20 Zeitschritten

Parameter	γ	m	*Schwellenwert*
Winkel	0,36	6	0,08262
Geschwindigkeit	0,01	2	1,3536000000000001
Beschleunigung	0,01	2	32,0

Tabelle A.5.: Parameterwerte der Inputvorhersage (20 Zeitschritte)

Parameter	γ	m	*Schwellenwert*
Outputwerte	0,01 - 0,01	2 - 2	0,002109375000000023 - 0,016171875000000006

Tabelle A.6.: Spannweite der Parameterwerte der Outputvorhersage (20 Zeitschritte)

A.1.4. Vorhersagehorizont von 30 Zeitschritten

Parameter	γ	m	*Schwellenwert*
Winkel	0,49	10	0,15146068499999998
Geschwindigkeit	0,01	2	3,0233088
Beschleunigung	0,01	2	77,76

Tabelle A.7.: Parameterwerte der Inputvorhersage (30 Zeitschritte)

Parameter	γ	m	*Schwellenwert*
Outputwerte	0,01 - 0,01	2 - 2	0,0028249410656250376 - 0,037209374999999996

Tabelle A.8.: Spannweite der Parameterwerte der Outputvorhersage (30 Zeitschritte)

A.2. Implementierungseinstellung *mit Löschen*

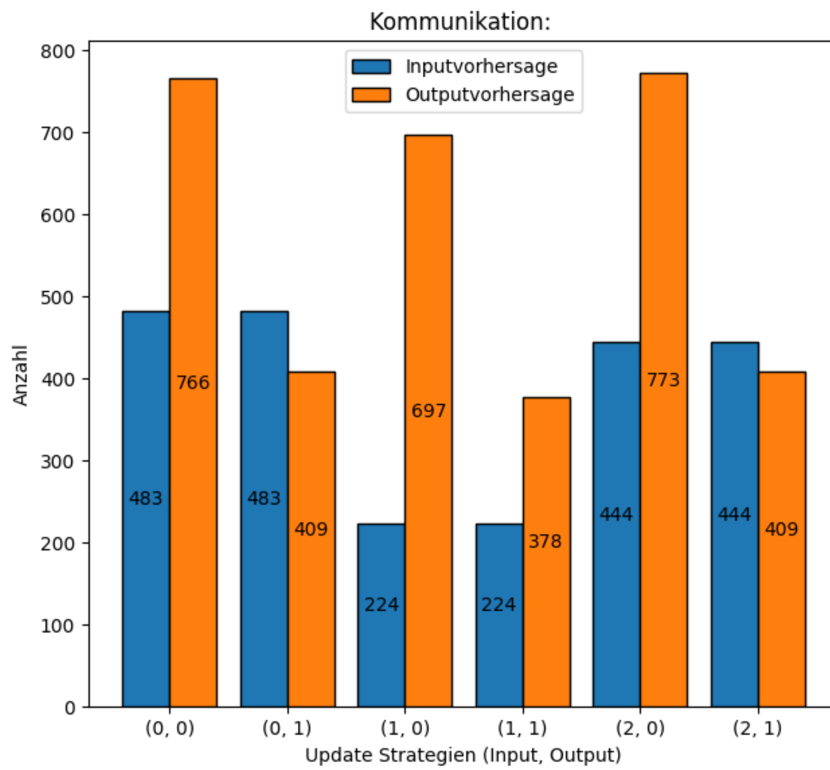

Abbildung A.1.: Kommunikation (mit Löschen)

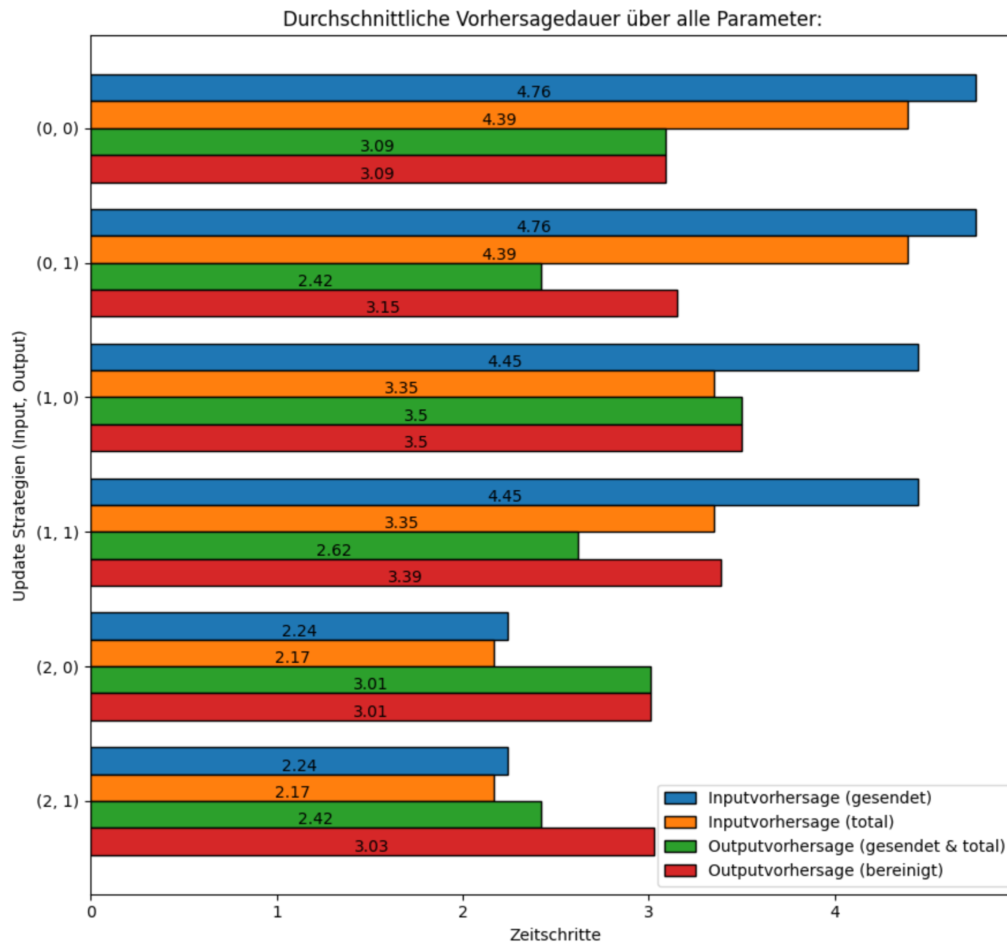

Abbildung A.2.: Vorhersagedauern (mit Löschen)

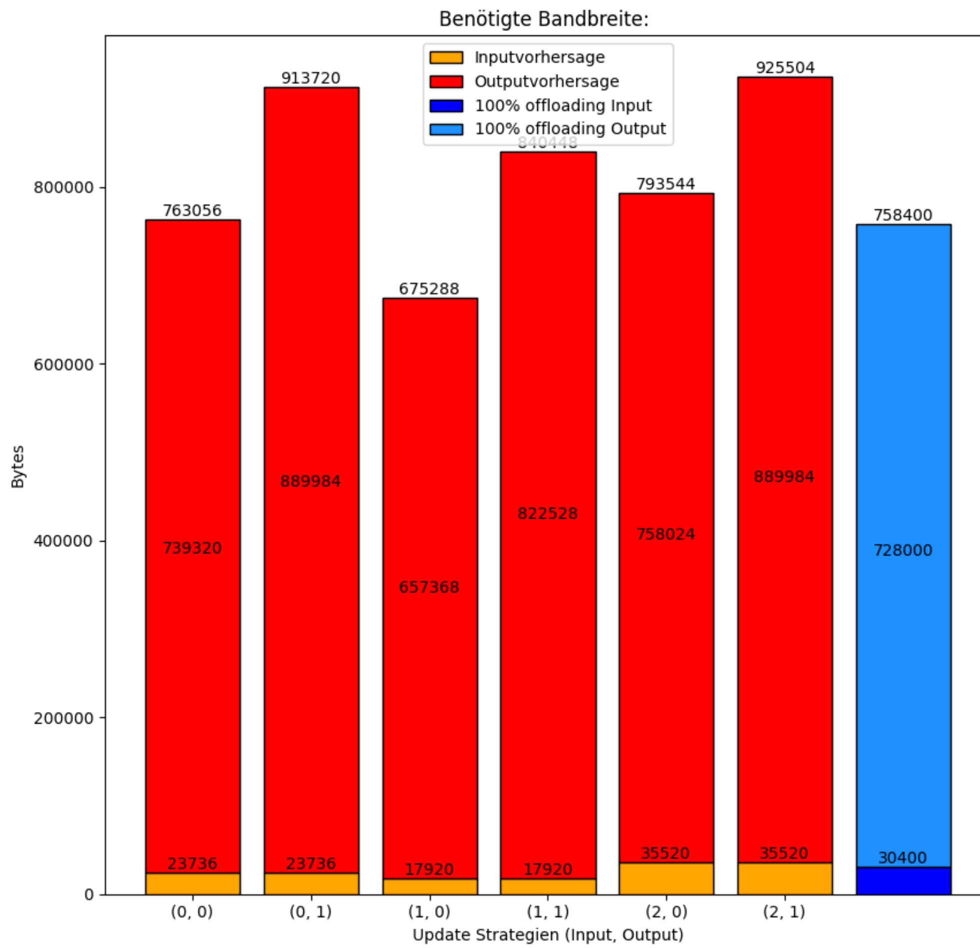

Abbildung A.3.: Bandbreite (mit Löschen)

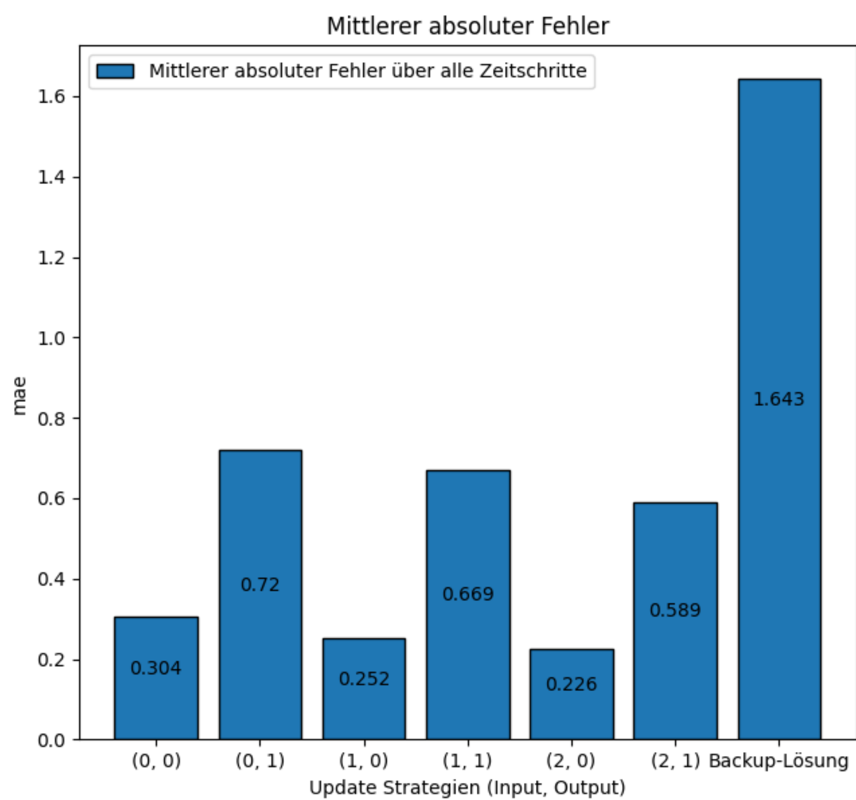

Abbildung A.4.: Gesamtqualität (mit Löschen)

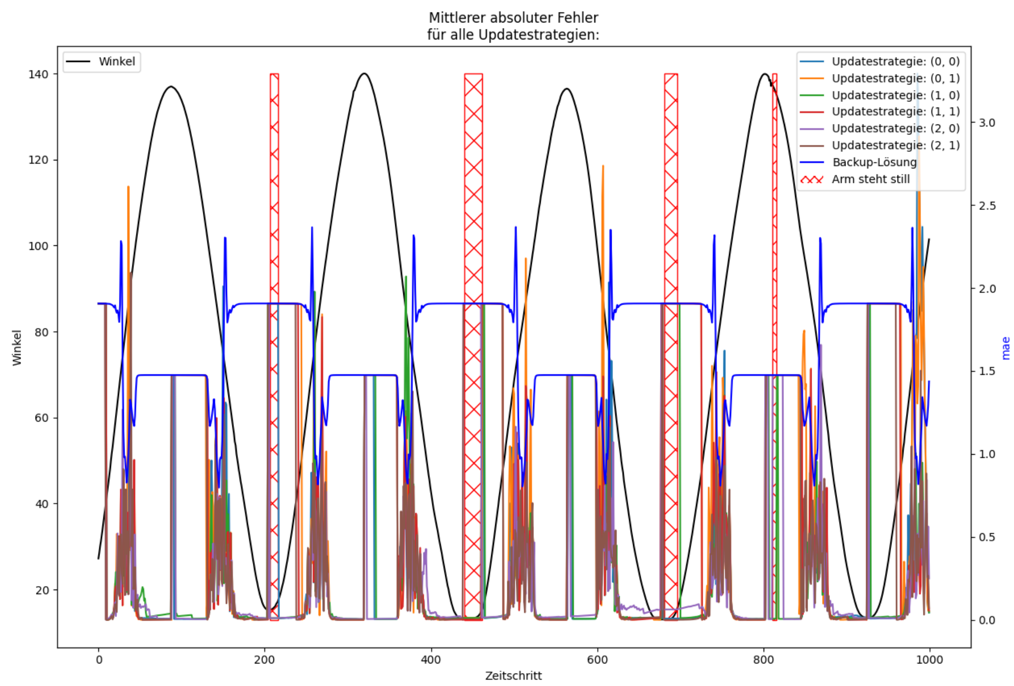

Abbildung A.5.: Qualität aller Updatestrategien in jedem Zeitschritt (mit Löschen)

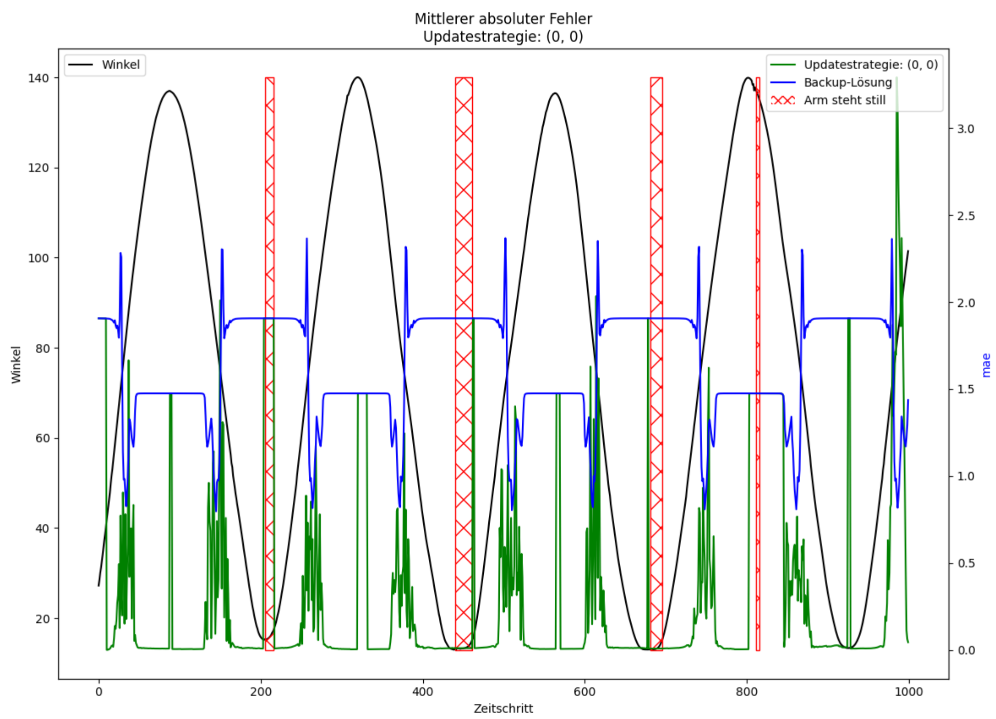

Abbildung A.6.: Qualität Updatestrategie (0, 0) (mit Löschen)

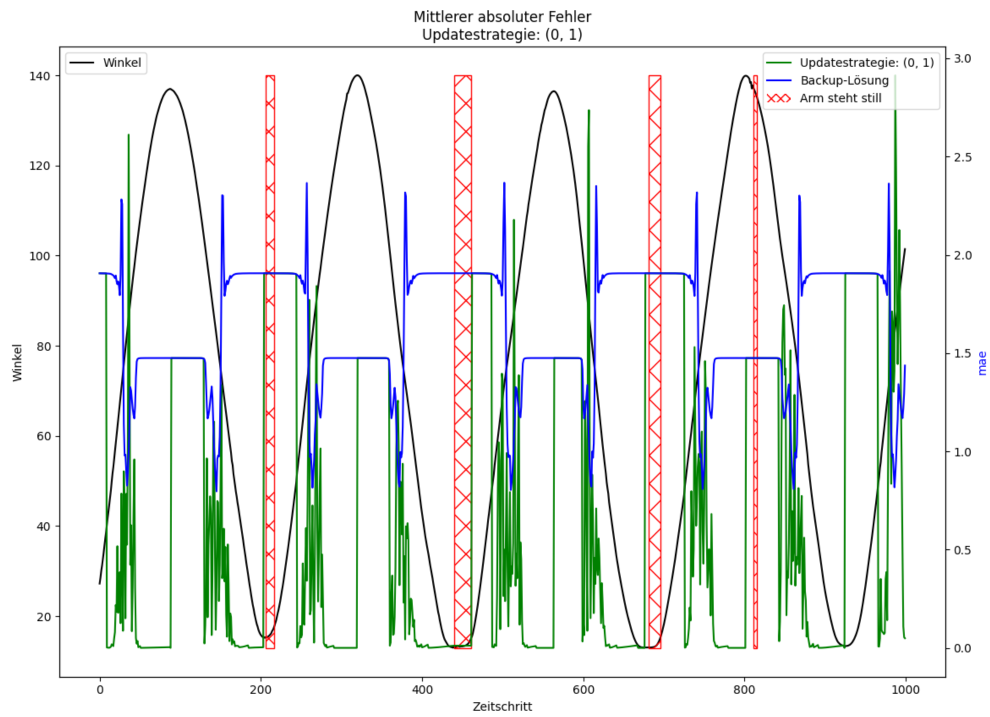

Abbildung A.7.: Qualität Updatestrategie (0, 1) (mit Löschen)

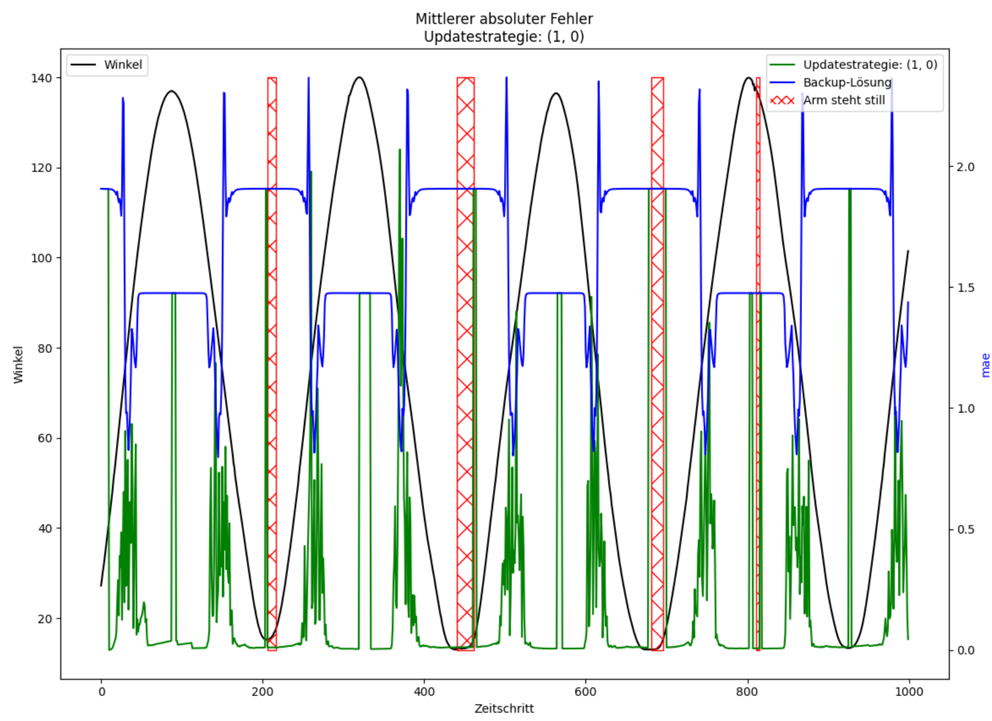

Abbildung A.8.: Qualität Updatestrategie (1, 0) (mit Löschen)

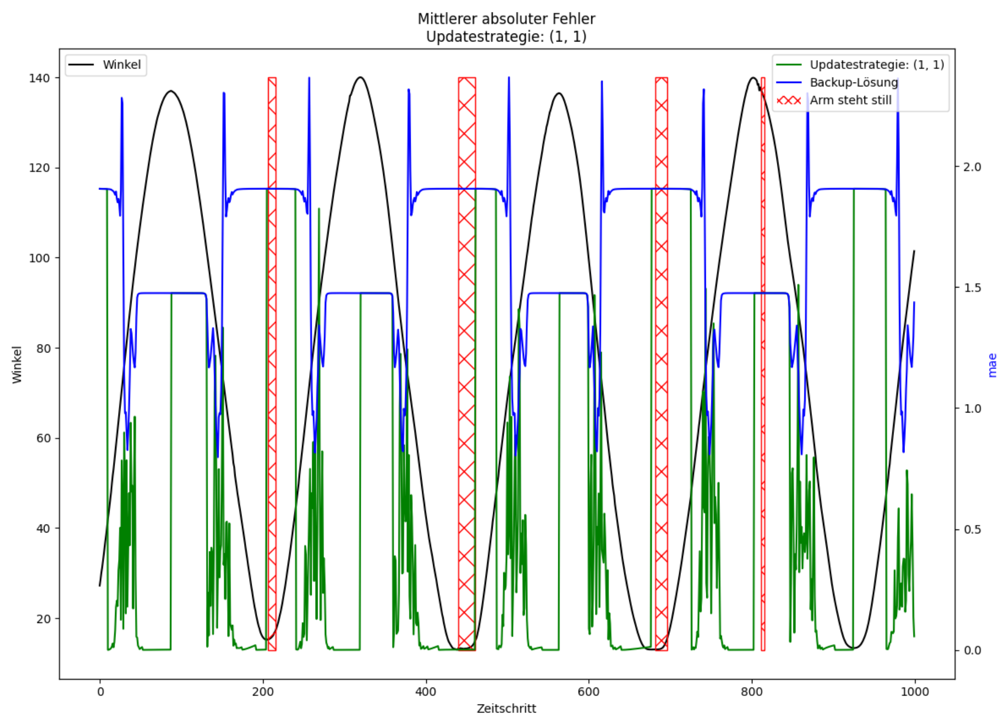

Abbildung A.9.: Qualität Updatestrategie (1, 1) (mit Löschen)

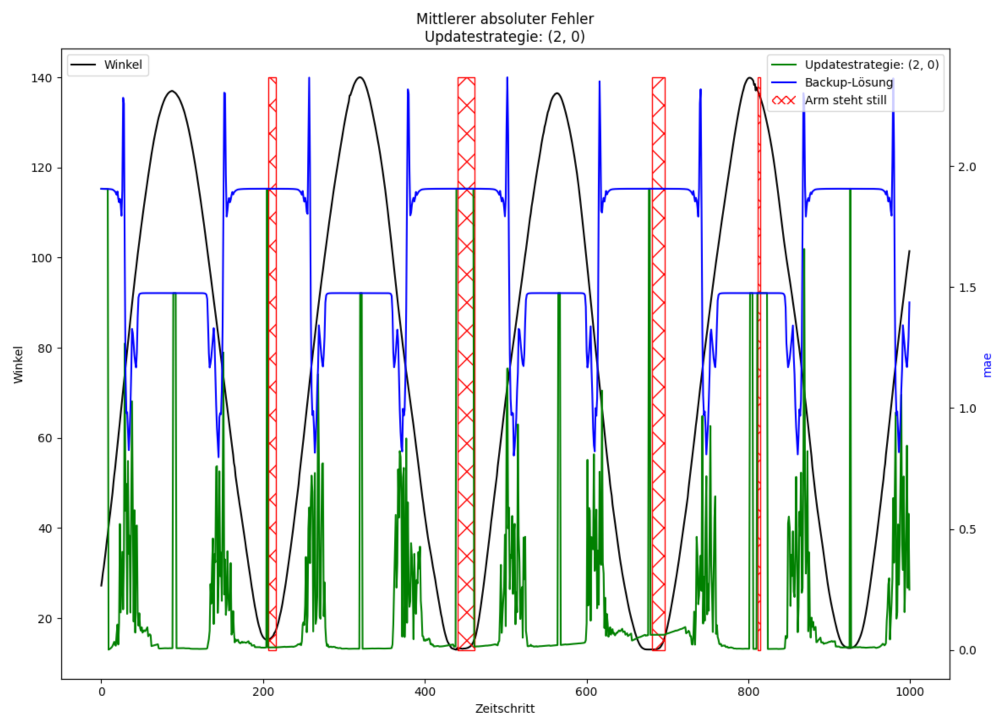

Abbildung A.10.: Qualität Updatestrategie (2, 0) (mit Löschen)

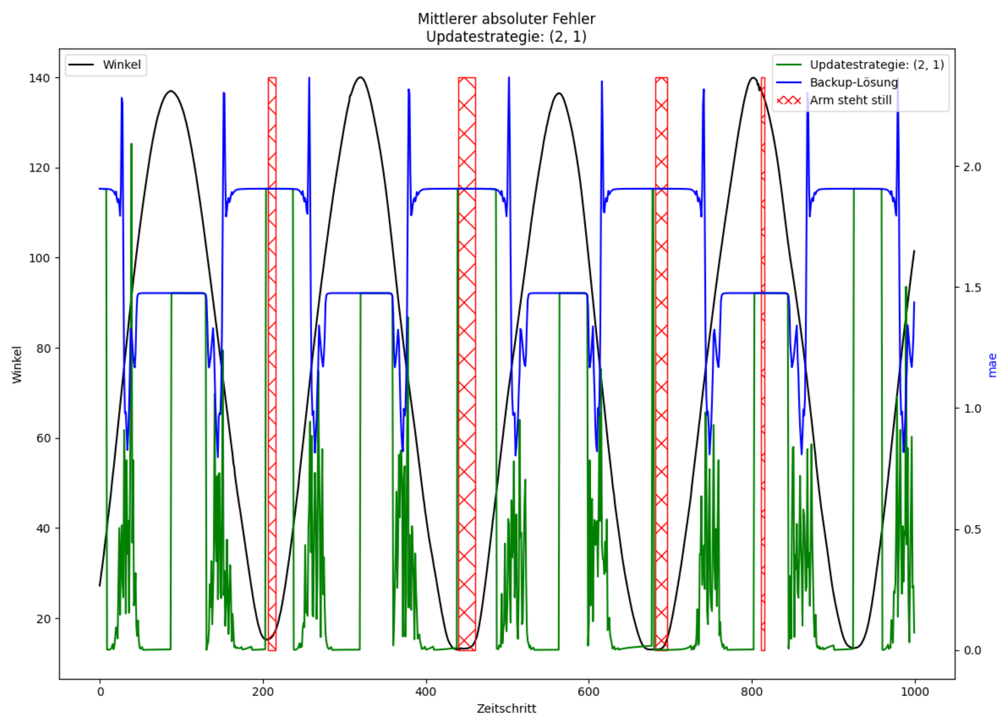

Abbildung A.11.: Qualität Updatestrategie (2, 1) (mit Löschen)

A.3. Implementierungseinstellung *ohne Löschen*

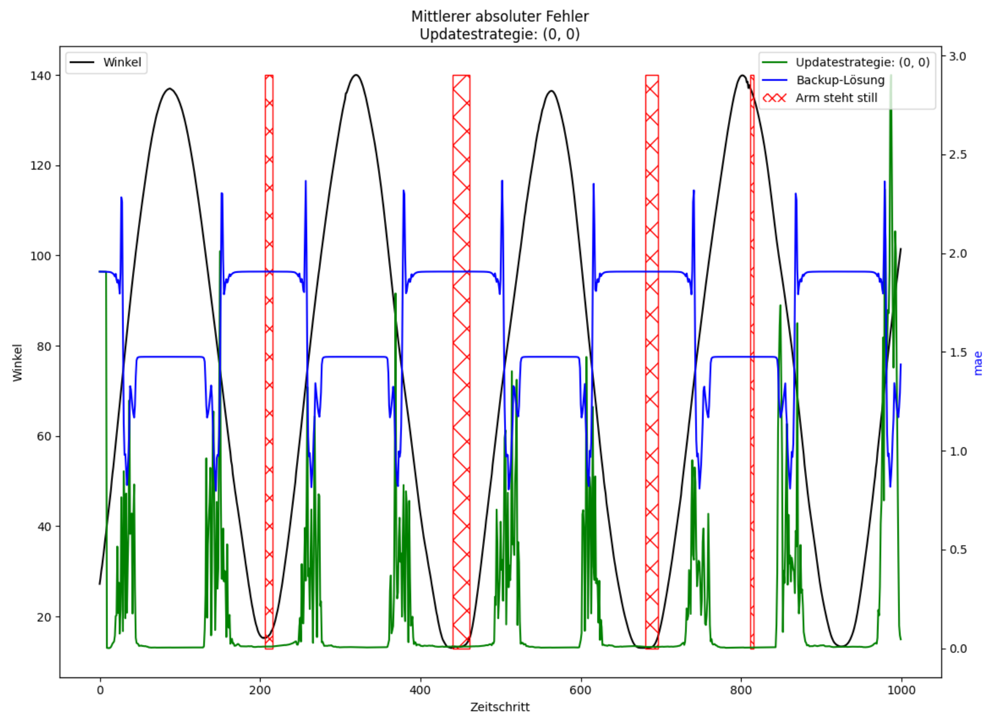

Abbildung A.12.: Qualität Updatestrategie (0, 0) (ohne Löschen)

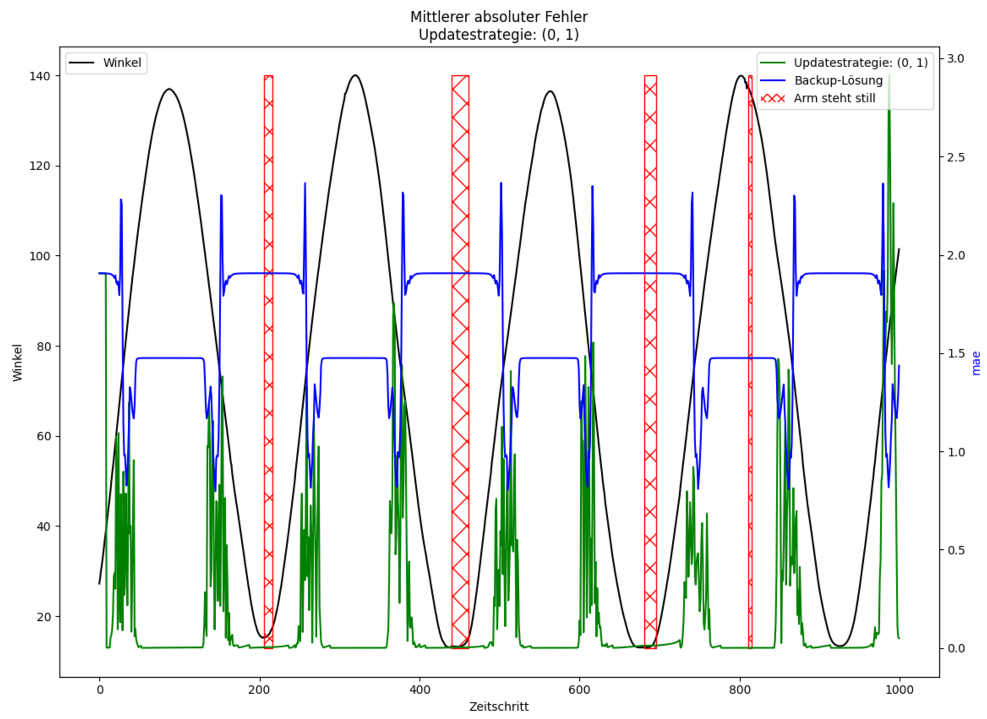

Abbildung A.13.: Qualität Updatestrategie (0, 1) (ohne Löschen)

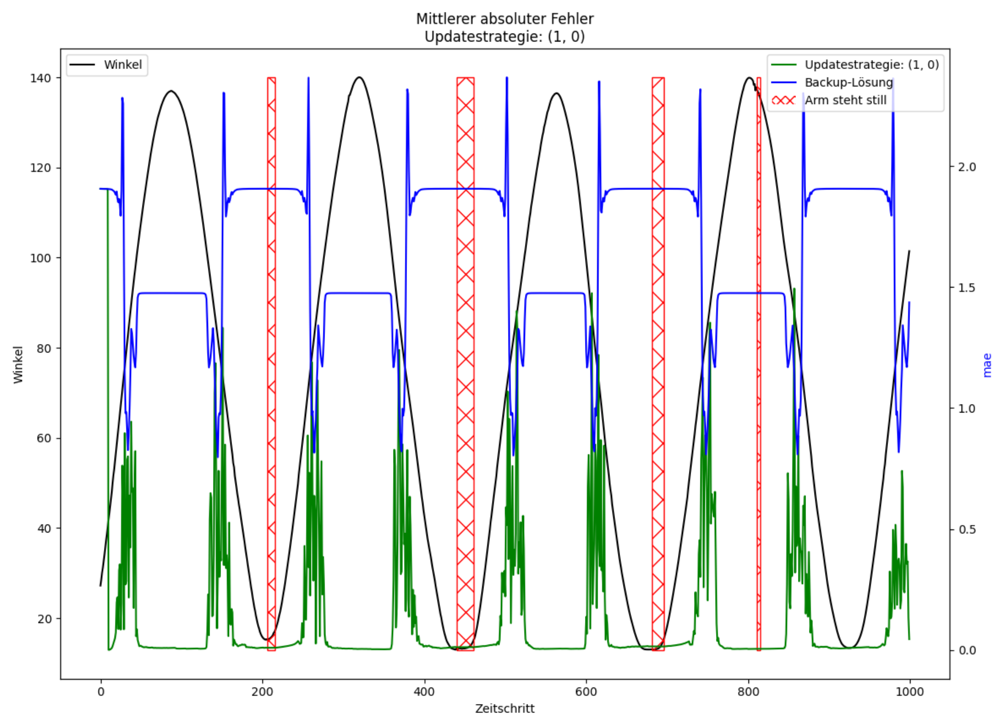

Abbildung A.14.: Qualität Updatestrategie (1, 0) (ohne Löschen)

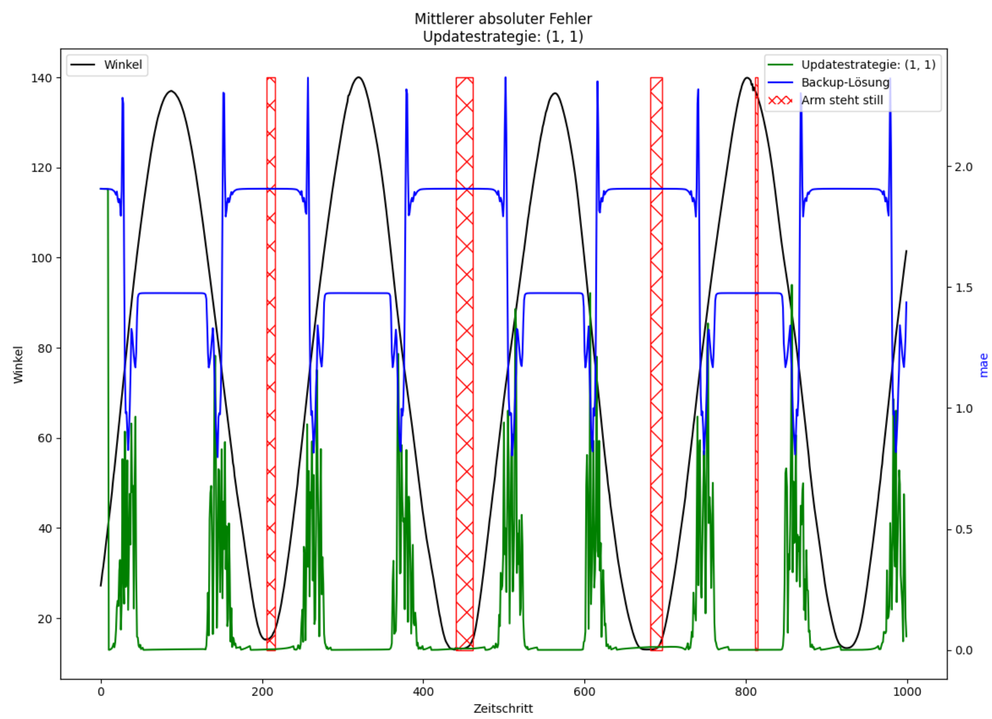

Abbildung A.15.: Qualität Updatestrategie (1, 1) (ohne Löschen)

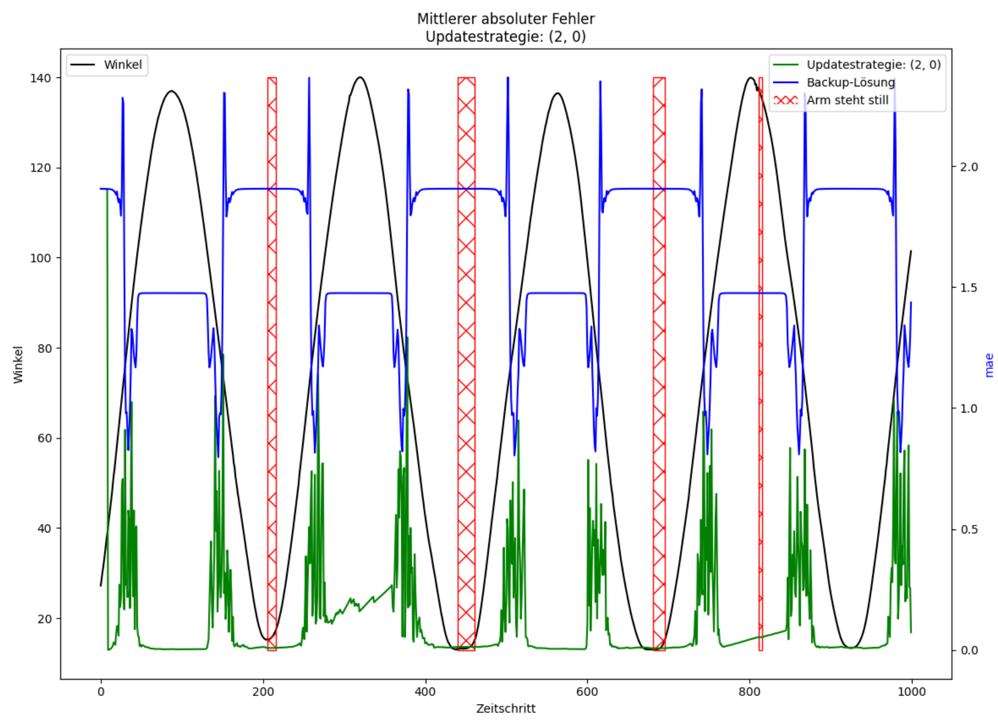

Abbildung A.16.: Qualität Updatestrategie (2, 0) (ohne Löschen)

A.4. 30 Zeitschritte bandbreitenoptimierter Input und Output *ohne Löschen*

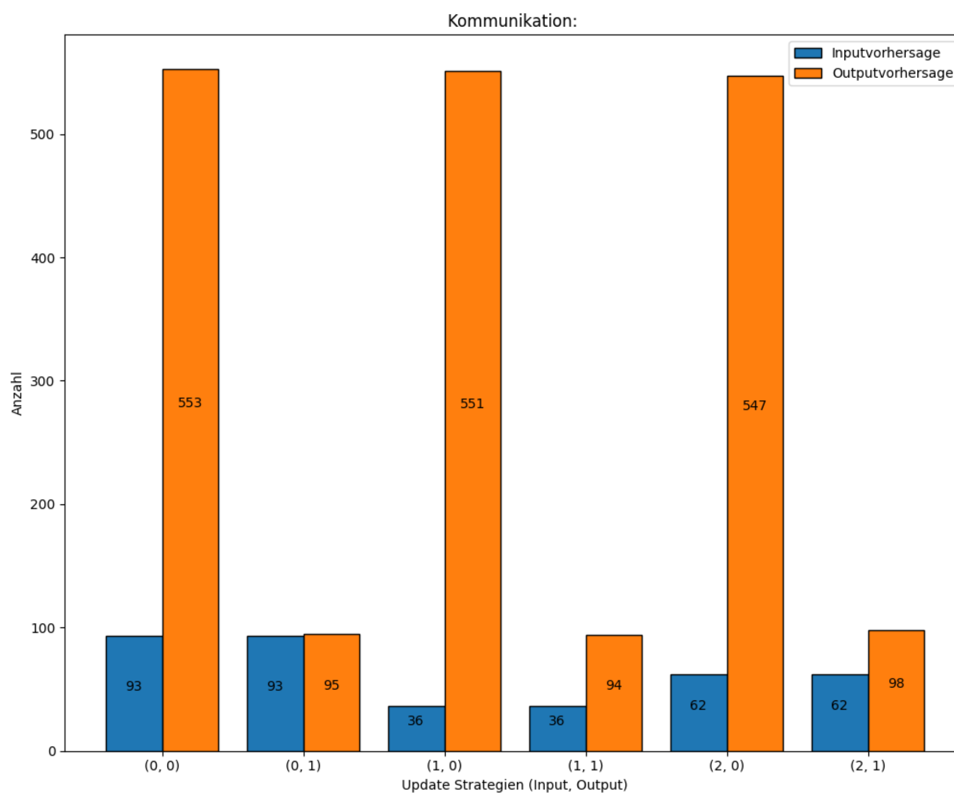

Abbildung A.17.: Kommunikation mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

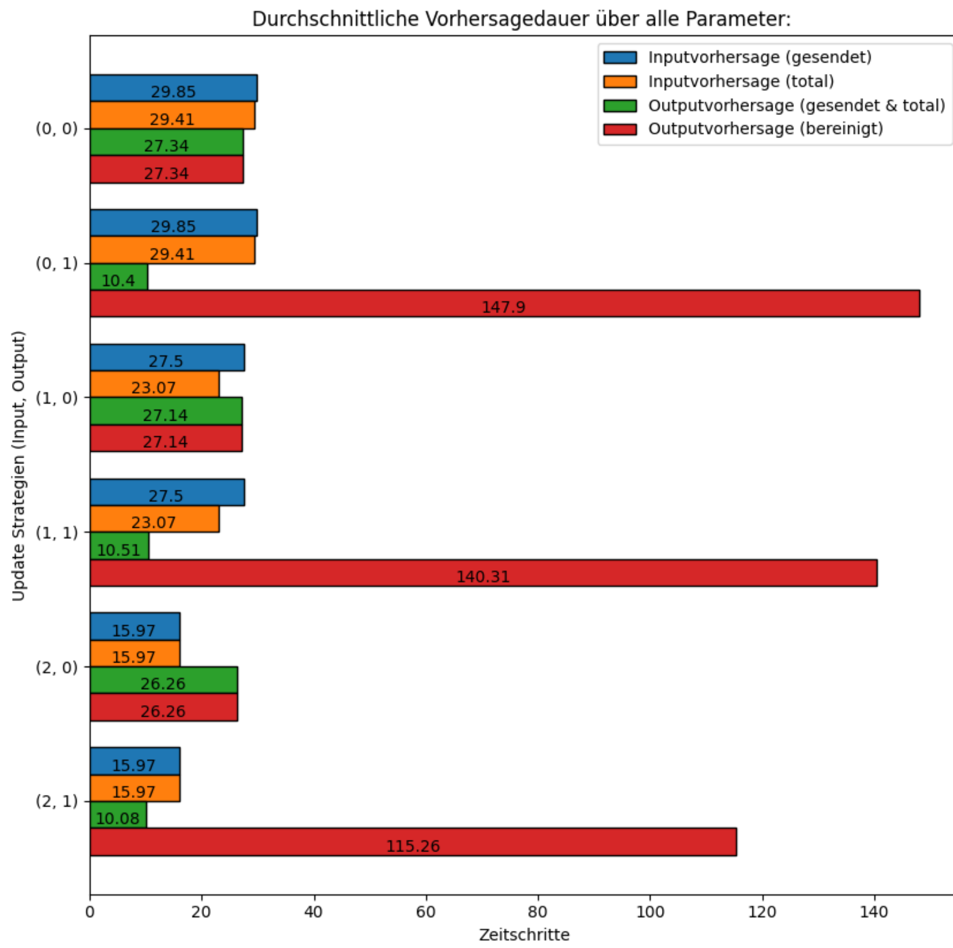

Abbildung A.18.: Vorhersagedauern mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

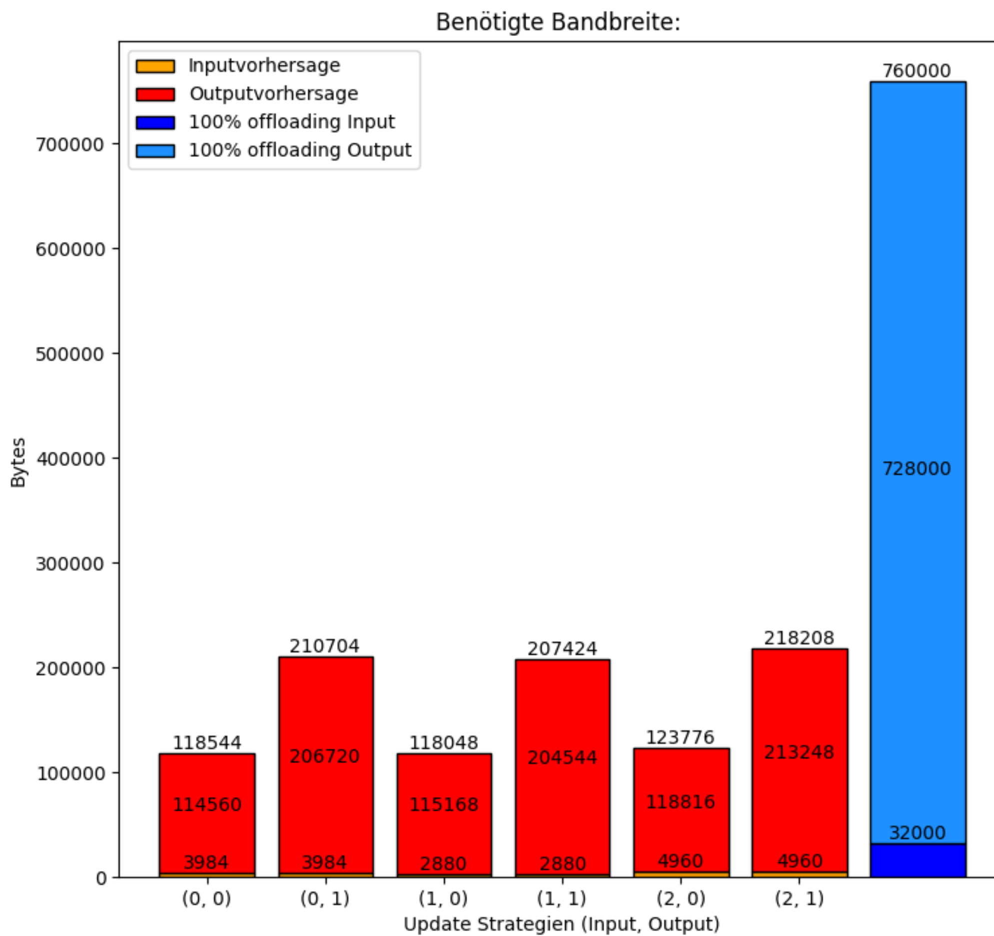

Abbildung A.19.: Bandbreite mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

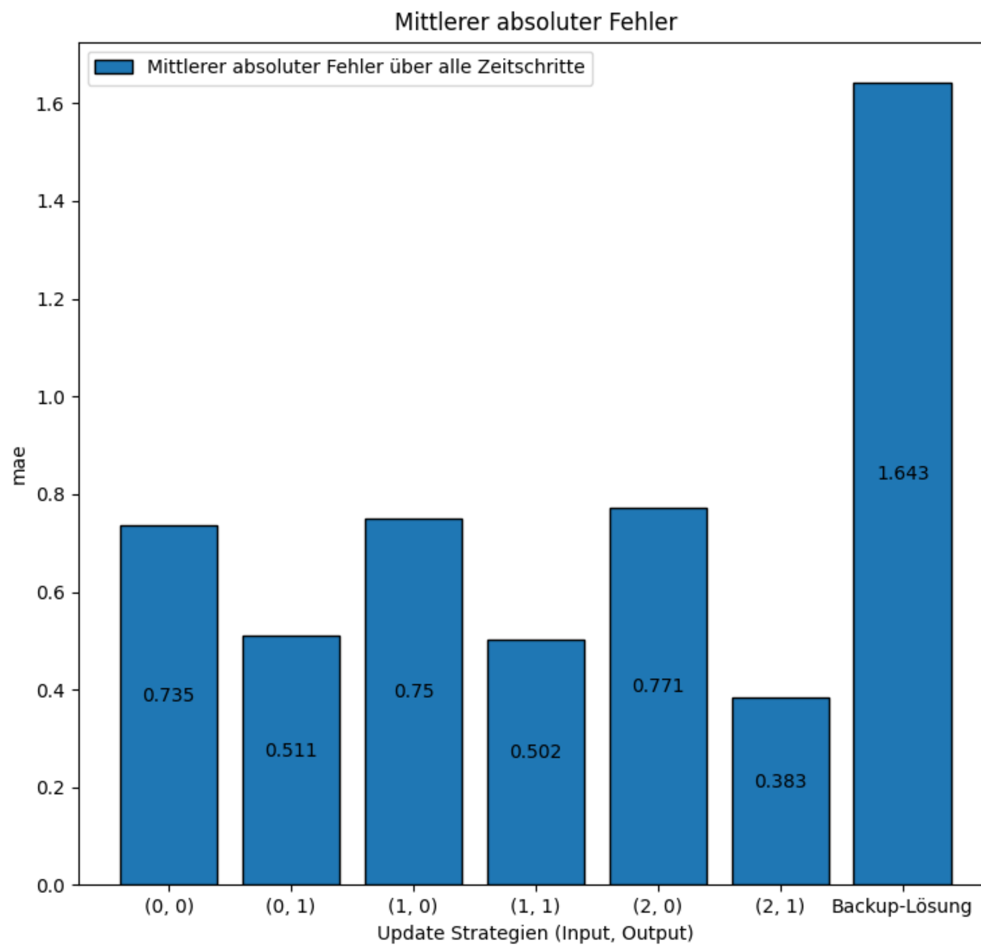

Abbildung A.20.: Gesamtqualität mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

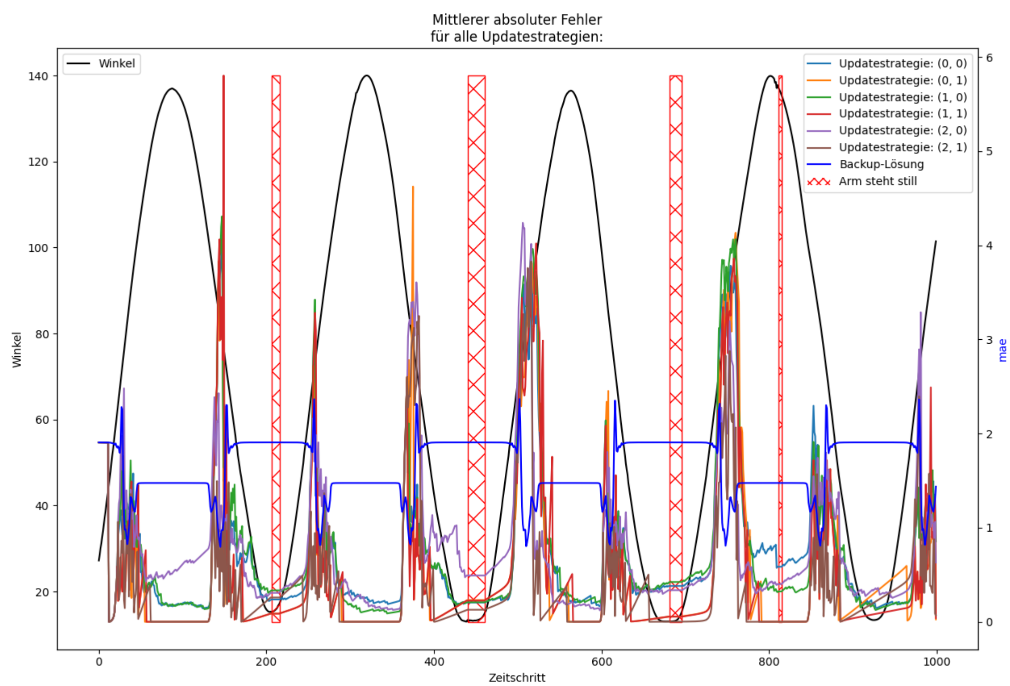

Abbildung A.21.: Qualität aller Updatestrategien in jedem Zeitschritt mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

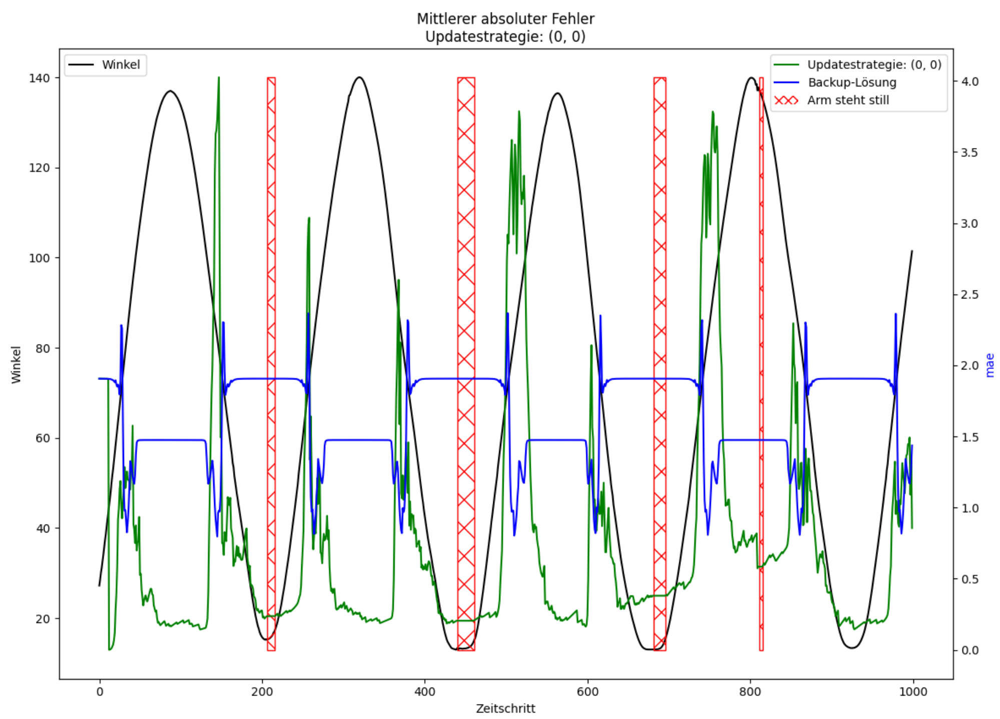

Abbildung A.22.: Qualität Updatestrategie (0, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

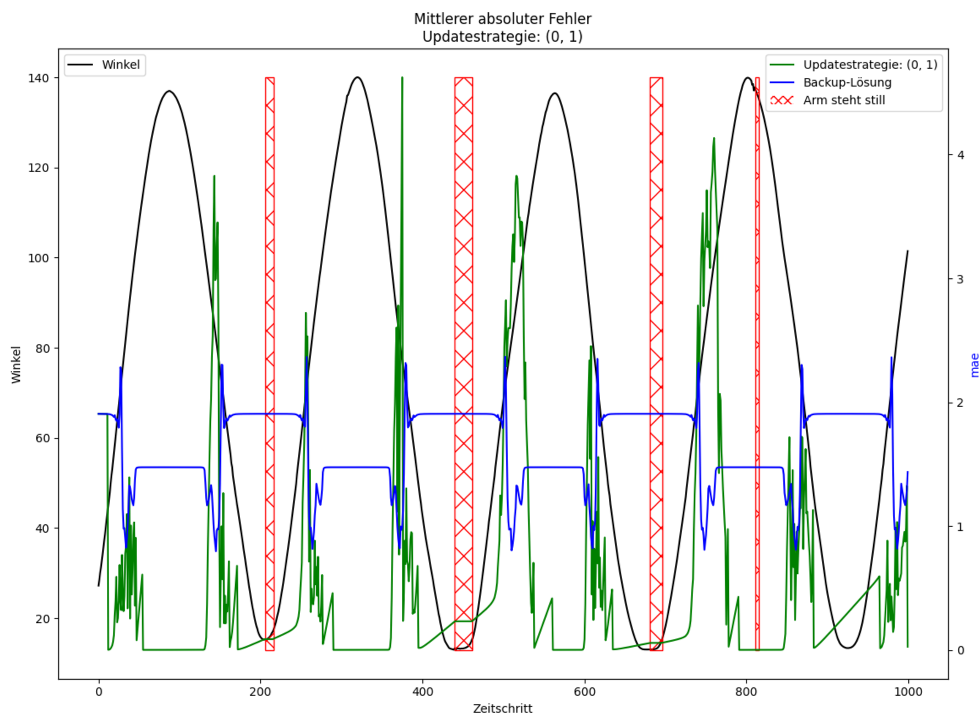

Abbildung A.23.: Qualität Updatestrategie (0, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

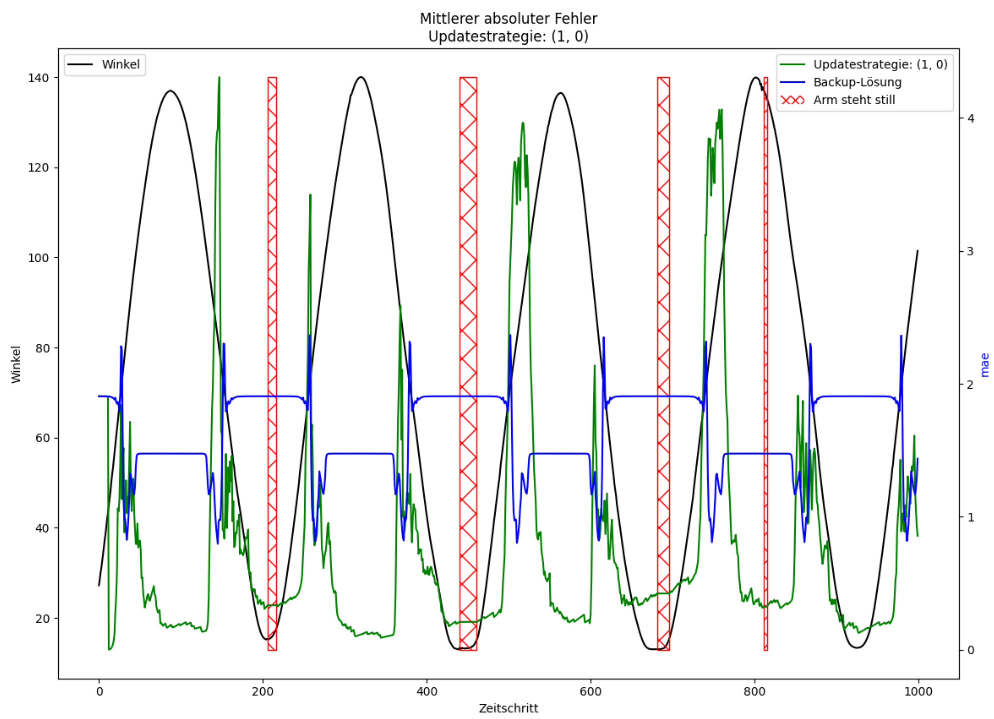

Abbildung A.24.: Qualität Updatestrategie (1, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

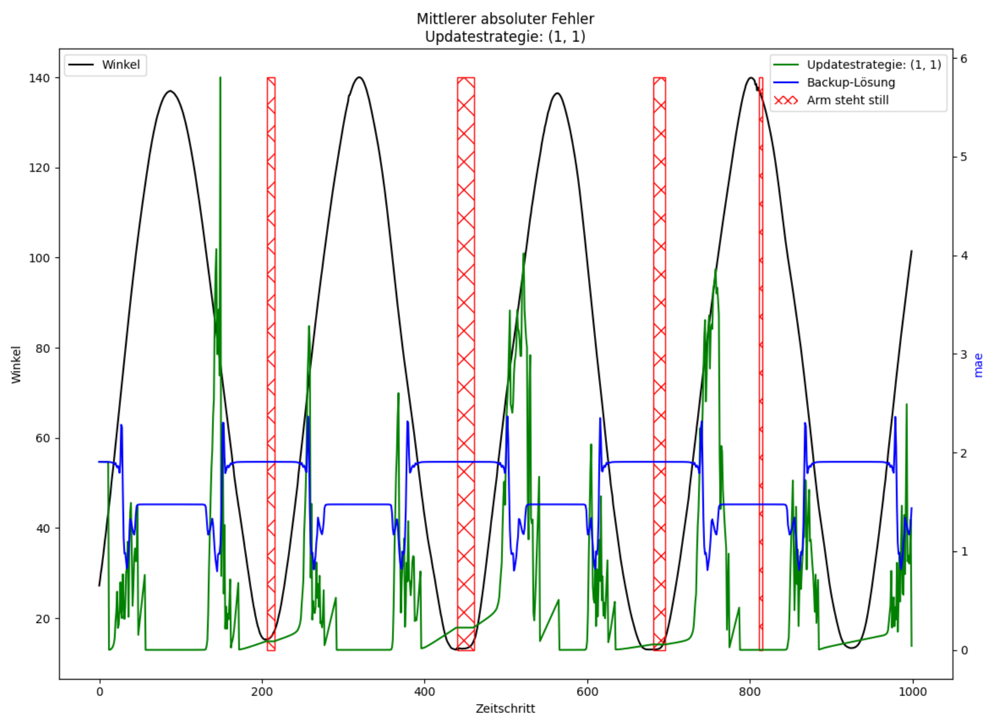

Abbildung A.25.: Qualität Updatestrategie (1, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

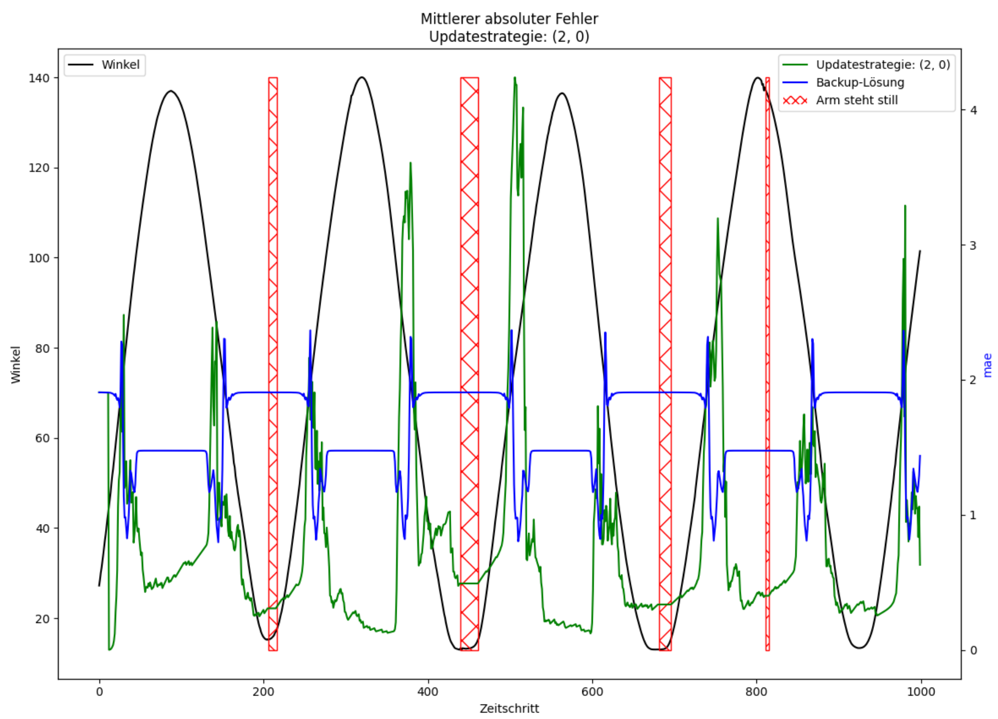

Abbildung A.26.: Qualität Updatestrategie (2, 0) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

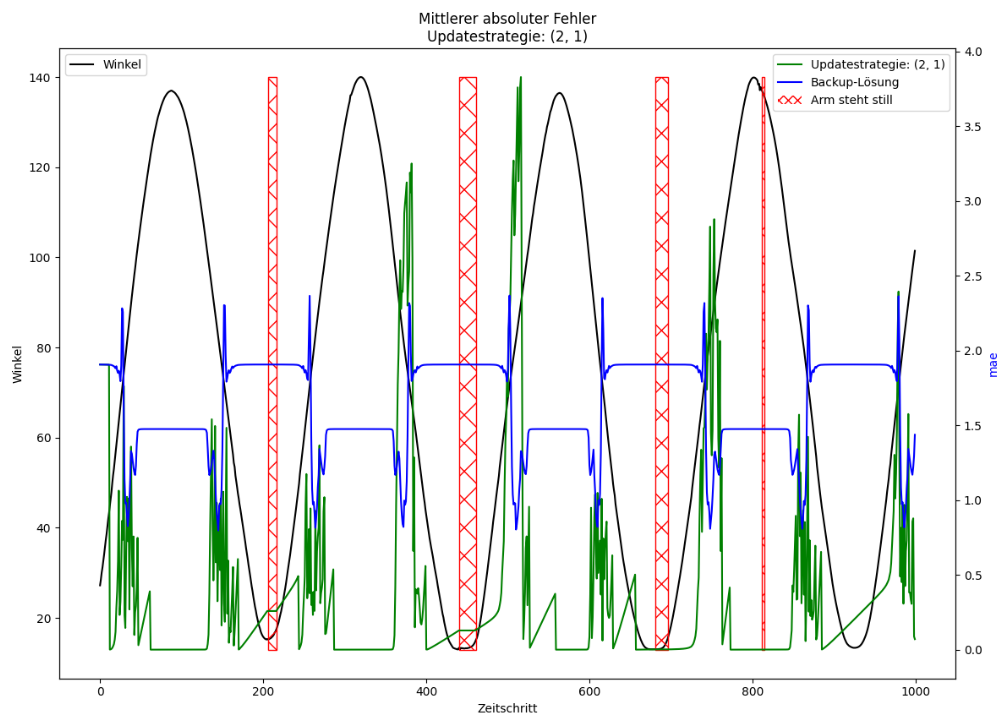

Abbildung A.27.: Qualität Updatestrategie (2, 1) mit 30 Zeitschritten bandbreitenoptimierter Input und Output (ohne Löschen)

A.5. 30 Zeitschritte bandbreitenoptimierter Output *ohne Löschen*

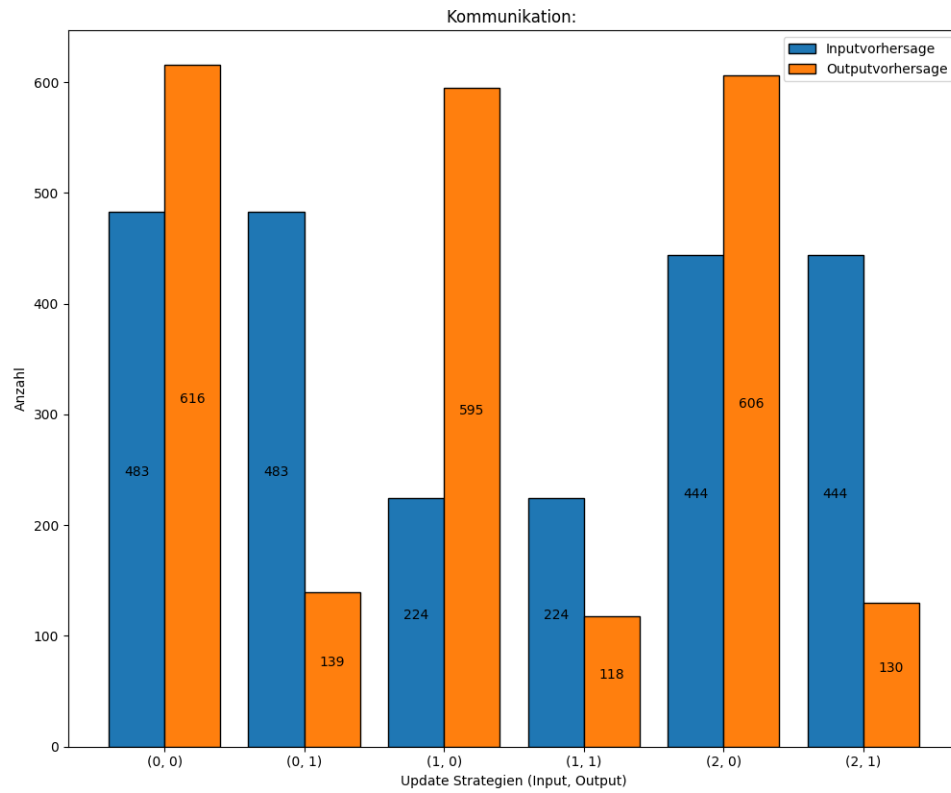

Abbildung A.28.: Kommunikation mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

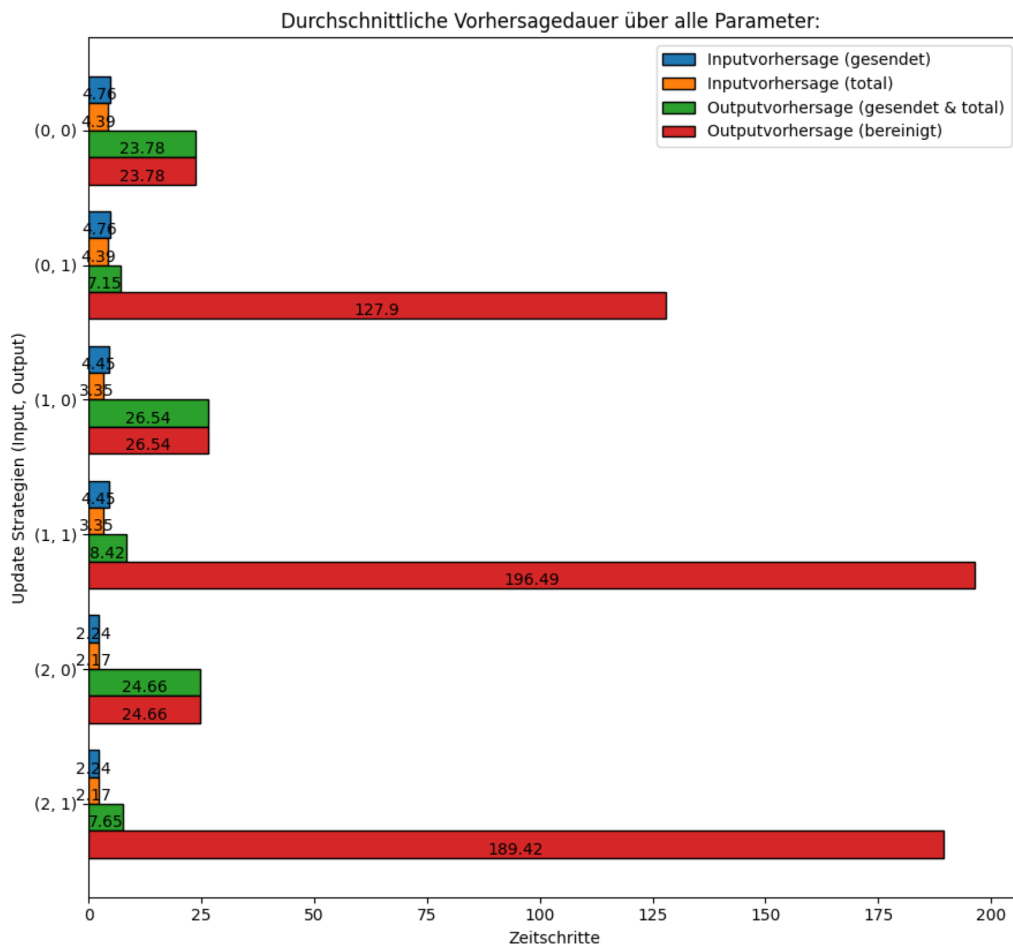

Abbildung A.29.: Vorhersagedauern mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

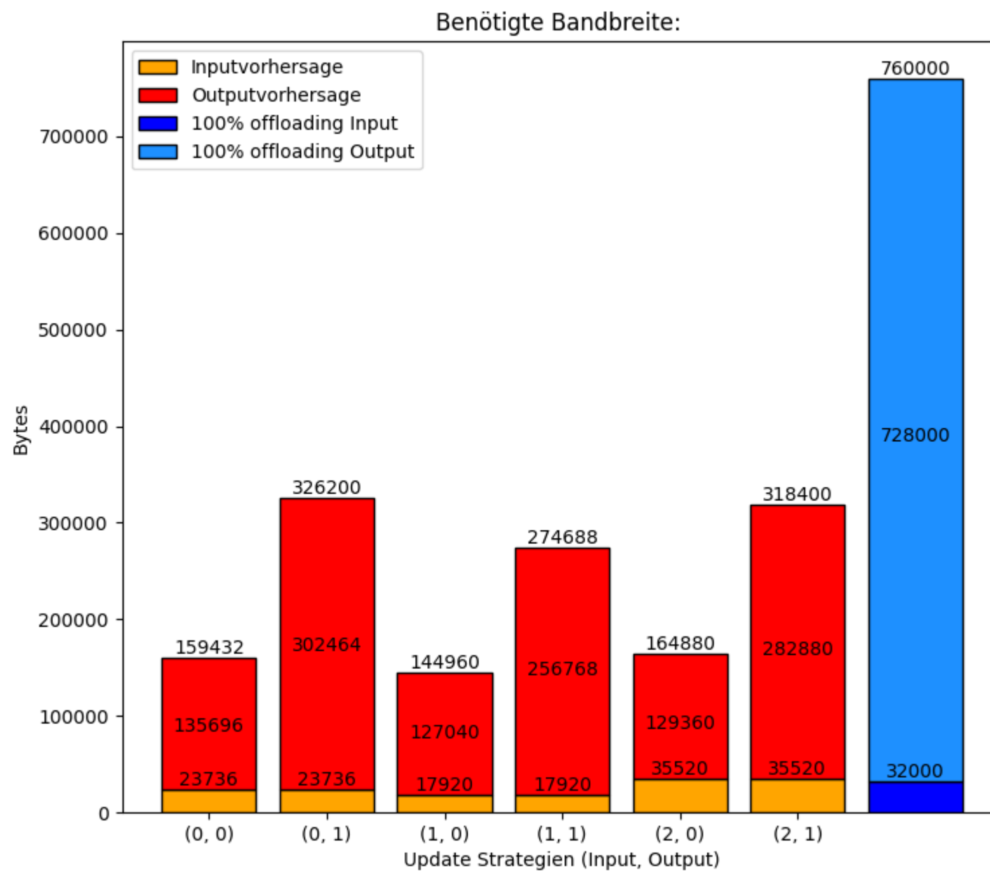

Abbildung A.30.: Bandbreite mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

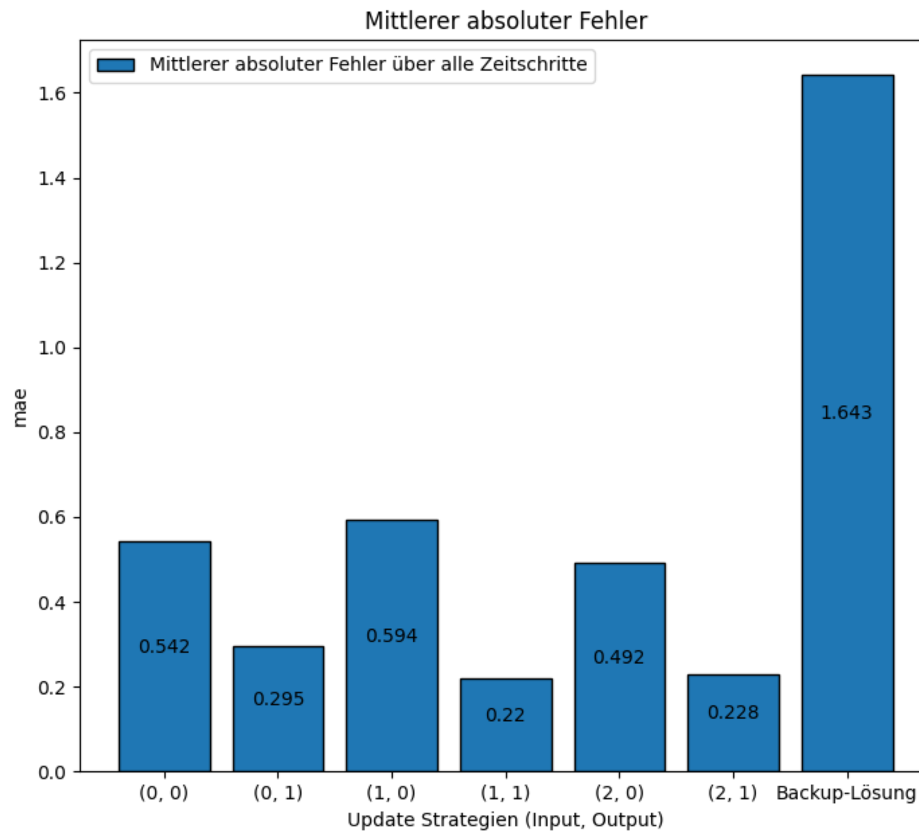

Abbildung A.31.: Gesamtqualität mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

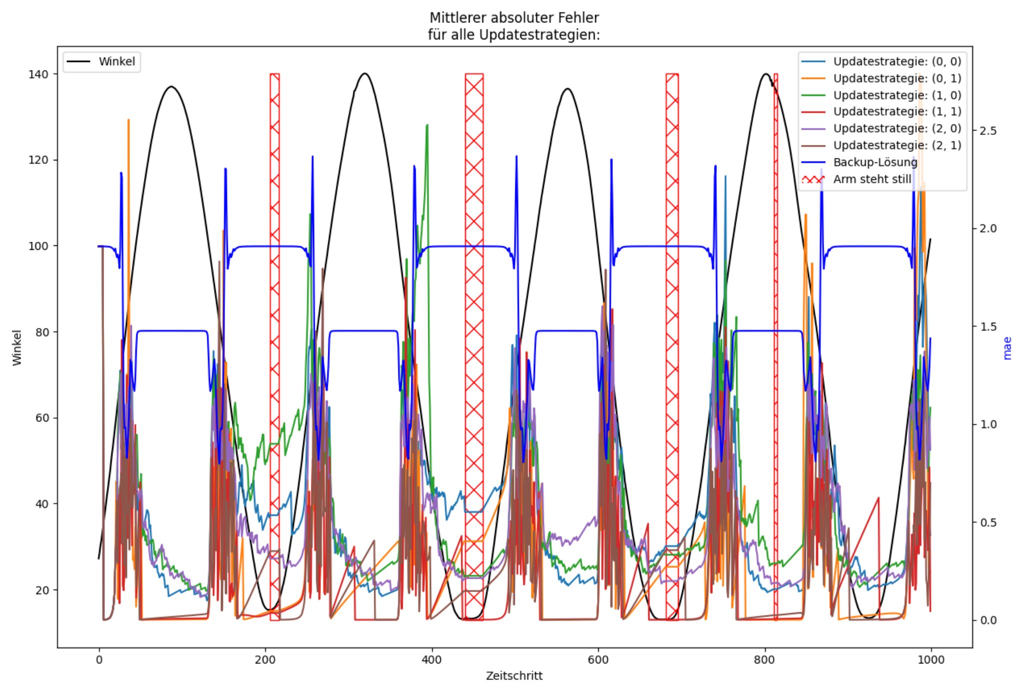

Abbildung A.32.: Qualität aller Updatestrategien in jedem Zeitschritt mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

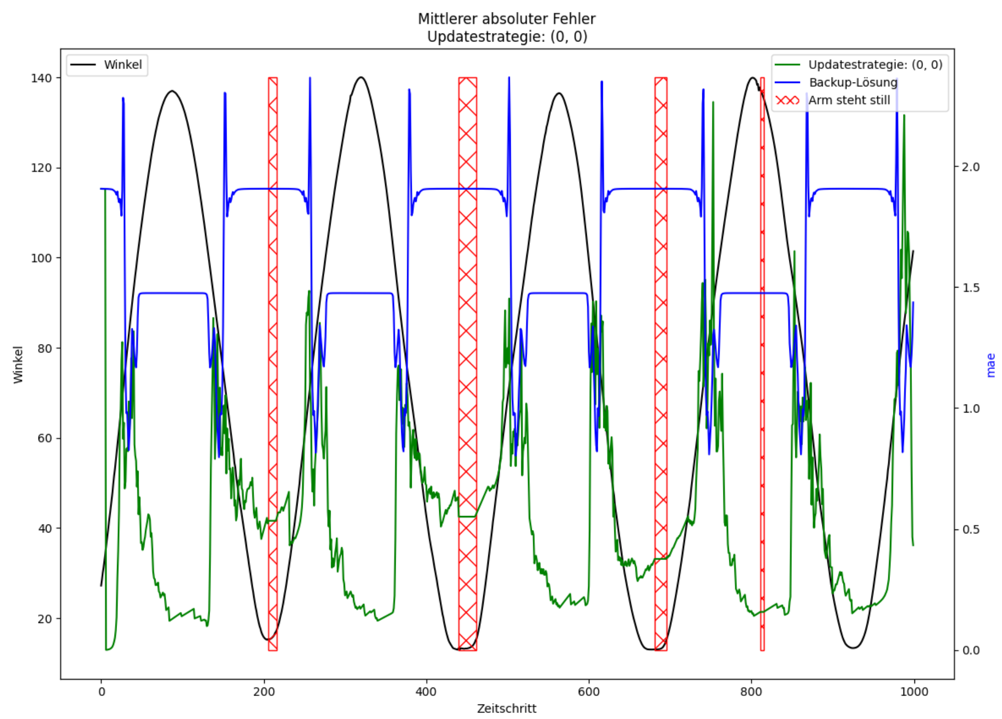

Abbildung A.33.: Qualität Updatestrategie (0, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

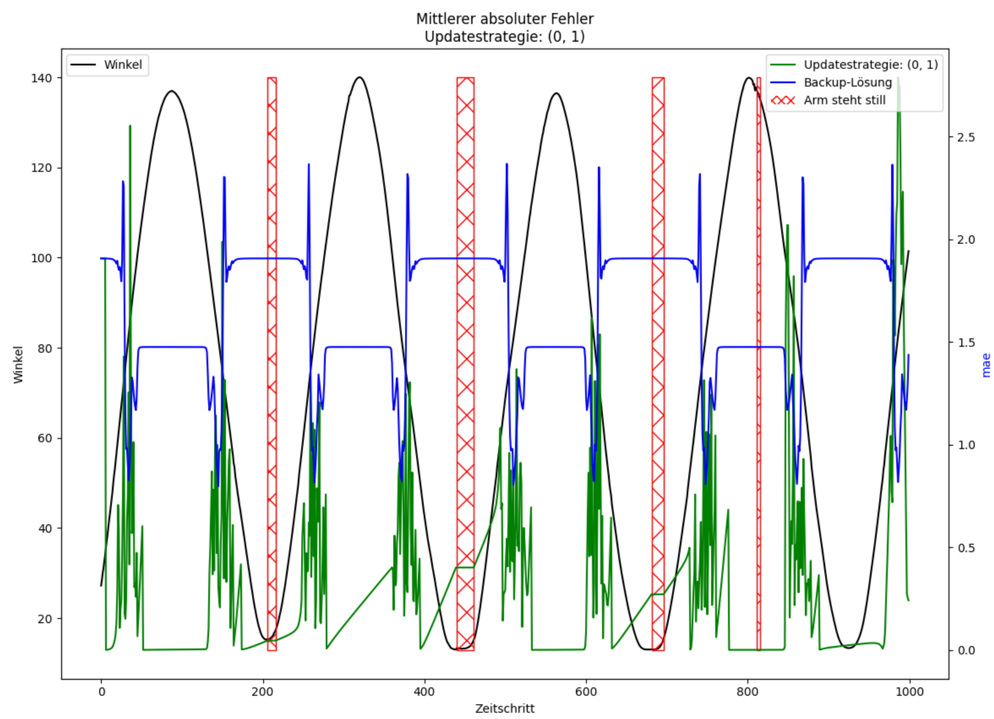

Abbildung A.34.: Qualität Updatestrategie (0, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

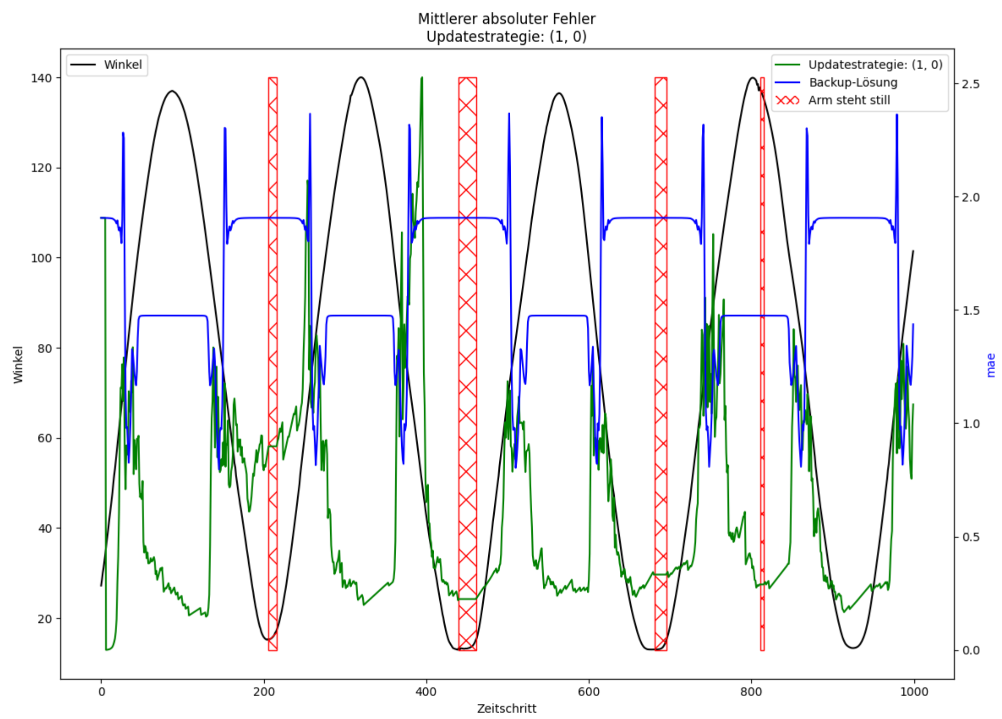

Abbildung A.35.: Qualität Updatestrategie (1, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

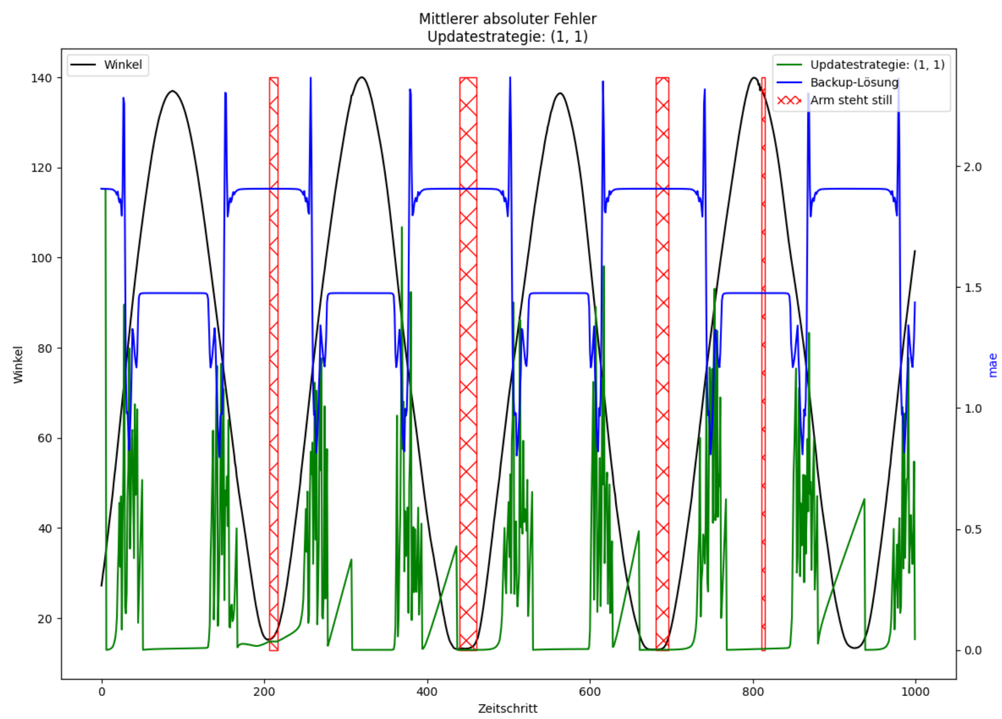

Abbildung A.36.: Qualität Updatestrategie (1, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

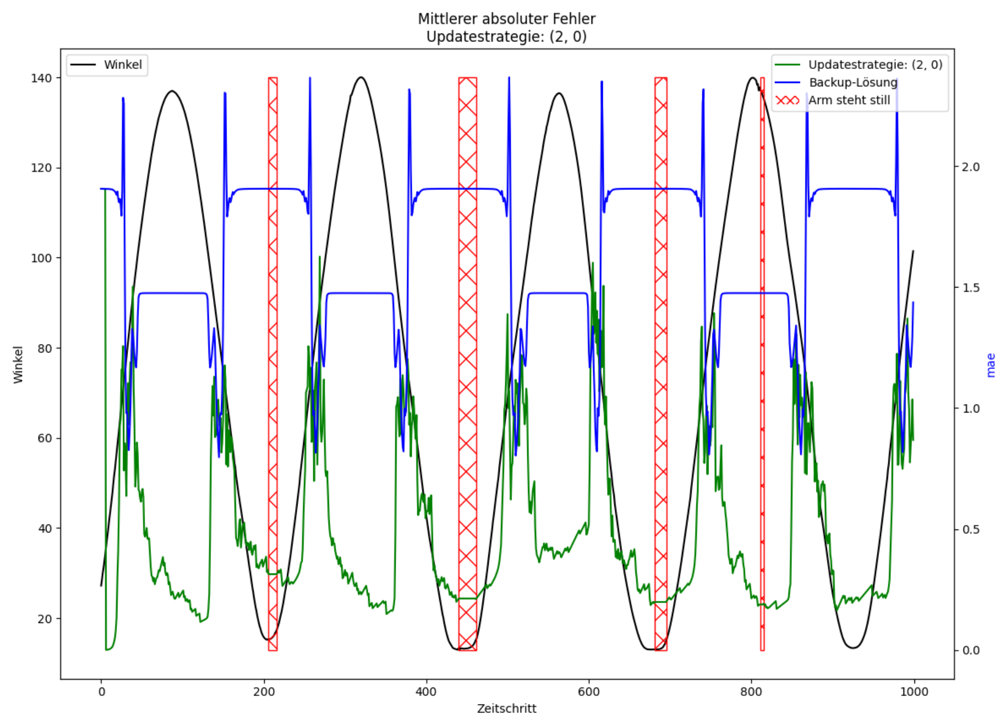

Abbildung A.37.: Qualität Updatestrategie (2, 0) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

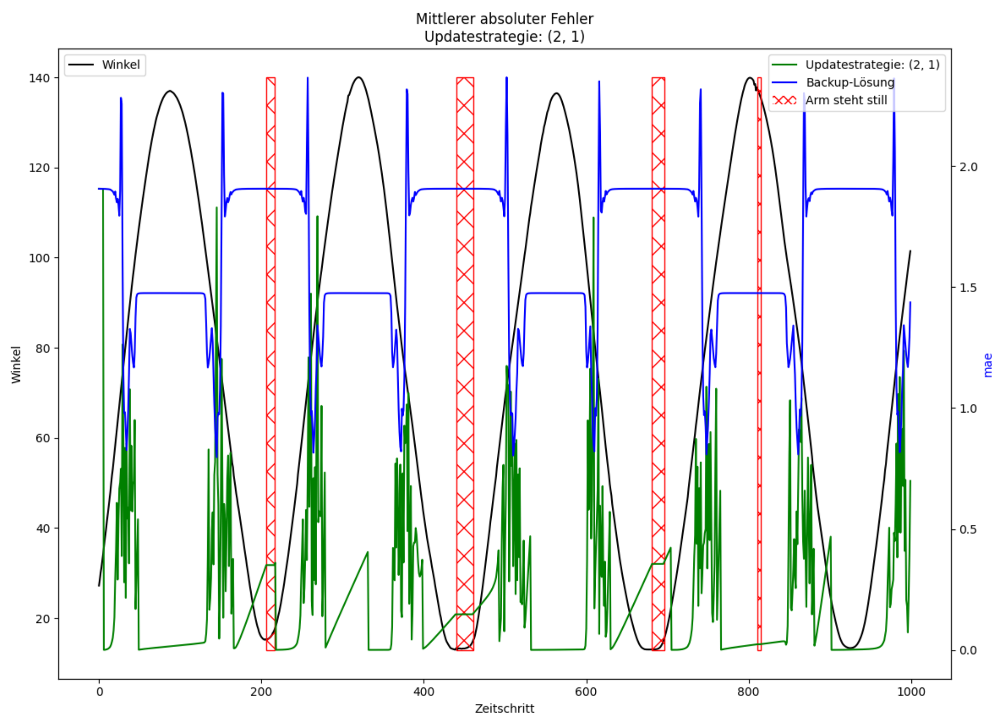

Abbildung A.38.: Qualität Updatestrategie (2, 1) mit 30 Zeitschritten bandbreitenoptimierter Output (ohne Löschen)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift