TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Simulation dynamischer Systeme mit neuronalen Netzen

ausgeführt am

Institut für
Analysis und Scientific Computing
TU Wien

unter der Anleitung von

Assistant Prof. Dr. techn. Andreas Körner

und

Dr. techn. Andreas Pfeffer

durch

Paul Strasser
Matrikelnummer: 11776791
Paniglgasse 24
1040 Wien

Wien, am 23. Oktober 2024

Danksagung

Ich möchte mich bei meinen Betreuern Andreas Pfeffer und Markus Gurtner bedanken, die mich mit ihren wertvollen Anregungen und konstruktiven Diskussionen bei der Ausarbeitung dieser Arbeit begleitet haben. Ebenso gilt mein besonderer Dank meiner Familie und meinen Freunden, die mich während meines gesamten Studiums stets ermutigt und unterstützt haben.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 23. Oktober 2024

Name des_r Autors_in

Abstract

This thesis explores the use of artificial neural networks to model dynamical systems. Recent improvements in machine learning have renewed interest in using these approaches for various modelling tasks. The use of artificial neural networks is promising since it allows the modelling of systems with limited knowledge of the underlying dynamics or in cases where internal system states can't be observed. This black-box modelling approach is especially suited for systems where a large amount of data is available, and could represent an alternative to state-of-the-art modelling techniques that are usually much more cost and time-intensive to develop.

In the first part of the thesis, we will introduce different network architectures, analyze their strengths and weaknesses give an overview of the mathematical foundations of neural networks and describe the algorithms that were used during the training process.

The next part is about the mathematical description of the types of dynamical systems we are considering and how their input-output relations are modelled. In this section, we will describe how neural networks are applied to model these systems.

In the last section, we will look at a real dynamical system and conduct tests of the neural network architectures we described by implementing them and comparing the performance of networks trained with either synthetic data or with data taken from measurements with well-established modelling approaches. In this context, we will describe how training data can be generated what techniques were used in preprocessing and what amount of data is sufficient for training.

As a final point, we will discuss our findings and use the obtained results to give recommendations for the types of network architectures and training strategies one should consider when attempting to model similar systems. This is especially important since there is no rigorous reasoning for which network architecture is best for a given task since in theory, the choice of network does not matter.

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit der Simulation von dynamischen Systemen unter Einsatz neuronaler Netze. Dafür wird eine Aufarbeitung der theoretischen Grundlagen vorgenommen, sowie Experimente zum Vergleich verschiedener Netzwerkarchitekturen und Trainingsmethoden angestellt. Weiters wird auf die relevanten Algorithmen und Details zur Implementierung eingegangen. Die Interpretation der Zustände eines dynamischen Systems als diskrete Folge erlaubt eine Black-Box-Modellierung, die eine einfache Umlegung der Eingangs-Ausgangs-Beziehung eines dynamischen Systems auf ein neuronales Netz ermöglicht.

Die Erstellung ausgefeilter Simulationsmodelle von realen Systemen ist oft sehr zeit- und kostenintensiv, andererseits nimmt die Menge und Qualität von Messdaten stetig zu, was datenbasierte Modellierungsansätze zunehmend attraktiver macht. Zusätzlich profitiert das maschinelle Lernen von fortlaufenden Verbesserungen der relevanten Algorithmen, immer höherer Rechenleistung besonders im Hinblick auf Parallelisierbarkeit von Berechnungen auf modernen Grafikkarten sowie der Verbreitung von benutzerfreundlicher Software-Infrastruktur.

Es ist im Allgemeinen unklar, welche Netzwerkarchitektur sich am besten eignet, um Problemstellungen dieser Art zu lösen. In dieser Arbeit wird auf Basis von experimentellen Resultaten untersucht, welche Netzwerkarchitektur, bei Bearbeitung eines ähnlichen Problems, eine gute Grundlage liefert. Um die Fähigkeiten der neuronalen Netze zu demonstrieren, werden diese mit synthetischen und mit Messdaten eines Pneumatikventils trainiert und anschließend mit professionellen Simulationsmodellen verglichen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

dt Ableitungsschätzer

LSTM Long-Short Term Memory neural network

LSTM-dt Long-Short Term Memory neural network zur Ableitungsschätzung

MLP MultiLayer Perceptron

MLP-dt MultiLayer Perceptron zur Ableitungsschätzung

MSE Mean Squared Error

OR Output Recurrence

OR-LSTM LSTM direkter Schätzer mit Output Recurrence

OR-LSTM-dt LSTM Ableitungsschätzer mit Output Recurrence

OR-MLP MLP direkter Schätzer mit Output Recurrence

OR-MLP-dt MLP Ableitungsschätzer mit Output Recurrence

OR-TCN TCN direkter Schätzer mit Output Recurrence

OR-TCN-dt TCN Ableitungsschätzer mit Output Recurrence

ResNet Residual Neural Network

RNN Recurrent Neural Network

TCN Temporal Convolutional Neural Network

TCN-dt Temporal Convolutional Neural Network zur Ableitungsschätzung

1 Einleitung

Die Simulation und Regelung von komplexen dynamischen Systemen ist, aufgrund ihrer enormen Bedeutung für praktische Anwendungen, ein sehr aktives Forschungsfeld. Die Entwicklung, Identifikation und Validierung von Simulationsmodellen ist oft zeit- und kostenintensiv, wodurch datenbasierte Ansätze zunehmend an Bedeutung gewinnen. Maschinelles Lernen hält durch die immer größer werdende Verfügbarkeit von Daten sowie Verbesserungen von Algorithmen, Rechenleistung und Verfügbarkeit von leicht zugänglicher Software auch in diese Forschungsgebiete Einzug. Ein sehr prominentes Beispiel der datenbasierten Methoden sind neuronale Netze, welche bereits für Simulation, [1, 2, 3], Identifikation [4], Sensitivitätsanalyse [5] und Regelung, [6, 7], dynamischer Systeme erforscht wurden. Zudem werden neue Netzwerkarchitekturen, die speziell für die Simulation von dynamischen Systemen geeignet sind, [8, 9, 10], untersucht. Wie in vielen Bereichen der künstlichen Intelligenz erleben neuronale Netze auch in diesem Zusammenhang eine neue Welle des Interesses, obwohl diese Konzepte bereits vor Jahrzehnten entwickelt wurden [11, 12].

Diese Arbeit befasst sich mit der Fragestellung, wie neuronale Netze zur Simulation dynamischer Systeme genutzt werden können und welche Netzwerkarchitekturen sich dafür eignen. Anhand von Experimenteren werden die effektivsten Netzwerkarchitekturen für diese Problemstellung identifiziert. Die Erklärungen der theoretischen Aspekte wurden auf Basis von Standardwerken, wie [13], erarbeitet. Die Beschreibung der Netzarchitekturen sowie der angewandten Methoden verlangen einige fachspezifische Begriffe. Zur leichteren Orientierung für Fachkundige, sowie zur besseren Einordnung in die Fachliteratur werden diese in dieser Arbeit bewusst in englischer Sprache formuliert.

Bei der Darstellung der drei Arten von neuronalen Netzen in Abschnitt 2.2, welche für die Experimente herangezogen werden, wird zum einen der Aufbau der Netze beschrieben. Zum anderen erfolgt eine Beschreibung des forward pass durch ein solches Netz. Dabei wird stets davon ausgegangen, dass die Netze für die Modellierung von Zeitreihen, siehe dazu Definition 2.1.1, verwendet werden. Die theoretische Grundlage für die Verwendung von neuronalen Netzen in diesem Kontext liefert das in Abschnitt 2.3 formulierte Approximationstheorem.

Die dynamischen Systeme, die wir in dieser Arbeit betrachten, haben eine Zustandsraumdarstellung der Form,

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{x} &= f(\boldsymbol{x}, \boldsymbol{u}), \\ \boldsymbol{y} &= g(\boldsymbol{x}, \boldsymbol{u}), \end{aligned}$$

dabei ist $\boldsymbol{x} = \boldsymbol{x}(t)$ der Systemzustand und $\boldsymbol{u} = \boldsymbol{u}(t)$ der Systemeingang. Der Systemausgang $\boldsymbol{y} = \boldsymbol{y}(t)$ wird in dieser Arbeit stets dem Systemzustand entsprechen, woraus

$g(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{x}$ folgt. Eine genaue Beschreibung erfolgt in Abschnitt 3.1. Für nichtlineare Funktionen f, lässt sich eine solche Differentialgleichung oftmals nicht lösen. Der von uns verfolgte Ansatz besteht nun darin, die Evolution des dynamischen Systems als eine diskrete Folge von Systemzuständen $\boldsymbol{x}(1), \boldsymbol{x}(2), \ldots, \boldsymbol{x}(T)$ zu betrachten und, mit einem neuronalen Netz, das nächste Folgenglied und damit die zukünftige Entwicklung des Systems in der Form

$$\hat{\boldsymbol{x}}(t+1) = \mathcal{N}(\boldsymbol{x}(t), \boldsymbol{u}(t)),$$

zu prädizieren. Dabei bezeichnet $\hat{\boldsymbol{x}}(t+1)$ eine Schätzung des Systemzustands zum Zeitpunkt $t+1$, mit einem neuronalen Netz $\mathcal{N}$. Eine schematische Darstellung dieses Vorgangs ist durch Abbildung 1.1 gegeben. Diese Black-Box-Modellierung hat den Vorteil, dass keinerlei Wissen über das System benötigt wird. Der Nachteil dabei ist, dass die Qualität des Modells stark von den Trainingsdaten abhängt. Dynamiken des Systems, die in den Trainingsdaten nicht, oder nicht ausreichend, vorhanden sind, können von dem neuronalen Netz nicht abgebildet werden.

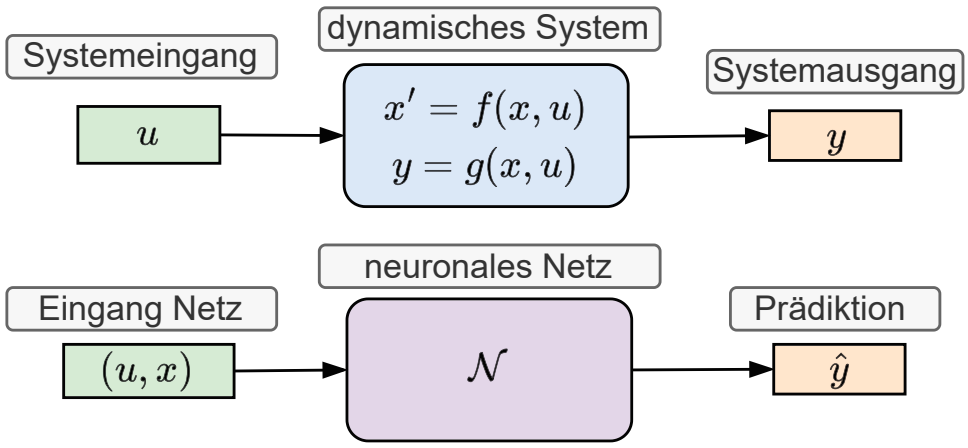

Figure 1.1: Überblicksdarstellung der Prädiktion des Ausgangs eines dynamischen Systems mithilfe eines neuronalen Netzes.

Nach der theoretischen Abhandlung befasst sich diese Arbeit auch mit der Anwendung. Der praktische Teil dieser Arbeit besteht darin, dass sämtliche Netzwerkarchitekturen in Python implementiert und anschließend mehrere Experimente durchgeführt wurden. Dafür wurde eine Optimierung diverser Hyperparameter vorgenommen und verschiedene mögliche Netzwerkarchitekturen verglichen. Schließlich wurden das MultiLayer Perceptron (MLP), als klassisches neuronales Netz, das Long-Short Term Memory neural network (LSTM) als Vertreter der Gruppe der Recurrent neural networks und das Temporal Convolutional Neural Network (TCN) als Form eines Convolutional neural networks, die sich besonders für das Arbeiten mit Zeitreihendaten eignet, ausgewählt, da sie repräsentativ für einen großen Anteil an Netzwerkarchitekturen sind. Beim Training der neuronalen Netze

wird zwischen zwei Methoden unterscheiden. Zum einen dem Teacher Forcing, einer Methode, bei der das Netzwerk darauf trainiert wird, aus Daten den jeweils nächsten Zeitschritt zu schätzen. Zum anderen der Methode der Output Recurrence, bei der im Gegensatz dazu, der Ausgang des Netzes bereits im Training als Eingang für die Berechnung des Systemzustands zum nächsten Zeitpunkt genutzt wird. Dabei lernt das Netzwerk also von seinen eigenen Prädiktionen weiterzurechnen, was bei der Auswertung des Netzes, unabhängig von der Methode, notwendig ist. Mit einem neuronalen Netz nur einen Zeitschritt in die Zukunft zu prädizieren, ist in manchen Szenarien zwar durchaus sinnvoll, der Fokus in dieser Arbeit liegt allerdings auf Simulation über längere Zeiträume.
Die Experimente stellen einen Vergleich zwischen den verschiedenen Netzwerkarchitekturen dar, wobei innerhalb jeder Architektur zwischen Netzen mit einer Skip-Connection, die aufgrund ihrer Ähnlichkeit zum Euler-Verfahren für gewöhnliche Differentialgleichungen, Ableitungsschätzer genannt werden und direkten Schätzern unterschieden wird. Außerdem wird noch zwischen den oben vorgestellten Lernmethoden, Teacher Forcing und Output Recurrence (OR), unterschieden. Aus der Analyse dieser Ergebnisse zeigt sich, dass es zwar keine eindeutig beste Architektur in allen Experimenten gibt, aber die Verwendung von Output Recurrence im Training erzielt stets die besseren Ergebnisse.

Der Aufbau der Arbeit gestaltet sich wie folgt. Der erste Abschnitt gibt einen Überblick über die mathematischen Grundlagen der Theorie der neuronalen Netze, wie sie in [13, 14, 15] beschrieben wird. Dabei wird der Fokus auf die Modellierung von Zeitreihen gelegt. Anschließend werden die Netzwerkarchitekturen der neuronalen Netze beschrieben, die im Zuge dieser Arbeit verglichen werden. Da es von jeder Familie von neuronalen Netzen diverse Unterarten gibt, die, allein schon aus Zeitgründen, nicht alle getestet werden können, wurden stattdessen einige repräsentative Architekturen ausgewählt. Danach beschreiben wir den Backpropagation Algorithmus in mehreren Formen, insbesondere für neuronale Netze, die mit Output Recurrence trainiert werden.

Im zweiten Teil der Arbeit werden dynamische Systeme beschrieben und ein Ansatz, wie solche Systeme mit neuronalen Netzen simuliert werden können, vorgestellt. Um unsere Methodik zu testen, führen wir drei Experimente durch. Zuerst simulieren wir ein motorisiertes Pendel und vergleichen die Prädiktionen unseres Modells mit den Lösungen eines Runge-Kutta-Verfahrens. Als Nächstes erzeugen wir mit einem MATLAB\SIMULINK-Modell eines Pneumatikventils der Firma FESTO SE & CO. KG. synthetische Trainingsdaten, mit denen wir verschiedene Arten von Netzwerken trainieren und die Genauigkeit ihrer Prädiktionen vergleichen. Danach, führen wir noch einen Vergleich zwischen dem MATLAB\SIMULINK-Modell und einem neuronalen Netz durch. Dafür wird das neuronale Netz mit Messdaten trainiert, die auf einem Prüfstand aufgenommenen wurden und anschließend mit dem Simulationsmodell verglichen. Dabei zeigt sich, dass das neuronale Netz eine vergleichbare Präzision wie das Simulationsmodell erzielt.

Abschließend erfolgt eine Zusammenfassung und Diskussion der wichtigsten Ergebnisse, sowie ein Ausblick, der mögliche Richtungen in die man in diesem Kontext weiter forschen könnte aufzeigt.

2 Neuronale Netze

2.1 Grundlagen und Funktionsweise neuronaler Netze

In diesem Kapitel wird die Funktion von neuronalen Netze erklärt und einige gängige Begriffe eingeführt. Neuronale Netze können für viele unterschiedliche Aufgaben verwendet werden. Zu den bekanntesten Anwendungen zählen die Bilderkennung, maschinelles Übersetzen, autonomes Fahren und Spracherkennung. Zwar finden sich in der Literatur auch biologische Motivationen für neuronale Netze, in der vorliegenden Arbeit liegt der Fokus allerdings auf der mathematischen Beschreibung.

Wir betrachten folgendes Problem. Es sei eine Menge von Daten $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$ gegeben. Weiters nehmen wir an, dass es eine Funktion $g : D \to W$ gibt, die $g(x_i) = y_i$ erfüllt. Unser Ziel ist es, eine Funktion f zu finden, die

$$\min_{\theta \in \Theta} \left\{ \sum_{i=1}^{n} L(f(x_i, \theta), y_i) \right\} \tag{2.1}$$

minimiert. Die Funktion $f : D \times \Theta \to W$ hängt von $x_i \in D$ und einer Parametermenge $\theta \in \Theta \subseteq \mathbb{R}^k$, mit $k \in \mathbb{N}$, ab. Die Wahl der Fehlerfunktion $L : W \times W \to \mathbb{R}$ zwischen $f(x_i, \theta)$ und y_i ist grundsätzlich frei wählbar. Meist wird, wie auch in dieser Arbeit, der quadratische Abstand verwendet. Da neuronale Netze Funktionen beliebig genau approximieren können, wie wir in Abschnitt 2.3.1 zeigen werden, sind sie für solche Probleme geeignet. Dadurch ist außerdem die Forderung, dass die Datenmenge D Teilmenge des Graphen von g ist, gerechtfertigt. Als nächstes, beschreiben wir den Aufbau eines einfachen neuronalen Netzes.

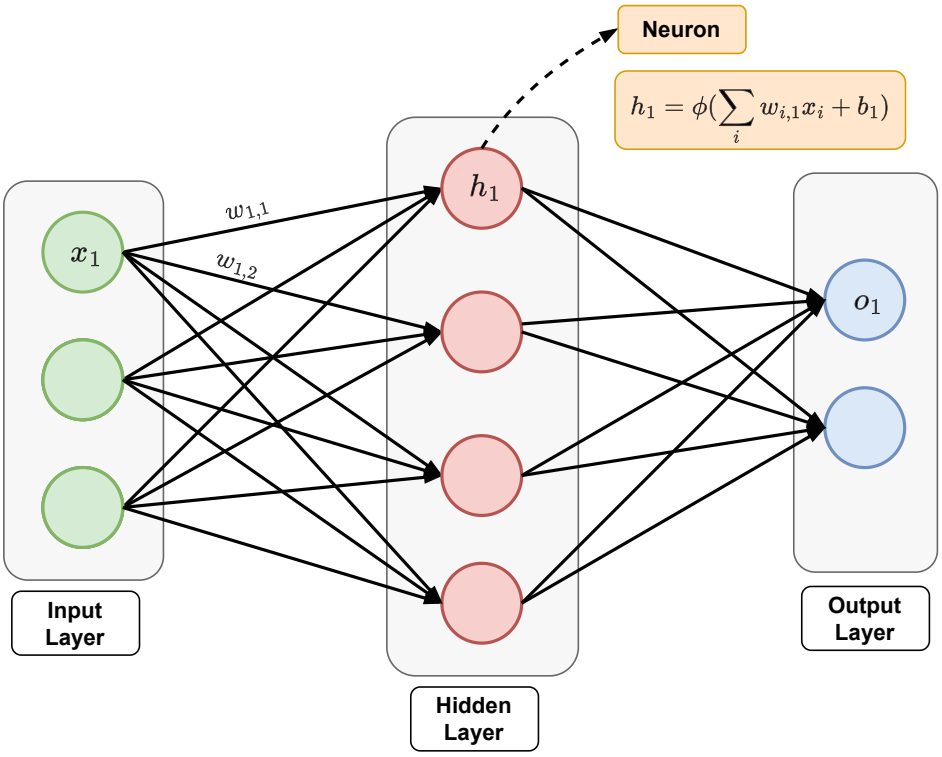

Abbildung 2.1: Darstellung eines neuronalen Netzes, mit *Input*, *Output* und einem hidden Layer.

Ein neuronales Netz besteht aus mehreren Layern, auf die jeweils eine affine Transformation, angewandt wird, gefolgt von einer nichtlinearen Aktivierungsfunktion. Durch diese sukzessive Aktivierung der einzelnen Zellen der Layer entsteht ein Fluss an Information durch das Netz. Die in dieser Arbeit wichtigsten Aktivierungsfunktionen werden in Abschnitt 2.1.2 diskutiert. Aktivierungsfunktionen sind essenziell für die Fähigkeit von neuronalen Netzen, Funktionen beliebig genau zu approximieren. Die Zellen, aus denen ein Layer besteht, heißen *Neuronen*. Die Anwendung der oben beschriebenen Abbildung auf die Neuronen eines Layers erzeugt eine Aktivierung in einem Neuron der darauffolgenden Layer. Ein neuronales Netz, in dem jedes Neuron in einem Layer auf diese Art mit allen Neuronen des vorherigen Layers verbunden ist, heißt *fully connected*. Ein einfaches Beispiel eines solchen Netzes ist in Abbildung 2.1 dargestellt. Der erste Layer wird als *Input Layer,* der letzte als *Output Layer* bezeichnet. Die Eingänge in neuronale Netz sind Vektoren von Eingangsdaten, deren Einträge die Werte der Neuronen im Input Layer sind. Die Werte können zum Beispiel die Pixelwerte eines Bilds, gemessene Zustände in einem physikalischen System, Worte in einem Satz, oder Aminosäuren in einem DNA-Segment darstellen, wobei die beiden letzteren zunächst auf passende numerische Werte abgebildet werden müssen. Die Werte der Neuronen im Output Layer müssen geeignet interpretiert werden. In Klassifikationsproblemen wird auf die Neuronen im Output Layer eine Softmax Aktivie-

rungsfunktion angewandt, die dem Eingangsdatenpunkt eine Wahrscheinlichkeit zuordnet in der jeweiligen Klasse zu sein. Bei der Approximation von Zeitreihen hingegen braucht es eine solche Funktion nicht. Die Größe des Output Layer und damit die Dimension des Netzausgangs, kann an die der Zeitreihe angepasst werden, wodurch der Netzausgang direkt als Prädiktion des nächsten Zeitschrittes verwendet werden kann. Abhängig von der Skalierung der Daten wird dabei auf eine Aktivierungsfunktion im Output Layer verzichtet.

Die übrigen Layer des Netzes werden als *hidden Layer* bezeichnet, wobei die Bezeichnung *hidden* sich darauf bezieht, dass diese Layer nicht von außen zugänglich sind, da sie weder Eingang noch Ausgang des Netzes darstellen und ihre Zustände durch die inneren Mechanismen des Netzes gesteuert werden, die sich im Allgemeinen kaum interpretieren lassen. Die Dimension der hidden Layern ist meistens deutlich höher, als die der Eingangsdaten. Diese erhöhte Dimension ermöglicht eine Extraktion einzelner Eigenschaften der Eingangsdaten im Zuge des Trainings des Netzes. Die Dimension der Layer bestimmt die Größe, also die Anzahl der Parameter, eines neuronalen Netzes, wobei die Dimension des Input Layer durch die Eingangsdaten, jene des Output Layer durch die Dimension der gewünschten Zielgrößen und die Dimension der hidden Layer frei wählbar ist. Dieser Logik folgend ist das Netz in Abbildung 2.1 eine Abbildung der Form

$$\mathcal{N}_\theta : \mathbb{R}^3 \to \mathbb{R}^4 \to \mathbb{R}^2,$$

mit $3 \cdot 4 + 4 + 4 \cdot 2 + 2 = 26$ Parametern.

Um diese Parameter passend zu wählen werden Gradientenverfahren, die in Abschnitt 2.3 genauer besprochen werden, angewandt. Diese Verfahren lösen, für eine $n-$elementige Datenmenge, das Minimierungsproblem

$$\min_{\theta \in \Theta} \left\{ \sum_{i=1}^{n} L(\mathcal{N}_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i) \right\}.$$

Verglichen mit (2.1), entspricht hierbei $\mathcal{N}_\theta(\boldsymbol{x}_i) = f(\boldsymbol{x}_i, \theta)$, für $\theta \in \Theta \subseteq \mathbb{R}^{26}$.

Definition 2.1.1 (Mean Squared Error)**.** In den meisten Anwendungen wird der Mean Squared Error (MSE) als Fehlerfunktion L verwendet. Für $\hat{\boldsymbol{y}}_i = \mathcal{N}(\boldsymbol{x}_i)$, $i = 1 \dots n$, ist dieser definiert als

$$MSE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{j=0}^{n} \|y_j - \hat{y}_j\|^2.$$

Der iterative Prozess, in dem die Parameter des neuronalen Netzes angepasst werden, nennt, man *Training*. Ein naiver Algorithmus zum Training eines neuronalen Netzes ist in Algorithmus 1 als Pseudocode beschrieben. Der Ablauf ist wie folgt. Zunächst wird das Netz erstellt. Dann wird das Netz für jeden Datenpunkt ausgewertet und der zugehörige Fehler berechnet. Die Auswertung bei einem Datenpunkt wird als *forward pass* bezeichnet. Anschließend werden die Gewichte, beispielsweise durch ein Gradientenverfahren, angepasst.

Algorithm 1: Neural Network Training

input : $i, h, o, epochs, data$
output: $\mathcal{N}_\theta$

```
begin
    # initialize the variables and create the network
    input_size = i
    hidden_size = h
    output_size = o
    θ_0 = initialize_weights(i, h, o)
    N_θ = create_neural_net(input_size, hidden_size, output_size, θ_0)
    for i = 1 : epochs do
        for x, y in data: do
            ŷ = N_θ(x) # forward pass
            error = error + L(y, ŷ) # calculate the errors for all datapoints
        end
        θ' = update_weights(error) # update the weights based on the errors
        N_θ = update_neural_net(θ') # update the network
    end
end
```

Bemerkung. Die Frage, wie genau die hidden Layer von dem Netzwerk genutzt werden und was ihre optimale Anzahl und Größe ist, ist eine offene Forschungsfrage. Diese Fragestellung hängt eng damit zusammen, wieso neuronale Netze allgemein so gut funktionieren. In diesem Kontext wollen wir noch die *manifold hypothesis* erwähnen, die die Vermutung anstellt, dass die meisten Mengen von hochdimensionalen Daten auf einer Mannigfaltigkeit mit geringerer Dimension liegen. Dadurch ist eventuell eine deutlich einfachere Repräsentation der Daten möglich, die in den hidden Layern des Netzwerks realisiert wird. Diese Vermutung wird beispielsweise in [16] und [17] geprüft, lässt sich jedoch noch nicht konkret beweisen.

2.1.1 Neuronale Netze zum Modellieren von Zeitreihen

Neuronale Netze eignen sich gut, um Daten, mit einer zeitlichen Komponente, zu verarbeiten. Dafür werden meist rekurrente Netzwerkarchitekturen verwendet. Eine umfangreichere Beschreibung dieser Thematik findet sich [18] und [15, Kapitel 7].

Definition 2.1.2 (Zeitreihen)**.** Wenn in dieser Arbeit von *Zeitreihen*, oder auch *Zeitreihendaten* gesprochen wird, sind $n-$elementige Mengen von Bildpunkten einer Funktion $z : \mathbb{R} \to \mathbb{R}^m$, der Form

$$\{z(t_i) \quad \text{für } i = 1 \dots n\},$$

gemeint. Wobei $t_i \in \mathbb{R}$, $t_i > t_j$ wenn $i > j$ und $z(t_i) \in \mathbb{R}^m$ für $m \in \mathbb{N}$ gilt. Da der Zeitpunkt t_i oft nur als Indizierung der Daten dient, geht man gelegentlich auf die Schreibweise

$$\{z_i \quad \text{für } i = 1 \dots T\},$$

mit $T \in \mathbb{N}$ über. In dieser Arbeit gilt zusätzlich meist, dass die Abtastrate, also $t_{i+1} - t_i$ konstant ist.

Die Aufgabenstellung, auf die wir uns fokussieren wollen, besteht darin, anhand von Zeitreihendaten $D = \{z(t_1), \ldots, z(t_n)\}$ zu lernen, den nächsten Zeitschritt $z(t_{n+1})$ zu prädizieren. Dafür wird ein neuronales Netz, wie oben beschrieben, trainiert und anschließend erhält man eine Prädiktion der Form

$$\hat{z}(t_{n+1}) = \mathcal{N}(z(t_n))$$

für den nächsten Zeitschritt t_{n+1}. Beispielsweise könnte $z(t_n)$ der Schlusskurs einer Aktie am Tag n sein und wir wollen daraus die Kursentwicklung am nächsten Tag evaluieren. Die Eingangs- und Ausgangsdaten des Netzes können auch unterschiedliche Dimension haben. Die Eingangsdaten hätten dann die Form $D = \{((x(t_1), y(t_1)), \ldots, (x(t_n), y(t_n))\}$ und der Ausgang $y(t_{n+1})$ würde durch ein neuronales Netz der Form

$$\hat{y}(t_{n+1}) = \mathcal{N}(x(t_n), y(t_n))$$

geschätzt werden.

Oft ist es nützlich nicht nur den vorherigen Zeitschritt, sondern ein Fenster an aufeinander folgenden Zeitschritten als Eingang für das Netzwerk zu verwenden. Das erlaubt dem Netzwerk leichter Zusammenhänge in den Eingangsdaten zu lernen. Für eine Fensterlänge k erhalten wir

$$\hat{y}(t_{n+1}) = \mathcal{N}(x(t_n), \ldots, x(t_{n-k})).$$

Die Länge des Fensters ist ein Parameter, der von der Beschaffenheit der Daten abhängt. Zum Beispiel wird bei stündlich gemessenen Blutzuckerwerten von Patienten, die Schätzung des nächsten Werts stark mit den Werten unmittelbar davor zusammenhängen. Will man hingegen in einer DNA-Sequenz bestehend aus tausenden Proteinen das Nächste in der Reihe vorhersagen, so ist es ratsam eine deutlich größere Fensterlänge zu wählen.

Beim Training unterscheiden wir zwischen zwei Ansätzen, dem *Teacher Forcing* und der *Output Recurrence*. Die Teacher Forcing Methode wurde bereits oben beschrieben. Im Training wird

$$\hat{z}(t_{i+1}) = \mathcal{N}(z(t_i)), \quad i = 1, \ldots, n-1,$$

berechnet. Im Gegensatz dazu wird beim Training mit Output Recurrence der letzte Ausgang des Netzes als Eingang zum nächsten Zeitpunkt verwendet. Wir erhalten

$$\begin{aligned} \hat{z}(t_2) &= \mathcal{N}(z(t_1)), \\ \hat{z}(t_{i+1}) &= \mathcal{N}(\hat{z}(t_i)), \quad i = 2, \ldots, n-1. \end{aligned}$$

Bei beiden Methoden wird jeweils der Fehler zum tatsächlichen Wert $L(z(t_{i+1}), \hat{z}(t_{i+1}))$ berechnet. Auf die Unterschiede der beiden Methoden im Hinblick auf Lernen und Genauigkeit werden wir später eingehen.

Bemerkung. Die beiden vorgestellten Methoden betreffen lediglich den Lernprozess. Wenn man mit derart trainierten Netzen mehrere Schritte in die Zukunft prädizieren möchte, gibt es keine Alternative als den vorherigen Ausgang des Netzwerks als nächsten Eingang zu nutzen.

2.1.2 Aktivierungsfunktionen

Aktivierungsfunktionen sind aufgrund ihrer Nichtlinearität ein essenzieller Teil von allen neuronalen Netzen. Die Wahl der Aktivierungsfunktion ist ein Hyperparameter, der von der Problemstellung abhängt und beim Erstellen des neuronalen Netzes gewählt wird. Die Wahl der Aktivierungsfunktionen kann Teil der Optimierung sein, allerdings ist bei vielen Netzwerkarchitekturen eine, in den meisten Fällen, beste Wahl der Aktivierungsfunktion bekannt. Die wichtigsten Aktivierungsfunktionen und ihre Ableitungen sind

- die rectified linear unit (ReLU) Funktion,

$$\begin{aligned} \mathrm{ReLU}(x) &= \begin{cases} 0, & \text{for } x < 0, \\ x, & \text{for } x \geq 0, \end{cases} \\ \mathrm{ReLU}'(x) &= \begin{cases} 0, & \text{for } x < 0, \\ 1, & \text{for } x \geq 0, \end{cases} \end{aligned} \tag{2.2}$$

- die Sigmoid Funktion,

$$\begin{aligned} \sigma(x) &= \frac{1}{1+\mathrm{e}^{-x}}, \\ \sigma'(x) &= \frac{\mathrm{e}^{-x}}{(1+\mathrm{e}^{-x})^2} = \sigma(1-\sigma), \text{ sowie} \end{aligned} \tag{2.3}$$

- der Tangens hyperbolicus,

$$\begin{aligned} \tanh(x) &= \frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}}, \\ \tanh'(x) &= 1-\tanh(x)^2. \end{aligned} \tag{2.4}$$

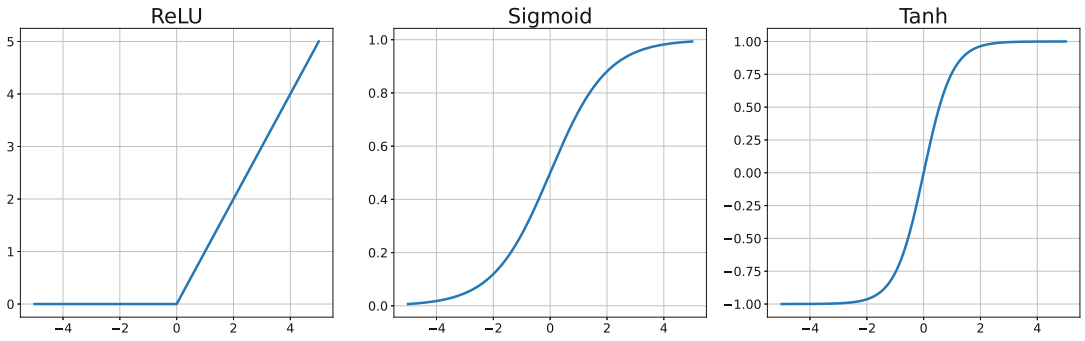

Abbildung 2.2: Darstellung drei wichtiger Aktivierungsfunktionen: rectified linear Unit *ReLU*, Sigmoid σ und Tangens Hyperbolicus tanh.

Die oben vorgestellten Aktivierungsfunktionen sind in Abbildung 2.2 dargestellt. An der Grafik ist erkennbar, dass die ReLU Funktion, als einzige der drei Funktionen, nicht beschränkt und bei $x = 0$ nicht differenzierbar ist. Anzumerken ist, dass diese Liste nur eine Auswahl zeigt und bei weitem nicht vollständig ist. Aktivierungsfunktionen, wie SoftMax, die bei Klassifikationsproblemen angewendet werden, wurden weggelassen. Wenn in dieser Arbeit von einer Aktivierungsfunktion ϕ gesprochen wird, nehmen wir stets $\phi \in \{\text{ReLU}, \sigma, \tanh\}$ an.

2.2 Netzwerkarchitekturen

In diesem Abschnitt stellen wir die verschiedenen Netzwerkarchitekturen vor, die in dieser Arbeit verwendet werden. Um ein möglichst aussagekräftige Ergebnisse, für die große Anzahl an unterschiedlichen Netzwerkarchitekturen zu erhalten, wurden eine repräsentative Auswahl getroffen. Das ist sinnvoll, da die meisten neuronalen Netze der Kategorie der rekurrenten oder der *Feedforward* Netze angehören und zusätzlich über teils anwendungsspezifische Adaptionen verfügen. Zusätzlich wurde ein Netz von der Gruppe der convolutional Netze ausgewählt, da diese einen wichtigen Teil der Feedforward Netze bilden.

2.2.1 MLP Architektur

Das MLP zählt zur Kategorie der fully connected Feedforward Netze und ist die einfachste Struktur, die ein neuronales Netz haben kann. Auf einen Eingangsvektor wird eine affine Transformation, gefolgt von einer Aktivierungsfunktion angewandt. Dies wird für jeden Layer wiederholt, wobei der jeweils vorherige Layer der neue Eingangsvektor ist. Im letzten Layer werden die Daten auf die, für den Ausgang gewünschte Dimension gebracht. Durch diesen Aufbau ist der Wert jedes Neurons von den Werten aller Neuronen, die in Layern davor liegen, abhängig.

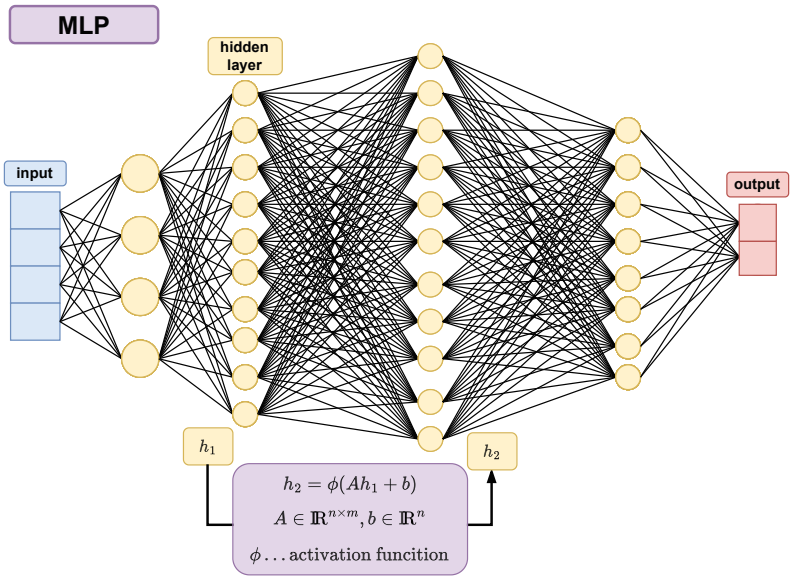

Abbildung 2.3: Ein Multilayer perceptron MLP mit einer Beschreibung der Transformation zwischen zwei hidden Layern.

Der Ausgang, für einen Eingangsvektor $\boldsymbol{x}$, wir durch

$$\boldsymbol{y} = T_k(T_{k-1} \dots (T_1(\boldsymbol{x}))), \quad T_i(\boldsymbol{x}) = \phi_i(A_i \boldsymbol{x}_i + \boldsymbol{b}_i), \quad \forall i \in \{1...k\}$$

berechnet. Hierbei sind $A_i \in \mathbb{R}^{n_i \times m_i}$ und $\boldsymbol{b}_i \in \mathbb{R}^{n_i}$ wobei m_i die Dimension des vorherigen Layers ist und n_i, mit $n_i = m_{i+1}$, die Dimensionen des darauffolgenden Layers. Die Aktivierungsfunktion der Neuronen im Layer i ist durch ϕ_i gegeben. Die insgesamte Anzahl an Layer k, des Netzes wird auch als Tiefe des Netzes bezeichnet. In Abbildung 2.4 wird ein Rechenschritt eines MLPs, dass in Abschnitt 4.3 verwendet wird, visualisiert. Die Daten, welche als Zeitreihe gegeben sind, werden geeignet transformiert und der Ausgang des Netzes zum letzten Zeitschritt addiert. Im Vergleich zu den rekurrenten Netzwerkstrukturen werden in einem MLP die Ausgangswerte nicht durch vorherige Eingänge beeinflusst. Um diesen Umstand auszugleichen, verwenden wir mehrere aufeinanderfolgende Zeitschritte einer Zeitreihe als Eingang.

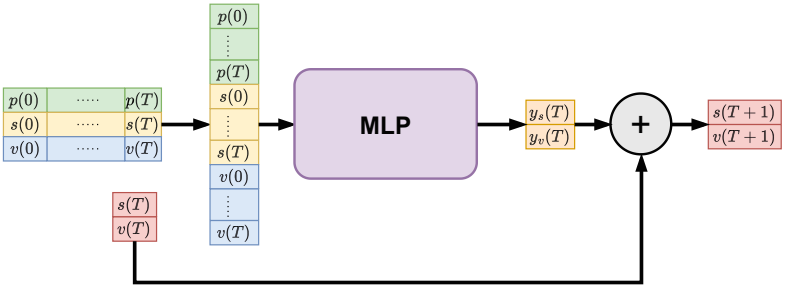

Abbildung 2.4: Forward pass in einem Multilayer Perceptron anhand des Beispiels der Hauptstufe des, in Abschnitt 4.1 beschriebenen, Pneumatikventils mit Druck, Position und Geschwindigkeit in Form von Zeitreihen als Netzeingang.

Wir sehen in Abbildung 2.4, dass die Daten zunächst in Form einer Zeitreihe gegeben sind, die anschließend in einen Vektor zusammengefügt werden. Der Ausgang des Netzes wird auf den letzten Systemzustand addiert, um den Schätzwert für den nächsten Zeitschritt zu erhalten. In Algorithmus 2 ist dargestellt, wie es mit dem Netz aus Abbildung 2.4 möglich ist N Zeitschritte in die Zukunft zu schätzen. Dabei ist T die Fensterlänge, also die Anzahl an aufeinanderfolgenden Zeitschritten, die einen gemeinsamen Netzeingang bilden.

Algorithm 2: MLP forward pass

input : $(p(0), \ldots, p(T+N); s(0), \ldots, s(T); v(0), \ldots, v(T))$
output: $(\hat{s}(T+1), \hat{v}(T+1), \ldots, \hat{s}(T+N+1), \hat{v}(T+N+1))$

begin
  $\hat{s}(T+1 : T+N+1) = 0$
  $\hat{v}(T+1 : T+N+1) = 0$
  output = empty.list()
  for $i = 0$ **to** N **do**
    inp $= (p(i), \ldots, p(T+i); s(i), \ldots, s(T+i); v(i), \ldots, v(T+i))$
    $x =$ transform(**inp**) # size(Input) = (3, T)
    ds, dv = model(x) # size(Input) = (3T, 1)
    if i == *0:* **then**
      $\hat{s}(T+1+i) = s(T+i) +$ ds
      $\hat{v}(T+1+i) = v(T+i) +$ dv
      else
        $\hat{s}(T+1+i) = \hat{s}(T+i) +$ ds
        $\hat{v}(T+1+i) = \hat{v}(T+i) +$ dv
      end
    end
    output.append($\hat{s}(T+1+i), \hat{v}(T+1+i)$)
  end
end

2.2.2 LSTM Architektur

Das von Hochreiter und Schmidhuber in [19] eingeführte LSTM, ist eine rekurrente Netzwerkarchitektur, die das Problem verschwindender beziehungsweise explodierender Gradienten bei dem BPTT (Backpropagation through time) Algorithmus löst. Das LSTM erreicht dies durch die Verwendung von Gates, die den Fluss von Information regulieren. Dadurch können Zusammenhänge in Daten auch über sehr große zeitliche Abstände gelernt werden. Die häufigsten Anwendungsfälle für LSTMs sind Übersetzung, Bilderkennung und Modellierung von Zeitreihen. LSTM ist eine der am weitest verbreiteten und effektivsten Architekturen und wird in Anwendungen, beispielsweise zur Übersetzung und Spracherkennung, von Konzernen wie Google [20] oder Microsoft [21] verwendet. Ein weniger naheliegender Anwendungsfall wäre, dass es dem Google DeepMind Team gelang, mit Hilfe von *Reinforcement Learning,* mit LSTMs, das hochkomplexe Strategiespiel StarCraft II, besser als jeder Mensch, zu spielen. Diese und weitere praktische Anwendungen wurden von Jürgen Schmidhuber unter [22] zusammengefasst. Wir wenden uns nun der konkreten Beschreibung der Architektur zu. In Abbildung 2.5 wird der Aufbau einer memory cell, also einer LSTM Zelle, wie er in [19] beschrieben wird, dargestellt.

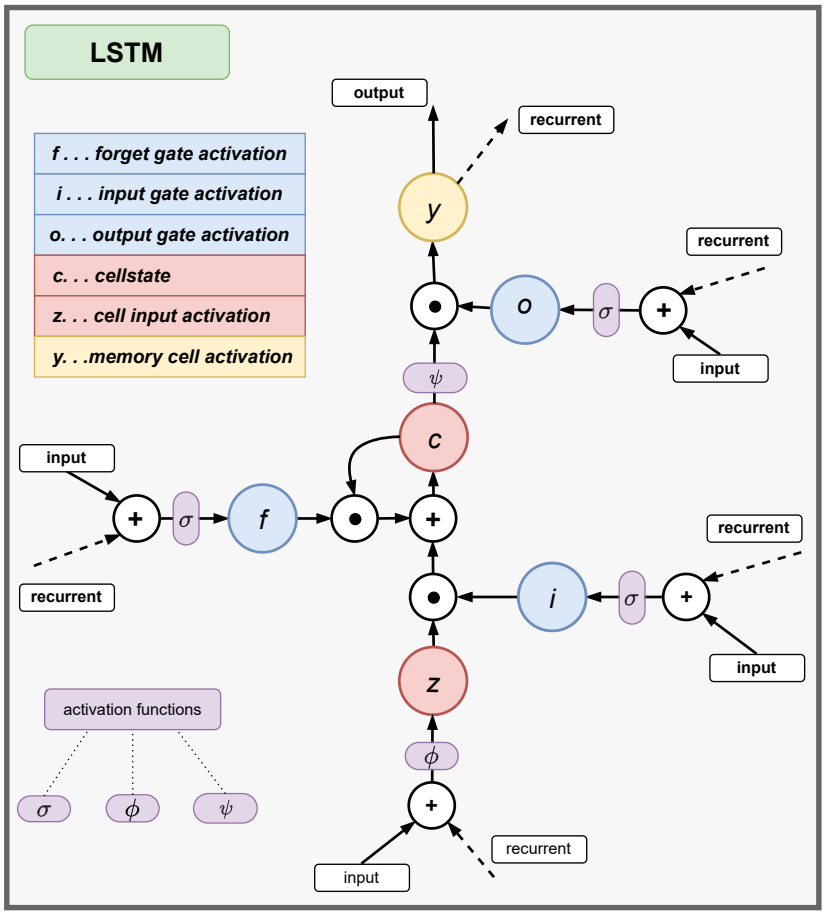

Abbildung 2.5: Darstellung einer LSTM-Zelle mit drei Gates.

Die Funktion der LSTM-Zelle wird durch folgende Gleichungen beschrieben,

$$
\begin{aligned}
i(t) &= \sigma(A_i x(t) + R_i y(t-1)),\\
o(t) &= \sigma(A_o x(t) + R_o y(t-1)),\\
f(t) &= \sigma(A_f x(t) + R_f y(t-1)),\\
z(t) &= \phi(A_z x(t) + R_z y(t-1)),\\
c(t) &= f(t) \odot c(t-1) + i(t) \odot z(t),\\
y(t) &= o(t) \odot \psi(c(t)).
\end{aligned}
$$

Wobei durch $\odot$ das Hadamard-Produkt, also die punktweise Multiplikation von zwei Vektoren notiert wird. Dabei gilt

$$\boldsymbol{v} \odot \boldsymbol{u} = w, \quad w_i = u_i v_i.$$

Die Aktivierungsfunktionen $\phi, \psi \in \{\sigma, \tanh, \mathrm{ReLU}\}$ sind frei wählbar. Für die Aktivierungsfunktion der Gates wird meist die Sigmoid Funktion (2.3) gewählt.

Die Aktivierungen der drei Gates und des Zelleneingangs hängen jeweils von dem neuen Eingang x zum Zeitpunkt t, sowie von dem vorherigen Ausgang der Zelle $y(t-1)$ ab. In den Gleichungen für die Aktivierung von Zellenzustand $c(t)$ und Zellenausgang $y(t)$ sehen wir, wie durch Multiplikation mit den Aktivierungen der Gates der Fluss an Information durch die Zelle reguliert wird.

Die folgende Abbildung 2.6 zeigt, wie wir mit einem LSTM zur Ableitungsschätzung, einen Zeitschritt in die Zukunft schätzen können. Ein Fenster an Werten, das sich aus Eingängen und Zuständen des Systems zusammensetzt, dient als Eingang für das Netzwerk. Der Ausgang zum letzten Zeitschritt wird auf den letzten Systemzustand addiert, um unsere Prädiktion zu erzeugen.

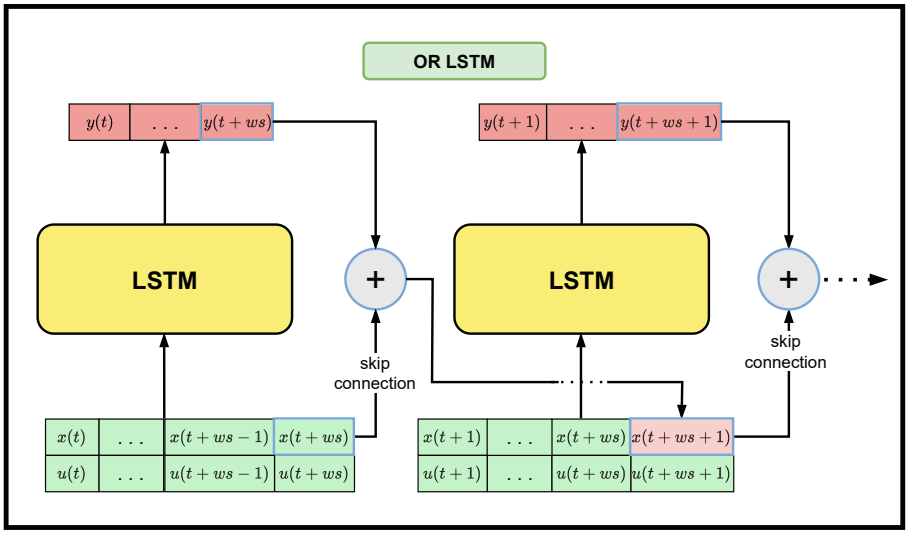

Abbildung 2.6: Prädiktion eines Zeitschritts mit einem LSTM Ableitungsschätzer mit Output Recurrence (OR-LSTM-dt), bei dem der Ausgang $y(t+ws)$ mit dem Eingang zum nächsten Zeitpunkt, über eine Skip-Connection, verbunden ist.

In Abbildung 2.6 ist ein LSTM Ableitungsschätzer mit Output Recurrence (OR-LSTM-dt), bestehend aus einer LSTM-Zelle, dargestellt. Die Darstellung zeigt, wie das Netz aus einem Eingang in Form einer Zeitreihe einen Ausgang erzeugt und anschließend Eingang und Ausgang zum letzten Zeitpunkt über eine Skip-Connection verbindet, um diesen als Teil des nächsten Eingangs zu nutzen. Eine Skip-Connection ist eine direkte Verbindung von Input zu Output Layer, die die hidden Layer des Netzes überspringt. Skip-Connections wurden in Zusammenhang mit *Residual neural networks (ResNets)*, die in [23] beschrieben werden, bekannt. Mittlerweile werden sie in diversen Netzwerkarchitekturen eingesetzt. Zur besseren Übersicht ist das Netzwerk dafür zweimal versetzt abgebildet. In den, für die Experimente in dieser Arbeit trainierten Modellen, wurden jedoch stets mehrere LSTM Zellen hintereinander geschaltet. Dabei dient der Ausgang aus der ersten Zelle als Eingang für die nächste Zelle.

2.2.3 TCN Architektur

Im folgenden Abschnitt beschreiben wir den Aufbau eines TCN, angelehnt an die Beschreibung in [24].

Der Aufbau des Netzes ergibt sich durch eine Hinteraneinanderreihung mehrerer *Temporal Blocks*, die jeweils zwei Faltungen mit Dilatation, die als *dilated Conv 1D* dargestellt sind, enthalten. Die dritte Faltung, *Conv 1D*, vor der Skip-Connection, entspricht einer Faltung mit Dilatation, für die d in Gleichung (2.2.3), gleich eins ist. Die Struktur eines solchen Blocks ist in Abbildung 2.7 angegeben.

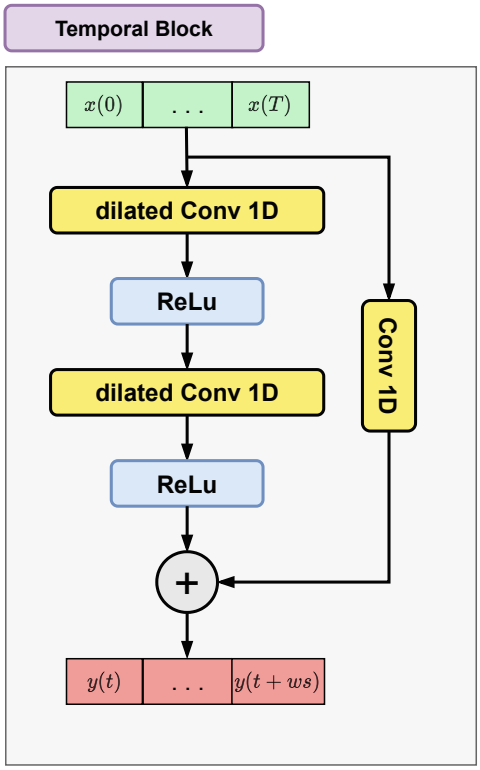

Abbildung 2.7: Darstellung eines Temporal Blocks, bestehend aus zwei Convolutional Layers mit ReLU Aktivierungsfunktionen und einer Skip-Connection.

In jeder der, in einem Block, aufeinander folgenden Faltungen erhöht sich, durch die Dilatation, der zeitliche Abstand zwischen den Eingangswerten, die in eine Faltung mit dem Filter eingehen. Dies erlaubt dem Ausgang des Netzes einen deutlich längeren zeitlichen Kontext in den Eingangsdaten zu erlernen. Die Faltung mit Dilatation für einen Eingang $\boldsymbol{x} \in \mathbb{R}^n$ und einen Filter $f \in \mathbb{R}^k$ ist durch

$$F(s) = (\boldsymbol{x} *_d f)(s) = \sum_{i=0}^{k-1} f_i x_{s-di}, \tag{2.5}$$

gegeben. Da ein TCN Block jeweils zwei Faltungen mit Dilatation enthält, erhöht sich das rezeptive Feld, also die Anzahl an Zeitschritten, die ein Eingang zurückliegen darf, um den Ausgang noch beeinflussen zu können, um $1 + 2d(k-1)$. Dabei ist k die Länge des Filters und d die Dilatation des TCN Blocks. Üblicherweise steigen die Dilatationen d_i von n aufeinander folgenden TCN Blöcken exponentiell an, also, $d_i = b^i$ mit $i \in \{1, \ldots, n\}$. Das

rezeptive Feld r des finalen Ausgangs beträgt damit

$$r = 1 + \sum_{i=0}^{n-1} 2(k-1)d_i,$$

beziehungsweise

$$r = 1 + \sum_{i=0}^{n-1} 2(k-1)b^i = 1 + 2(k-1)\frac{b^n - 1}{b-1}.$$

In Abbildung 2.8, wird das rezeptive Feld eines Ausgangs eines TCNs dargestellt, wobei, zur besseren Übersicht, der Übergang von einer Ebene zur jeweils darüberliegenden, nur aus einer Faltung mit Dilatation besteht. Dadurch erhöht sich das rezeptive Feld pro Ebene nur um $(k-1)d_i$ und wir erhalten ein rezeptives Feld von

$$r = 1 + \sum_{i=0}^{2} (k-1)d_i = 1 + 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 = 13.$$

Da für die Faltung mit den Filtern oftmals mehr, als die verfügbaren Werte nötig sind, werden fehlende Werte beim *padding*, meist mit 0, aufgefüllt.

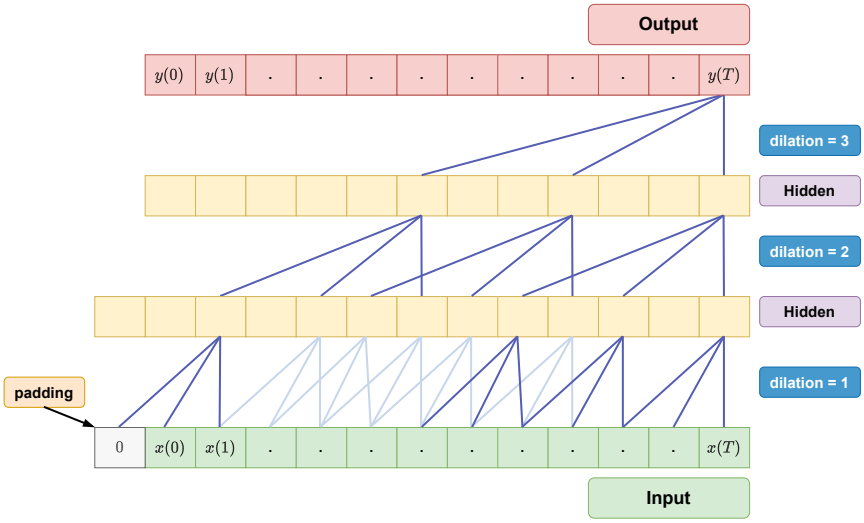

Abbildung 2.8: Darstellung des rezeptiven Felds eines Ausgangswerts in einem TCN. Die Abbildung zeigt die Berechnung von $y(T)$ zum Zeitpunkt $T = 12$, wobei der fehlende Wert durch padding aufgefüllt wurde.

Eine detaillierte Darstellung des forward pass durch ein TCN bestehend aus einem Temporal Block, ist in Abbildung 2.9, dargestellt. Die Zeitreihe, die den Netzeingang bildet, wird mit 0 aufgefüllt und anschließend wird eine Faltung gefolgt von einer Addition mit einem Bias Term durchgeführt. Nach Anwendung einer ReLU Aktivierungsfunktion (2.2) erhalten wir eine hidden state und wiederholen den Vorgang, wobei der hidden state anstelle

des Eingangs tritt. Nach Addition des Eingangs über eine Skip-Connection und Anwendung einer affinen Transformation erhalten wir den Netzausgang. Die lernbaren Parameter eines TCN sind die Gewichte der Filter, wobei ein dritter Filter für die Faltung vor der Skip-Connection hinzugefügt wurde. Dieser zusätzliche Filter versichert, dass die Dimension von Eingang und Ausgang gleich sind.

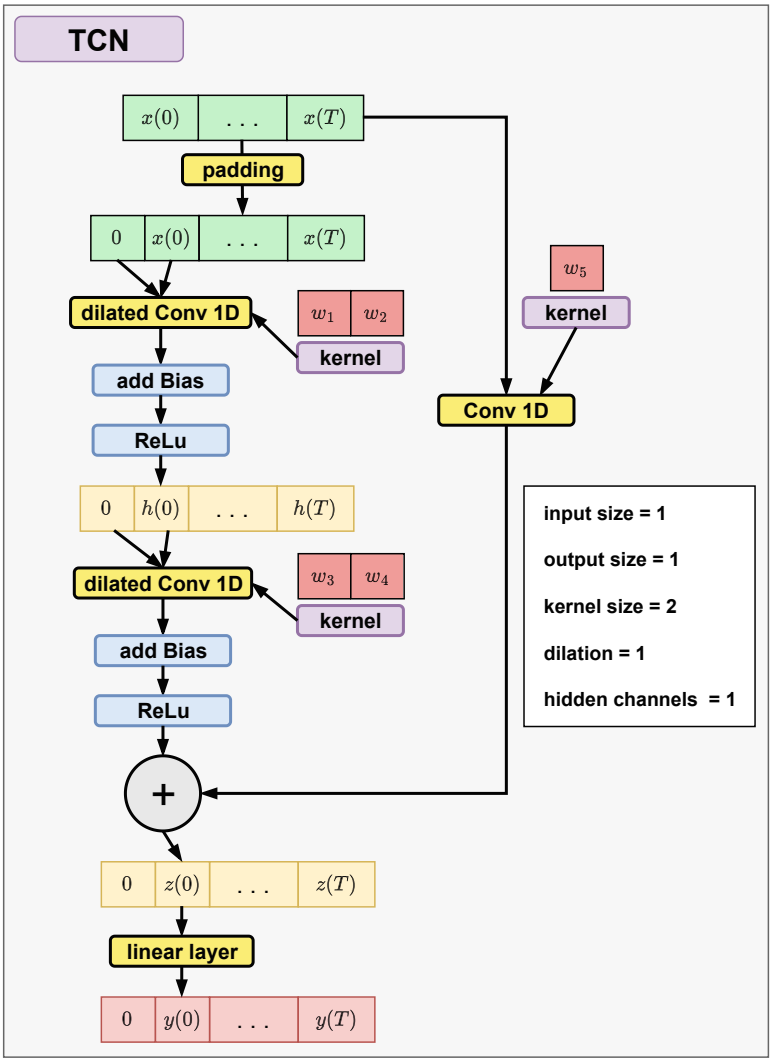

Abbildung 2.9: Detaildarstellung des forward pass eines TCN Netzes bestehend aus einem Temporal Block.

Bemerkung. Abbildung 2.9 ist eine simplifizierte Darstellung. Hierbei wurde der Dropout Layer und Weight normalization Layer, die man in den meisten Implementierungen findet, weggelassen wurden. Diese können in vielen Netzwerkarchitekturen eingesetzt werden und sind für das Verständnis der TCN Architektur nicht essenziell.

2.2.4 Neural controlled differential equation

Neural controlled differential equations (NCDEs) sind neuronale Netze, die eine Erweiterung von *Neural ordinary differential equations* (NODEs) darstellen. NODEs, die in [25] vorgestellt werden, kann man sich als eine stetige Fortsetzung eines Recurrent Neural Network (RNN) oder eines Residual Neural Network (ResNet) vorstellen, siehe dazu [13, Kapitel 10] sowie [23]. Dabei wird die Evolution der hidden state des Netzes durch eine Differentialgleichung der Form

$$\frac{d}{dt}h(t) = f(h(t), t, \theta),$$

beschrieben. Der Vorteil dabei besteht darin, dass man beliebige numerische Lösungsverfahren für gewöhnliche Differentialgleichungen nutzen kann, um die obige Gleichung, für einen Anfangszustand $h(0)$, zu lösen.

Ein Nachteil der NODEs ist, dass, nach dem Anfangswert, keine neuen Daten berücksichtigt werden. Das wird durch NCDEs, wie sie in [26] beschrieben werden, gelöst. Wir werden bei unserer Definition striktere Annahmen als notwendig treffen, um zu umfangreiche Erklärungen zu vermeiden.

Definition 2.2.1 (Neural controlled differential equation)**.** Es seien $n, m \in \mathbb{N}$, und $x : [0, T] \to \mathbb{R}^n$ eine differenzierbare Funktion, sowie $f : \mathbb{R}^m \to \mathbb{R}^{m\times n}$ eine Lipschitz-stetige Funktion. Dann existiert für $y_0 \in \mathbb{R}$, eine eindeutige und stetige Lösung $y : [0, T] \to \mathbb{R}^m$, die

$$\begin{aligned} y(0) &= y_0, \\ y(t) &= y(0) + \int_0^t f(y(s))\frac{\mathrm{d}x(s)}{\mathrm{d}s}\mathrm{d}s \quad \text{für } t \in (0, T], \end{aligned} \tag{2.6}$$

erfüllt. Wir nennen (2.6) eine *controlled differential equation*. Mit einem neuronalen Netz f_θ an Stelle von f und neuronalen Netzen h_θ und l_θ, die wir jeweils auf Anfangswert und Lösung anwenden können, erhalten wir

$$\begin{aligned} y(0) &= h_\theta(y_0), \\ y(t) &= y(0) + \int_0^t f_\theta(y(s))\frac{dx(s)}{\mathrm{d}s}\mathrm{d}s, \\ o(t) &= l_\theta(y(t)), \end{aligned} \tag{2.7}$$

eine NCDE. Die Funktionen $f_\theta, l_\theta, h_\theta$ sind von der Parametermenge $\theta \in \Theta \subseteq \mathbb{R}^k$ für $k \in \mathbb{N}$ abhängig.

Der Ausgang des NCDE ist dann durch $o(t)$ gegeben. Die Architekur der neuronalen Netze, die in Definition 2.2.1 vorkommen ist frei wählbar, wobei MLPs die häufigste Wahl sind.

Des Weiteren ist das Lösungsverfahren für die Differentialgleichung von $y(t)$ frei wählbar. Am interessantesten für unsere Zwecke ist jedoch, dass die Steuerung x als stetiges Eingangssignal vorhanden sein muss. Da diese Daten meist durch diskrete Folgen von Messwerten gegeben sind, ist eine Interpolation der Daten notwendig.

Bei Betrachtung von realen Systemen ist es durchaus denkbar, dass Daten zu irregulären Zeitpunkten gemessen werden. Beispielsweise könnte ein Sensor einen neuen Messwert nur dann aufzeichnen, wenn eine gewisse Änderung zur letzten Messung besteht. Umgang mit solchen irregulär gemessenen Daten könnte mit den bereits vorgestellten Netzwerkarchitekturen unter Umständen nur schwierig möglich sein. Im Gegensatz dazu sind Implementierungen von NCDEs bereits mit verschiedenen Interpolationsverfahren ausgestattet und damit auch für solche Problemstellungen geeignet.

2.2.5 Physics-informed neural networks

Die, in [27] eingeführten, Physics-informed neural networks (PINNs) sind neuronale Netze, die zusätzlich zum üblichen Fehler, der den Abstand zwischen dem Ausgang des Netzwerks und dem Zielwert der Daten misst, auch einen sogenannten *physics loss* bewerten. Dieser beschreibt, wie gut das Netz die beschreibende Differentialgleichung, oder andere physikalische Randbedingungen des Systems, erfüllt. Häufig werden PINNs im Zusammenhang mit partiellen Differentialgleichungen genutzt, die numerisch nur schwer oder nicht lösbar sind. Wollen wir zum Beispiel ein neuronales Netz $\mathcal{N}(t)$ trainieren, dass die Differentialgleichung

$$\mathcal{D}(x) = a\frac{\mathrm{d}^2}{\mathrm{d}t^2}x(t) + b\frac{\mathrm{d}}{\mathrm{d}t}x(t) + \gamma(t) = 0, \quad \forall t \in [t_0, t_1],$$

mit $a, b \in \mathbb{R}$ und $\gamma \in \mathcal{C}(\mathbb{R})$ löst, dann können wir zusätzlich zu dem üblichen Fehler

$$L_{MSE} = \frac{1}{n}\sum_{i=1}^{n} \|x(t_i) - \mathcal{N}(t_i)\|^2,$$

einen physics loss,

$$L_{physics} = \frac{1}{m}\sum_{j=1}^{m} \|\mathcal{D}(\mathcal{N}(t_j))\|^2,$$

mit $n, m \in \mathbb{N}$, einführen. Dabei sind die Trainingsdaten $x(t_i)$ Punkte einer Lösung der Differentialgleichung und $t_i, t_j \in [t_0, t_1]$. Zur Berechnung des Gradienten, wird der neue Fehler

$$L_{PINN} = L_{MSE} + L_{physics},$$

betrachtet.

Besonders im Zusammenhang mit dynamischen Systemen sind PINNs interessant, da oft Systemgleichungen bekannt sind, wie dies beispielsweise in [1] gezeigt ist, wo das Flugverhalten eines Quadrocopters mit PINNs modelliert wird.

2.3 Zentrale Theoreme und Algorithmen

Im Jahr 1847 führte Augustin-Louis Cauchy in seinem Papier [28], hier in englischer Übersetzung, die Idee des Gradientenabstiegsalgorithmus ein. Seitdem ist die auf Gradienten basierende Optimierung die am häufigsten verwendete Art der Optimierung, insbesondere im Bereich des maschinellen Lernens.

In diesem Abschnitt werden wir die wichtigsten Algorithmen auflisten, die beim Training von neuronalen Netzwerken in diesem Projekt verwendet wurden, und einen Überblick darüber geben, wie diese Algorithmen in moderner Software implementiert sind. Die vorgestellten Algorithmen existieren in vielen verschiedenen Versionen, die in den letzten Jahren immer weiter optimiert wurden. Wir werden jedoch ihre grundlegenden Formen betrachten, um ein besseres Verständnis für die genutzten Verfahren zu erhalten und um den Fokus auf die zugrunde liegende Mathematik richten zu können.

2.3.1 Approximationstheorem für neuronale Netze

Dieser Abschnitt beschäftigt sich mit der, in Abschnitt 2.1 erwähnten, Fähigkeit neuronaler Netze, stetige Funktionen beliebig genau zu approximieren. Die Grundlage dafür liefert, das Approximationstheorem. Für einen detaillierteren Beweis wird auf [29] und [30] verwiesen. In diesem Abschnitt werden wir [29, Theorem 1] beweisen und besprechen, wie es mit Lernalgorithmen in Zusammenhang steht. Der Satz bezieht sich nur auf neuronale Netze mit einem hidden Layer und einer nicht-Linearität.

Zunächst geben wir einige Definitionen, die wir als bekannt voraussetzen wollen.

- $I_n = [0,1] \times ... \times [0,1]$ stellt den n-dimensionalen Einheitswürfel dar,
- $(C(I_n, \mathbb{R}), \|.\|_\infty)$ den Raum der stetigen Funktionen auf I_n mit der Supremumsnorm,
- $\mathcal{M}(I_n)$ die Menge der signierten endlichen regulären Borelmaße.

Definition 2.3.1 (sigmoidartig)**.** Wir nennen σ *sigmoidartig*, wenn

$$\sigma(t) = 1 \quad \text{für } t \to \infty \quad \text{und} \quad \sigma(t) = 0 \quad \text{für } t \to -\infty, \tag{2.8}$$

gilt.

Definition 2.3.2 (diskriminierend)**.** Wir nennen σ *diskriminierend*, wenn für ein Maß $\mu \in \mathcal{M}(I_n)$ gilt, dass aus

$$\int_{I_n} \sigma(\langle \boldsymbol{y}, \boldsymbol{x} \rangle + \beta) \mathrm{d}\mu(\boldsymbol{x}) = 0, \quad \forall \boldsymbol{y} \in \mathbb{R}^n, \beta \in \mathbb{R},$$

$\mu = 0$ folgt, wobei $\boldsymbol{x} \in \mathbb{R}$.

Das folgende Lemma zeigt eine, für den Beweis des Approximationstheorems, essenzielle Eigenschaft von Aktivierungsfunktionen. Für den Beweis verweisen wir auf [29, Lemma 1]. Der Begriff *messbar*, in diesem Abschnitt, ist im Sinne der Maßtheorie zu verstehen.

Lemma 2.3.1. Jede beschränkte, messbare, sigmoidartige Funktion ist diskriminierend.

Nach Lemma 2.3.1 ist insbesondere auch jede stetige sigmoidartige Funktion diskriminierend.

Schließlich wollen wir die Menge der neuronalen Netze mit einem einzigen hidden Layer der Dimension N und einer Aktivierungsfunktion σ mit

$$S_\sigma = \{G(\boldsymbol{x}) = \Sigma_{j=1}^{N} \alpha_j \sigma(\langle \boldsymbol{y}_j, \boldsymbol{x} \rangle + \beta_j) \quad | \quad \alpha_j, \beta_j \in \mathbb{R}, \boldsymbol{y}_j \in \mathbb{R}^n\}, N \in \mathbb{N}\},$$

bezeichnen. Dabei ist $\langle \cdot, \cdot \rangle$ das Standardskalarprodukt im $\mathbb{R}^n$. Die Parameter α_j, β_j, und $\boldsymbol{y}_j$ entsprechen den lernbaren Parametern eines neuronalen Netzes im Kontext des maschinellen Lernens. Die, für den Beweis des folgenden Theorems erwähnten, Sätze aus dem Gebiet der Funktionalanalysis können in [31] und [32] nachgelesen werden.

Theorem 2.3.1. Es sei σ eine beliebige stetige diskriminierende Funktion. Dann ist die Menge S_σ dicht in $C(I_n)$.

Beweis. Zunächst gilt, dass S_σ Teilmenge und sogar Untervektorraum von $C(I_n)$ ist, also $S_\sigma \leq C(I_n)$.

Wir wollen $\overline{S_\sigma} = C(I_n)$ zeigen, das heißt, dass S_σ dicht in der Menge der stetigen Funktionen auf dem n-dimensionalen Einheitswürfel ist. Wir wollen einen Beweis durch Widerspruch führen und nehmen dafür an, dass S_σ nicht dicht in $C(I_n)$ ist. Das bedeutet, dass $\overline{S_\sigma} \subsetneq C(I_n)$ gilt. Nach dem Satz von Hahn-Banach, existiert nun ein stetiges lineares Funktional F, für das $F \neq 0$ und $F(\overline{S_\sigma}) = 0$ gilt.

Nach dem Darstellungssatz von Riesz gibt es ein Maß $\mu \in \mathcal{M}(I_n)$, sodass

$$F(h) = \int_{I_n} h(\boldsymbol{x}) \mathrm{d}\mu(\boldsymbol{x}), \quad \forall h \in C(I_n),$$

gilt.

Da wegen $F(\overline{S_\sigma}) = 0$ auch $F(S_\sigma) = 0$ gilt, erhalten wir

$$\int_{I_n} \sigma(\langle \boldsymbol{y}_j, \boldsymbol{x} \rangle + \beta_j) \mathrm{d}\mu(\boldsymbol{x}) = 0$$

für alle $\beta \in \mathbb{R}$ und alle $\boldsymbol{y} \in \mathbb{R}^n$. Da wir σ als diskriminierend vorausgesetzt haben, muss nun $\mu = 0$ gelten. Daraus folgt, $F = 0$ was wiederum ein Widerspruch zur Folgerung aus dem Satz von Hahn-Banach und damit zur Annahme, woraus $\overline{S_\sigma} = C(I_n)$ folgt. □

Aus Lemma 2.3.1 und Theorem 2.3.1 erhalten wir, dass für stetige sigmoidartige Funktionen S_σ dicht in $C(I_n)$ ist. Das heißt, für eine beliebige Funktion $f(\boldsymbol{x}) \in C(I_n)$ und ein festes $\varepsilon > 0$, gibt es $G(\boldsymbol{x}) \in S_\sigma$, sodass

$$|G(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon, \quad \forall \boldsymbol{x} \in I_n$$

gilt.

Es lassen sich also sämtliche stetige Funktionen auf dem n-dimensionalen Einheitswürfel durch ein neuronales Netz, mit einem hidden Layer und einer sigmoidartigen Aktivierungsfunktion, beliebig genau approximieren. Dabei ist es wichtig zu beachten, dass nicht nur die Parameter der Netze variieren, sondern auch die Größe des Netzes. Somit ist nicht garantiert, dass ein Netz mit fixierter Größe eine Funktion beliebig genau approximieren kann, indem nur die Parameter $\alpha_i, \beta_i, \boldsymbol{y}_i$, $\forall i = 1, \ldots, N$ passend gewählt werden.

Bemerkungen.

(a) Aus dem obigen Satz folgt unmittelbar, dass sich auch vektorwertige Funktionen beliebig genau approximieren lassen. Um das zu erreichen, müssen wir die Funktion lediglich komponentenweise approximieren. Weiters lässt I_n auf beliebige Intervalle transformieren. Im Fall $n = 1$ kann zum Beispiel durch die Wahl

$$\phi(x) = \frac{x - a}{b - a},$$

das Intervall $[a, b]$ auf $[0, 1]$ transformieren.

(b) Bei Klassifikationsproblemen, wie zum Beispiel in der Bilderkennung, möchte man gerne Funktionen approximieren, die entscheiden können, ob ein Element Teil einer gewissen Menge ist. Für eine Grundmenge M suchen wir eine Funktion $f(x)$, für die gilt $f(x) = 1, \quad \forall x \in A$ und $f(x) = 0, \quad \forall x \in M \setminus A$. Offenbar ist ein solches f nicht stetig. Der Aufbau und Beweis von Theorem 2.3.1 lässt sich aber leicht auf Funktionen in $\mathcal{L}^1(I_n, \mu)$ umbauen, wie in [29, Theorem 4] demonstriert wird.

Es ist wichtig zu beachten, dass der hier angeführte Approximationssatz, nur die Existenz einer Folge von neuronalen Netzwerken garantiert, die eine gegebene Funktion mit einem beliebig kleinen positiven Fehler approximieren. Über einen Weg eine solche Folge zu konstruieren wird nichts ausgesagt. Die Anzahl der Summanden N für eine Funktion G, also die Anzahl der Neuronen in dem neuronalen Netz und damit könnten, notwendiger Speicherplatz und Rechenzeit, beliebig anwachsen. Daher bleibt die Aufgabe, ein passendes Netzwerk zu finden, hochgradig nichttrivial.

2.3.2 Backpropagation Algorithmus

In sämtlichen Gradientenverfahren wollen wir unsere Gewichte in Richtung eines Minimums der Fehlerfunktion

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|^2$$

bringen, indem wir uns einen kleinen Schritt in Richtung des negativen Gradienten der Fehlerfunktion bewegen. Die Gewichte des Netzes werden im Gradientenverfahren, mit

$$W_{neu} = W_{alt} - \eta \nabla_W, \tag{2.9}$$

aktualisiert. Der Parameter η ist die Lernrate. Später werden wir noch einen effektiveren Algorithmus vorstellen. Es stellt sich dennoch die Frage, wie man diese Gradienten berechnet. Der Backpropagation Algorithmus liefert eine Möglichkeit, Gradienten schnell und effizient zu berechnen. Für eine detailierte Beschreibung dieses Verfahrens siehe [**?**].

Im Folgenden betrachten wir ein MLP mit n Layern. Dabei bezeichnen wir mit

$$\boldsymbol{a}^l = f(\boldsymbol{s}^l) = f(W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l),$$

die Aktivierung des Layers l und mit $\boldsymbol{s}^l$ die Voraktivierung. Der Netzeingang ist durch $\boldsymbol{a}^0$ und der Netzausgang durch $\hat{\boldsymbol{y}} = \boldsymbol{a}^n$ gegeben. Um uns einen Index zu ersparen, berechnen wir die Ableitung der Fehlerfunktion für ein Gewicht aus dem ersten Layer,

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^1} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{s}^n} \cdot \ldots \cdot \frac{\partial \boldsymbol{s}^l}{\partial \boldsymbol{s}^{l-1}} \cdot \ldots \cdot \frac{\partial \boldsymbol{s}^1}{\partial w_{ij}^1}. \tag{2.10}$$

Definition 2.3.3 (Delta-Fehler)**.** Wir nennen,

$$\delta^l = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^l}$$

den Delta-Fehler von Layer l.

Die Aktivierungsfunktion für den Ausgang ist die Identität, womit wir

$$\delta^n = \hat{\boldsymbol{y}} - \boldsymbol{y},$$

erhalten.

Offenbar ist

$$\boldsymbol{s}^l = W^l f(\boldsymbol{s}^{l-1}) + \boldsymbol{b}^l,$$

woraus

$$\frac{\partial \boldsymbol{s}^l}{\partial \boldsymbol{s}^{l-1}} = W^l \mathrm{diag}(f'(\boldsymbol{s}^{l-1}))$$

folgt.

Durch Ableiten nach der Voraktivierung eines Layers l mit $l < n$, erhalten wir mit Hilfe der Kettenregel eine rekursive Darstellung für die Delta-Fehler in der Form

$$\delta^{l-1} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{l-1}} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^l} \frac{\partial \boldsymbol{s}^l}{\partial \boldsymbol{s}^{l-1}} = \delta^l W^l \mathrm{diag}(f'(\boldsymbol{s}^{l-1})) \tag{2.11}$$

angeben. Mit $\mathrm{diag}(f'(\boldsymbol{s}^{l-1}))$ bezeichnen wir eine Diagonalmatrix D mit $D_{ii} = f'(\boldsymbol{s}^{l-1})_i$. Die Diagonalgestalt der Jacobi-Matrix ergibt sich wegen

$$\frac{\partial s_i^l}{\partial s_j^{l-1}} = 0, \quad \forall i \neq j.$$

Schließlich leiten wir noch die Voraktivierung nach dem Gewicht ab und erhalten,

$$\frac{\partial \boldsymbol{s}^1}{\partial w_{ij}^1} = \frac{\partial W^1 \boldsymbol{a}^0}{\partial w_{ij}^1} = a_j^0 v_i,$$

wobei v_i den $i-$ten Einheitsvektor bezeichnet. Damit haben wir alles, was wir für die Darstellung des Gradienten benötigen. Wir erhalten

$$\frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w_{ij}^l} = \frac{\partial L(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^l} \frac{\partial \boldsymbol{s}^l}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}. \tag{2.12}$$

Bemerkung. Die Effizienz des Backpropagation Algorithmus kommt daher, dass man nach jeder Auswertung des Netzes, die Aktivierungen $\boldsymbol{a}^l$, $l = 0, \dots, n$, für jeden Layer speichern kann und anschließend beginnend mit dem letzten Layer, durch nur eine Matrix Multiplikation, die Delta-Fehler berechnen kann.

Will man für eine Menge an Daten $\mathcal{D}$ die zugehörigen Gradienten berechnen, geht man wie folgt vor:

1 Berechne $\hat{\boldsymbol{y}} = \mathcal{N}(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{D}$.

2 Speichere dabei $\boldsymbol{a}^l_{\boldsymbol{x}}, \quad \forall l \in \{1, \dots, L\}, \forall \boldsymbol{x} \in \mathcal{D}$.

3 Berechne $\delta^l_{\boldsymbol{x}} \quad \forall l \in \{1, \dots, L\}$.

4 Berechne daraus $\nabla_{\boldsymbol{x}}, \quad \forall \boldsymbol{x} \in \mathcal{D}$.

5 Berechne daraus $\nabla = \frac{1}{n} \sum_{\boldsymbol{x} \in \mathcal{D}} \nabla_{\boldsymbol{x}}, \quad n = |\mathcal{D}|$.

6 Aktualisiere die Gewichte mit $\mathcal{W}_{new} = \mathcal{W}_{old} - \eta \nabla$.

Die meisten dieser Schritte, können für alle Datenpunkte parallel berechnet werden, womit, besonders in Verbindung mit modernen GPUs, eine enorme Reduktion der Laufzeit, im Gegensatz zu sequentieller Berechnung, erzielt werden kann. Genaue Ausführungen dazu findet man in [15].

2.3.3 Backpropagation through Time Algorithmus

Als Nächstes betrachten wir die Berechnung von Gradienten für rekurrente Netzwerkarchitekturen, wie sie in [13] beschrieben wird. Dafür sei ein einfaches rekurrentes Netz, mit der Struktur,

$$\begin{aligned} \boldsymbol{h}(t) &= f(W\boldsymbol{x}(t) + R\boldsymbol{h}(t-1)) = f(\boldsymbol{s}(t)) \\ \hat{\boldsymbol{y}}(t) &= f(V\boldsymbol{h}(t)) \end{aligned} \tag{2.13}$$

gegeben. Diesmal müssen wir die Änderung über mehrere Zeitschritte zurückführen, anstatt über die mehreren Layer des Netzes. Für ein Update der Gewichte $\mathcal{W} = (W, R, V)$ wie in (2.9), benötigen wir $\frac{\partial L}{\partial \mathcal{W}} = (\frac{\partial L}{\partial W}, \frac{\partial L}{\partial R}, \frac{\partial L}{\partial V})$.

Die Fehlerfunktion L_Σ ist durch die Summe der Fehler über die Anzahl der berechneten Zeitschritte gegeben, wobei für

$$L_\Sigma(\boldsymbol{y}(1), \dots, \boldsymbol{y}(T), \hat{\boldsymbol{y}}(1), \dots, \hat{\boldsymbol{y}}(T)) = \sum_{t=1}^{T} L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t)),$$

Für eine einfache Darstellung kann man die halbierten Fehlerquadrate als Fehlerfunktion L wählen und erhält mit $L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t)) = \frac{1}{2}\|\boldsymbol{y}(t) - \hat{\boldsymbol{y}}(t)\|_2^2$, dass

$$e_k(t) = \frac{\partial L_\Sigma}{\partial \hat{y}_k(t)} = \hat{y}_k(t) - y_k(t)$$

gilt. Zunächst berechnen wir $\frac{\partial L}{\partial V}$. Durch Verwendung der Kettenregel erhalten wir,

$$\frac{\partial L_\Sigma}{\partial v_{ik}} = \sum_{t=1}^{T} \frac{\partial L(y_k(t), \hat{y}_k(t))}{\partial v_{ik}} = \sum_{t=1}^{T} \frac{\partial L(y_k(t), \hat{y}_k(t))}{\partial \hat{y}_k(t)} \frac{\partial \hat{y}_k(t)}{\partial v_{ik}}, \tag{2.14}$$

wobei

$$\hat{y}_k(t) = f\left(\sum_{j=1}^{I} v_{jk} h_j(t)\right),$$

die $k-$te Komponente des Ausgangs zum Zeitpunkt t ist.

Als Nächstes berechnen wir die Ableitung des Ausgangs $\hat{y}_k(t)$ im Hinblick auf v_{ik}. Zusammen mit der Formel für die Ableitung,

$$\frac{\partial \hat{y}_k(t)}{\partial v_{ik}} = f'\left(\sum_{j=1}^{I} v_{jk} h_j(t)) h_i(t)\right), \tag{2.15}$$

erhalten wir den Anteil des gesuchten Gradienten für V durch

$$\frac{\partial L_\Sigma}{\partial v_{ik}} = \sum_{t=1}^{T} e_k(t) f'\left(\sum_{j=1}^{I} v_{jk} h_j(t)) h_i(t)\right). \tag{2.16}$$

Für die Ableitung der Gewichte W und R definieren wir erneut den Delta-Fehler mit einer in der Zeit rekursiven Darstellung.

Definition 2.3.4 (Delta-Fehler BPTT)**.** Der Delta-Fehler BPTT δ ist durch die Ableitung der Fehlerfunktion $\frac{\partial L}{\partial \boldsymbol{s}(t)}$ gegeben. Weiters genügt der Delta-Fehler der rekursiven Darstellung

$$\begin{aligned}
\delta(t)^T &= \frac{\partial L_\Sigma}{\partial \boldsymbol{s}(t)} = \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(\frac{\partial L}{\partial \boldsymbol{h}(t)} + \frac{\partial L}{\partial \boldsymbol{s}(t+1)} \frac{\partial \boldsymbol{s}(t+1)}{\partial \boldsymbol{h}(t)}\right) \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(\frac{\partial L}{\partial \hat{\boldsymbol{y}}(t)} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{h}(t)} + \frac{\partial L}{\partial \boldsymbol{s}(t+1)} \frac{\partial \boldsymbol{s}(t+1)}{\partial \boldsymbol{h}(t)}\right) \frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{s}(t)} \\
&= \left(e(t)^T \operatorname{diag}(f'(V^T \boldsymbol{h}(t))) V^T + \delta(t+1)^T R^T\right) \operatorname{diag}(f'(\boldsymbol{s}(t))).
\end{aligned}$$

Die obige Definition, in Kombination mit der Kettenregel, erlaubt uns eine kompakte Darstellung für die beiden übrigen Ableitungen. Wir erhalten

$$\frac{\partial L_\Sigma}{\partial r_{ij}} = \sum_{t=1}^{T} \frac{\partial L(y_j(t), \hat{y}_j(t))}{\partial r_{ij}(t)} = \sum_{t=1}^{T} \frac{\partial L(y_j(t), \hat{y}_j(t))}{\partial s_i(t)} \frac{\partial \boldsymbol{s}(t)}{\partial r_{ij}(t)} = \sum_{t=1}^{T} \delta_i(t) h_j(t-1), \tag{2.17}$$

für die den Anteil von R und

$$\frac{\partial L_\Sigma}{\partial w_{id}} = \sum_{t=1}^{T} \frac{\partial L(y_d(t), \hat{y}_d(t))}{\partial w_{id}(t)} = \sum_{t=1}^{T} \frac{\partial L(y_d(t), \hat{y}_d(t))}{\partial s_i(t)} \frac{\partial \boldsymbol{s}(t)}{\partial w_{id}(t)} = \sum_{t=1}^{T} \delta_i(t) x_d(t-1) \tag{2.18}$$

für den Anteil von W am Gradienten. Schließlich können wir das Update sämtlicher Gewichte mit

$$\mathcal{W}_{neu} = \mathcal{W}_{alt} - \eta \frac{\partial L_\Sigma}{\partial \mathcal{W}},$$

durchführen.

2.3.4 Backpropagation Algorithmus für neuronale Netze mit output recurrence

Die Backpropagation für neuronale Netze mit OR wird in [8] beschrieben, wobei die Gradienten nicht für ein LSTM, sondern nur für ein RNN, berechnet werden. Bis auf den Unterschied, dass das von uns betrachtete RNN, nur den Ausgang des Netzes zum vorherigen Zeitpunkt als Eingang des Netzes erhält, vergleiche $x(t)$ in (2.13), wollen wir hier genauso vorgehen. Diese Variation enthält einen zusätzlichen Delta-Fehler, da der Ausgang $\boldsymbol{y}(t)$, nicht nur mit dem Fehler $L(\boldsymbol{y}(t), \hat{\boldsymbol{y}}(t))$, sondern auch mit dem hidden Layer $\boldsymbol{h}(t+1)$, verbunden ist. Um den Überblick zu behalten, wird eine etwas kompaktere Notation genutzt.

Das von uns betrachtete RNN ist durch

$$\boldsymbol{h}(t) = \tanh(R\boldsymbol{h}(t-1) + U\hat{\boldsymbol{y}}(t-1)), \tag{2.19}$$
$$\hat{\boldsymbol{y}}(t) = V\boldsymbol{h}(t), \tag{2.20}$$

gegeben. Wir beginnen die Berechnung der Delta-Fehler wieder bei T. Für den letzten Zeitpunkt erhalten wir

$$\delta_{\hat{\boldsymbol{y}}(T)} = \frac{\partial L_\Sigma}{\partial \hat{\boldsymbol{y}}(T)} = \frac{\partial L_\Sigma}{\partial L(T)} \frac{\partial L(T)}{\partial \hat{\boldsymbol{y}}(T)} = \hat{\boldsymbol{y}}(T) - \hat{\boldsymbol{y}}(T),$$
$$\delta_{\boldsymbol{h}(T)} = \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(T)} = \frac{\partial L_\Sigma}{\partial \hat{\boldsymbol{y}}(T)} \frac{\partial \hat{\boldsymbol{y}}(T)}{\partial \boldsymbol{h}(T)} = V^T(\hat{\boldsymbol{y}}(T) - \boldsymbol{y}(T)).$$

Mit der Kettenregel erhalten wir die rekursiven Darstellungen,

$$\begin{aligned}
\delta_{\hat{\boldsymbol{y}}(t)} &= \frac{\partial L_\Sigma}{\partial L(t)} \frac{\partial L(t)}{\partial \hat{\boldsymbol{y}}(t)} + \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(t+1)} \frac{\partial \boldsymbol{h}(t+1)}{\partial \hat{\boldsymbol{y}}(t)} \\
&= (\hat{\boldsymbol{y}}(t) - \boldsymbol{y}(t)) + U\mathrm{diag}(1 - (\boldsymbol{h}(t+1)^2)\delta_{\boldsymbol{h}(t+1)}, \\
\delta_{\boldsymbol{h}(t)} &= \frac{\partial L_\Sigma}{\partial \hat{\boldsymbol{y}}(t)} \frac{\partial \hat{\boldsymbol{y}}(t)}{\partial \boldsymbol{h}(t)} + \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(t+1)} \frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)} \\
&= V^T(\hat{\boldsymbol{y}}(t) - \boldsymbol{y}(t)) + (V^T U + R)\mathrm{diag}(1 - (\boldsymbol{h}(t+1)^2)\delta_{\boldsymbol{h}(t+1)}
\end{aligned}$$

für die Delta-Fehler, dabei wurde

$$\begin{aligned}\frac{\partial \boldsymbol{h}(t+1)}{\partial \hat{\boldsymbol{y}}(t)} &= U \operatorname{diag}(\tanh'(R\boldsymbol{h}(t) + U\hat{\boldsymbol{y}}(t))) \\ &= U \operatorname{diag}((1 - \tanh(R\boldsymbol{h}(t) + U\hat{\boldsymbol{y}}(t))^2)) \\ &= U \operatorname{diag}((1 - \boldsymbol{h}(t+1)^2)), \\ \frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)} &= R \operatorname{diag}(\tanh'(R\boldsymbol{h}(t) + U\hat{\boldsymbol{y}}(t))) \\ &= R \operatorname{diag}((1 - \tanh(R\boldsymbol{h}(t) + U\hat{\boldsymbol{y}}(t))^2)) \\ &= R \operatorname{diag}((1 - \boldsymbol{h}(t+1)^2)),\end{aligned}$$

verwendet, wobei $(\cdot)^2$ und $\tanh(\cdot)$ wieder punktweise angewendet werden. Um den Gradienten von L_Σ, der Summe der Fehler L zu jedem Zeitschritt, zu berechnen, leiten wir jeden Summanden ab und addieren diese anschließend. Bis auf den komplizierteren Delta-Fehler, entspricht die Formel jener des normalen Backpropagation through time Algorithmus. Die Gradienten sind durch

$$\begin{aligned}\nabla_V &= \sum_{t=1}^{T} \frac{\partial L_\Sigma}{\partial \hat{\boldsymbol{y}}(t)} \frac{\partial \hat{\boldsymbol{y}}(t)}{\partial V} = \sum_{t=1}^{T} \delta_{\hat{\boldsymbol{y}}(t)} \boldsymbol{h}(t), \\ \nabla_R &= \sum_{t=1}^{T} \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial R} = \sum_{t=1}^{T} \delta_{\boldsymbol{h}(t)} \operatorname{diag}((1 - h(t)^2)\boldsymbol{h}(t-1), \\ \nabla_U &= \sum_{t=1}^{T} \frac{\partial L_\Sigma}{\partial \boldsymbol{h}(t)} \frac{\partial \boldsymbol{h}(t)}{\partial U} = \sum_{t=1}^{T} \delta_{\boldsymbol{h}(t)} \operatorname{diag}((1 - h(t)^2)\hat{\boldsymbol{y}}(t),\end{aligned}$$

gegeben. Mit den berechneten Gradienten können nun wiederum sämtliche Gewichte aktualisiert werden.

2.3.5 Computation Graphs als Beispiel einer modernen Implementierungsvariante für neuronale Netze

Moderne Software Packages wie `PyTorch` verwenden sogenannte *Computation Graphs*, wie sie in [13] beschrieben werden, für effizientes Auswerten von neuronalen Netzen und rasche Berechnung der Gradienten. In einem solchen Graphen sind die Knoten Variablen und die Kanten mathematische Operationen.

Möchte man zum Beispiel für

$$\boldsymbol{z}(\boldsymbol{x}) = f(g(\boldsymbol{x})),$$

mit $\boldsymbol{y} = g(\boldsymbol{x})$, den Gradienten berechnen, so erhält man mit der Kettenregel

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}.$$

Ein Knoten im Graph, muss also nur seine Ableitung in Bezug auf seinen Eingang kennen, um seine Ableitung zu berechnen. Einmalige Traversierung des Graphen erlaubt also die Auswertung der Funktion, also auch die Berechnung sämtlicher Ableitungen.

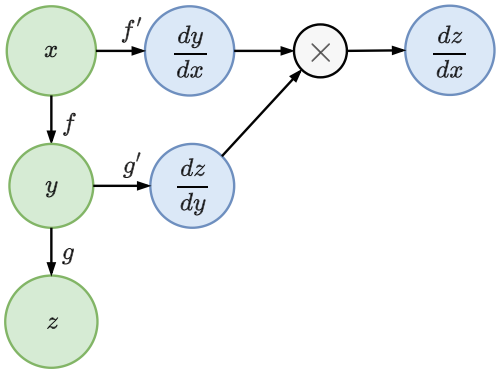

Abbildung 2.10: Beispiel eines einfaches Computation Graphs für die Funktion $f(g(\boldsymbol{x}))$.

In der Abbildung 2.10, ist ein Computation Graph für das oben angegebene Beispiel dargestellt. Die Vorteile von Computation Graphs sind:

- Übersichtlicher und strukturierter Aufbau,
- Modularität und Wiederverwendbarkeit von Komponenten,
- effiziente Berechnung der Ableitungen und auswerten des Graphen, sowie
- Parallelisierbarkeit der Berechnungen von Teilgraphen.

Die ersten beiden Punkte sind besonders in praktischer Hinsicht relevant, da sie einfache Implementierungen von komplizierten Netzwerkarchitekturen, mit benutzerfreundlichen APIs, wie denen von `PyTorch` oder `TensorFlow`, ermöglichen.

2.3.6 Adam Algorithmus

Einer der meistverwendeten Algorithmen beim Training von neuronalen Netzen ist der, in [33] vorgestellte, *Adam* Algorithmus. Dieser ist ein modernes stochastisches Gradientenverfahren, das eine Kombination aus dem AdaGrad und dem RMSProp Algorithmus darstellt. Der Algorithmus kann verwendet werden, um eine beliebige stochastische skalarwertige Funktion zu minimieren. Da wir die Summe von Fehlern über alle Datenpunkte

$$\frac{1}{n}\sum_{i}^{n} L(\boldsymbol{x}_i, \hat{\boldsymbol{x}}_i, \theta),$$

mit einem Gradientenverfahren minimieren wollen, müssten wir für jeden Summanden den Gradienten $\nabla_\theta L(\boldsymbol{x}_i, \hat{\boldsymbol{x}}_i, \theta)$ berechnen. Da diese Berechnung bei sehr großen Datensätzen

sehr aufwendig wird, approximieren wir stattdessen den Gradienten. Eine zufällige Stichprobe an Datenpunkten wird ausgewählt und die zugehörigen Gradienten berechnet. In unserem Fall ist das die Fehlerfunktion, $L(\theta)$, wobei Realisationen der Funktion $L_0(\theta), \ldots, L_T(\theta)$ sind. Diese erhalten wir, wenn wir unsere Fehlerfunktion auf einem Batch auswerten. Bevor wir den Ablauf des Algorithmus genauer betrachten, geben wir einen Pseudocode für den Algorithmus an. Die Operationen $.^2$ und $\sqrt{.}$ sind punktweise zu verstehen.

Algorithm 3: ADAM

input : $\alpha, \beta_1\beta_2, L(\theta), \theta_0$
output: θ_T

begin
  $m_0, v_0 = 0, 0$ # initialize
  $t = 0$
  for $t = 0$ **to** T **do**
    $g_t = \nabla_\theta L_t(\theta_{t-1})$ # calculate gradient
    $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ # update first moment estimate
    $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ # update second raw moment estimate
    $\hat{m}_t = m_t / (1 - \beta_1^t)$ # correct bias
    $\hat{v}_t = v_t / (1 - \beta_2^t)$
    $\theta_t = \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ # update parameters
  end
  return θ_T
end

Nach Berechnung der Gradienten werden die exponentiell geglätteten Durchschnitte vom erstem Moment und vom zweiten zentralen Moment berechnet. Die exponentiell abfallenden Gewichte β_1, β_2 werden nahe bei 1 initialisiert, wodurch m_t und v_t, besonders für kleine t, einen Bias zu 0 haben. Dieser wird durch Division durch $(1 - \beta_i^t)$ korrigiert. Schließlich werden die Parameter mit,

$$\theta_t = \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

aktualisiert, wobei aufgrund der Abschätzung

$$\left| \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \right| \leq \frac{(1 - \beta_1)}{\sqrt{1 - \beta_2}} \leq 1,$$

die Größe der Änderung in jeder Komponente des Parametervektors durch die Lernrate α begrenzt ist. Dadurch ist der Gradienten, insbesondere invariant in Bezug auf Skalierung.

3 Simulation von dynamischen Systemen

In diesem Abschnitt werden die Grundlagen dynamischer *Systeme*, wie sie in [34], [35] und [36] beschrieben werden, kurz zusammengefasst und der System Begriff eingeführt.

3.1 Dynamische Systeme

Ein System ist ein Objekt, indem verschiedene Variablen unter Einfluss von externen Eingän-gen und Störungen miteinander interagieren Ausgänge erzeugt werden. Der Systembegriff findet in diversen wissenschaftlichen Disziplinen Anwendung und ist daher bewusst sehr universell formuliert. In dieser Arbeit werden die betrachteten Systeme stets elektromechanische Systeme sein, wobei wir den Einfluss von Störungen großteils vernachlässigen und uns auf die Simulation der zeitlichen Evolution von Ausgangsgrößen fokussieren wollen.

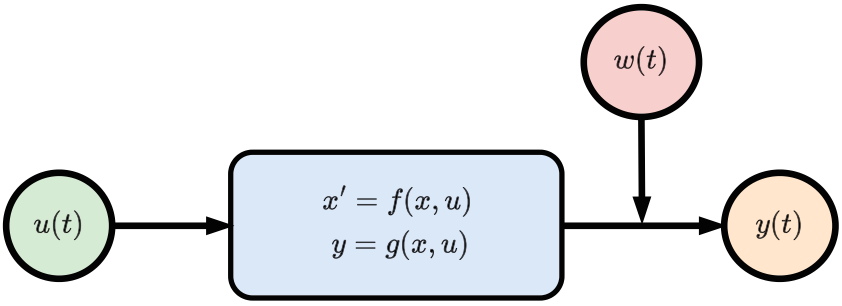

Abbildung 3.1: Blockdiagramm eines dynamischen Systems, mit Eingangssignal $u(t)$, Systemzustand $x(t)$, Ausgangssignal $y(t)$, und einem Störsignal $w(t)$.

Oft ist es hilfreich, dynamische Systeme, durch Blockdiagramme zu visualisieren, diese geben einen Überblick über den Signalfluss durch das System. In Abbildung 3.1 ist ein dynamisches System, mit Eingang $u = u(t)$, Systemzustand $x = x(t)$, Ausgang $y = y(t)$ und Störung $w = w(t)$, dargestellt. Nun wollen wir uns einer formalen Definition zuwenden, wir übernehmen dafür jene aus [36, Definition 2.1.1], da sie einen sehr allgemeinen Systembegriff darstellt, wobei wir die einzelnen Axiome nicht ausführen wollen.

Definition 3.1.1 (Dynamisches System)**.** Wir nennen das Tupel $\Sigma = (T, U, \mathcal{U}, X, Y, \varphi, \eta)$ ein dynamisches System mit Zeitbereich T, Eingangswertebereich U, Raum der Eingangsfunktionen $\mathcal{U}$, Zustandsraum X, Ausgangswertebereich Y, Zustandsfunktion φ und Ausgangsfunktion η wenn,

- $T, U, \mathcal{U}, X, Y$ nicht leer sind und $T \subset \mathbb{R}$ sowie $U \subset U^T$,

- $\eta : T \times X \times U \to Y$,
- $\varphi : T^2 \times X \times \mathcal{U} \to X$

und das Intervalaxiom, das Konsitenzaxiom, das Kausalitätsaxiom und die Cocycle property erfüllt sind.

Wählen wir einen Anfangswert (t_0, x_0) und einen Eingang $u(t)$, so erhalten wir eine Zustandstrajektorie

$$t \mapsto x(t) = \varphi(t, t_0, x_0, u(t)), \quad t \in [t_0, t_1)$$

und die zugehörige Ausgangstrajektorie

$$t \mapsto y(t) = \eta(t, x(t), u(t)).$$

Für alle, im Kontext dieser Arbeit, relevanten Systeme, lässt sich die Zustandstrajektorie zu gegebenem Eingang $u(t)$ und Anfangswert x_0 als Lösung eines Differentialgleichungssystems modellieren. Für eine formale Definition, siehe [36, Definition 2.1.12]. Da unser Simulationsansatz nicht von der Art der Differentialgleichung abhängt, werden wir auf die übliche Unterscheidung zwischen linearen und nichtlinearen Systemen verzichten.

Definition 3.1.2 (Differenzierbares dynamisches System)**.** Ein differenzierbares dynamisches System ist ein dynamisches System, für das eine Funktion $f : T \times X \times U \to \mathbb{R}^n$ existiert, sodass für alle $\boldsymbol{x}_0 \in X$ und alle $u(t) \in \mathcal{U}$, das Anfangswertproblem

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{x}(t) &= f(t, \boldsymbol{x}(t), \boldsymbol{u}(t)), \quad t \geq t_0, \\ \boldsymbol{x}(t_0) &= \boldsymbol{x}_0 \end{aligned} \tag{3.1}$$

eine eindeutige Lösung besitzt.

Diese Definition ist zwar um einiges praktischer, jedoch nur, solange die rechte Seite der Differentialgleichung f explizit bekannt ist. Bei den betrachteten mechanischen Systemen lässt sich die Form von f, aus physikalischen Gesetzmäßigkeiten ableiten, für elektrische Systeme beispielsweise durch die Maxwell-Gleichungen und für mechanische Systeme beispielsweise durch den Energieerhaltungssatz und die Newtonschen Gesetze. Das Bestimmen, sämtlicher Parameter der rechten Seite der Differentialgleichung kann sich als sehr schwierig herausstellen, da diese von Materialeigenschaften und Alterungseffekten abhängen und oft nur experimentell ermittelt werden können. Als Nächstes wird eine diskrete Version von (3.1.2) beschrieben, in der zwei äquivalente Darstellungen formuliert werden, die im weiteren Verlauf dieser Arbeit zentral sein werden.

Definition 3.1.3 (Diskretisierung eines Differenzierbaren dynamischen Systems)**.** Für ein dynamisches System der Form (3.1.2) und eine Menge von Abtastpunkten $\{t_k,\ k = 1, \ldots, n,\ n \in \mathbb{N}\} \subseteq T$ mit erhält man für die Systemzustände $\boldsymbol{x}_k = \boldsymbol{x}(t_k)$ die diskrete Darstellung,

$$\begin{aligned} \boldsymbol{x}_{k+1} &= F(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0 \ldots n, \\ \boldsymbol{x}_0 &= \boldsymbol{x}(0), \end{aligned} \tag{3.2}$$

wobei

$$F(\boldsymbol{x}_k, \boldsymbol{u}_k) = \boldsymbol{x}_0 + \int_{t_0}^{t_{k+1}} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t$$

gilt. Alternativ erhält man,

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + G(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0 \ldots n, \\ \boldsymbol{x}_0 &= \boldsymbol{x}(0), \end{aligned} \tag{3.3}$$

mit

$$G(\boldsymbol{x}_k, \boldsymbol{u}_k) = \int_{t_k}^{t_{k+1}} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t.$$

Diese beiden Formulierungen sind offenbar analytisch äquivalent, da

$$\begin{aligned} F(\boldsymbol{x}_k, \boldsymbol{u}_k) &= \boldsymbol{x}_0 + \int_{t_0}^{t_{k+1}} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t \\ &= \boldsymbol{x}_0 + \int_{t_0}^{t_k} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t + \int_{t_k}^{t_{k+1}} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t \\ &= \boldsymbol{x}_0 + (\boldsymbol{x}_k - \boldsymbol{x}_0) + \int_{t_k}^{t_{k+1}} f(t, \boldsymbol{x}(t), \boldsymbol{u}(t))\mathrm{d}t \\ &= \boldsymbol{x}_k + G(\boldsymbol{x}_k, \boldsymbol{u}_k) \end{aligned}$$

gilt. Sie sind jedoch im Kontext der Modellierung verschieden, wie im Abschnitt 3.2 beschrieben wird.

3.2 Modellierungsansätze für neuronale Netze

In diesem Abschnitt wird beschrieben, wie neuronale Netze zur Modellierung und Simulation von dynamischen Systemen verwendet werden können. Wir beschränken uns in dieser Arbeit, auf dynamische Systeme die durch Gleichungen der Form (3.1.2) beschrieben werden, wobei stets auf deren Diskretisierungen (3.1.3) beziehungsweise (3.1.3) übergegangen wird. Die Formulierung (3.1.2) ähnelt einerseits dem Euler-Verfahren,

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + hf(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k),$$

wobei h die Schrittweite und $t_{k+1} = t_k + h$ ist. Andererseits entspricht sie bis auf den zusätzlichen Eingang $\boldsymbol{u}_k$, genau dem Aufbau eines ResNet, wie in [37] genauer ausgeführt wird. Ziel ist es, die Funktionen F aus Darstellung (3.1.3), beziehungsweise G aus Darstellung (3.1.3), zu approximieren. An die Stelle von F beziehungsweise G, tritt nun ein neuronales Netz. Ein solches neuronales Netz zur Simulation zu nutzen, bedeutet, aus einem Datenpunkt, der aus Zustand und Eingang eines Systems zum Zeitpunkt t_k besteht, den Zustand des Systems zum Zeitpunkt t_{k+1} zu berechnen. Der Teil des Datenpunktes, der den Systemzustand beschreibt, kann entweder ein gemessener Wert oder ein vom Netzwerk zu einem früheren Zeitpunkt berechneter Wert sein. Der Systemeingang $\boldsymbol{u}_k$ ist stets durch die Daten vorgegeben.

Definition 3.2.1 (direkte Schätzer)**.** Neuronale Netze, unabhängig von ihrer sonstigen Struktur, werden in dieser Arbeit *direkte Schätzer* genannt, wenn sie zukünftige Zustände des Systems mittels,

$$\hat{\boldsymbol{x}}_{k+1} = \mathcal{N}_{\mathcal{F}}(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{3.4}$$

direkt aus dem vorherigen Datenpunkt, bestehend aus, Zustand und Eingang, schätzen.

Definition 3.2.2 (Ableitungsschätzer)**.** Ein Neuronales Netze wird *Ableitungsschätzer* genannt, wenn es eine ableitungsartige Größe bestimmt. Diese wird auf den vorherigen Zustand des Systems addiert, um eine Prädiktion der Form

$$\hat{\boldsymbol{x}}_{k+1} = \boldsymbol{x}_k + \mathcal{N}_{\mathcal{G}}(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{3.5}$$

zu erhalten.

Bemerkung. Die Bezeichnung Ableitungsschätzer kommt davon, dass $\mathcal{N}_{\mathcal{G}}$ als Ableitung in einem Euler-Verfahren mit $h = 1$, interpretiert werden könnte.

Wir wollen an dieser Stelle die Ähnlichkeit von dem, in 3.2, beschriebenen Netzwerk und einem Residual neural network diskutieren. ResNet verwenden eine Skip-Connection, die eine direkte Verbindung zwischen Eingang des Netzes und dem Ausgangswert eines Layers darstellen. Diese Verbindung erlaubt einem Netzwerk mit vielen Layern seine optimale Tiefe selbst zu wählen, indem überflüssige Layer übersprungen werden. Im Gegensatz dazu, wird in einem Ableitungsschätzer, nur ein Teil des Netzeingangs, nämlich der Systemzustand zum letzten Zeitpunkt, durch eine Skip Connection mit dem Ausgang des Netzes verbunden.

Die andere Inspiration für 3.2 ist das weiter oben angesprochene Euler-Verfahren. Tatsächlich sind die beiden Formulierungen (3.4) und (3.5) zwar theoretisch äquivalent, jedoch

ist es durchaus denkbar, dass Ableitungsschätzer weniger stark von Änderungen in den Anfangswerten betroffen sind, insbesondere wenn diese Anfangswerte nicht Teil der Trainingsdaten waren. Ein Experiment dazu führen wir im Kapitel 4.3 durch.

Bemerkung. In [38] wird beschrieben, dass Systeme der Form (3.1.3) auch Darstellungen der Form

$$\boldsymbol{x}_{k+1} = H(\boldsymbol{x}_k, \cdots, \boldsymbol{x}_{k-n}, \boldsymbol{u}_k, \cdots, \boldsymbol{u}_{k-n}), \tag{3.6}$$

erlauben. Abgesehen von diesem theoretischen Ergebnis, ist es oft sinnvoll, einem neuronalen Netzwerk so viele Daten wie möglich zur Verfügung zu stellen. Beim Lernen von Zeitreihen ist es üblich nicht nur den letzten Zeitschritt, sondern die letzten n Zeitschritte als Eingang zu benutzen. Die Darstellung (3.6) wird in allen Implementierungen in dieser Arbeit verwendet. Dennoch beschränken wir uns in diesem Abschnitt auf die kompakteren Darstellungen (3.1.3) beziehungsweise (3.1.3).

3.3 Black-Box-Modellierungsansatz

In diesem Abschnitt wollen wir die verwendeten Modellierungsansätze beschreiben, um die Methoden dieser Arbeit besser in die Gebiete der Modellierung und Systemidentifikation einordnen zu können. Für detaillierte Erklärungen dieser Konzepte siehe [39, Kapitel 1.4], sowie [40].

Die von uns betrachteten Systemmodelle, werden durch *White-Box* oder *Grey-Box* Modellen beschrieben. Hierbei wird Wissen über die Eigenschaften des Systems verwendet, um Gleichungen aufzustellen, die das System vollständig beziehungsweise teilweise beschreiben. Mit passenden Techniken aus der Systemidentifikation werden die notwendigen Paramter bestimmt, wie zum Beispiel Reibkraftkoeffizienten, welche von Materialeigenschaften abhängen. Das fertige Modell ist gut interpretierbar, da der Zusammenhang zwischen Eingängen und Ausgängen durch mathematische Gleichungen beschrieben ist.

Im Gegensatz dazu ist für die Erstellung eines *Black-Box* Modells kein Wissen über die physikalischen Zusammenhänge des Systems notwendig. Es gilt lediglich eine Funktion zu finden, die gegebene Eingangsgrößen auf die zugehörigen Ausgänge abbildet. In dieser Arbeit ist diese Funktion durch ein Neuronales Netz gegeben. Das Bestimmen der Parameter des Netzes erfolgt im Training, wobei diese Parameter nicht mit den oben erwähnten physikalischen Parametern verwechselt werden dürfen. Ein großer Nachteil der Black-Box Modellierung ist, dass es unmöglich ist, das Modell zu interpretieren um daraus ein besseres Verständnis über das zugrunde liegende System zu erlangen. Der große Vorteil, jedoch, liegt darin, dass allein auf Basis von Messdaten ein Modell erstellt werden kann. Es sind folglich keine, oftmals nur Mithilfe von Expertenwissen möglichen, mathematischen Modellierungsschritte notwendig. Auch die, für mathematisch-physikalische Modelle notwendige Parameteridentifikation basierend auf oftmals maßgeschneiderten Experimenten entfällt.

3.4 Beispiel eines Pendels mit äußerer Krafteinwirkung

Bevor wir uns mit komplexeren Systemen befassen, werden die in Abschnitt 3.2 vorgestellten Methoden anhand eines einfachen Beispiels betrachtet. Als Beispiel für ein dynamisches

System dient ein motorisiertes Einfachpendel, das in Abbildung 3.2 dargestellt ist. Man stelle sich einen masselosen Stab vor, der am oberen Ende drehbar gelagert ist und an dessen Ende eine Punktmasse befestigt ist. Dieser schwingt von links nach rechts, wobei seine Geschwindigkeit gegebenenfalls durch einen Motor gesteuert wird.

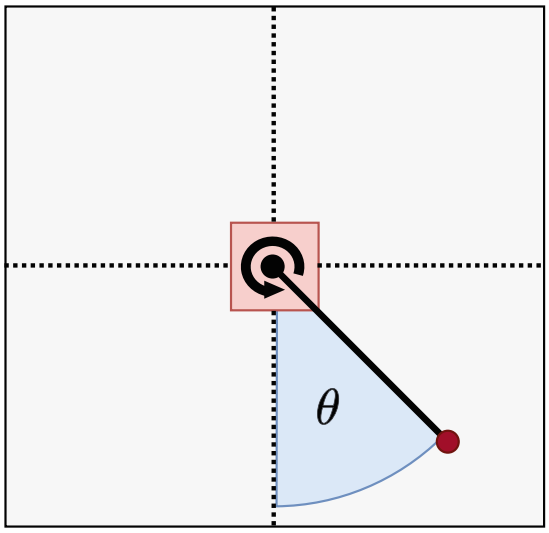

Abbildung 3.2: Überblicksdarstellung eines motorisierten Pendels mit Auslenkungswinkel θ.

Das System wird durch die Gleichungen

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\theta(t) &= \omega(t),\\ \frac{\mathrm{d}}{\mathrm{d}t}\omega(t) &= -\frac{g}{l}\sin(\theta(t) - \frac{b}{ml^2}\omega(t) + \frac{a}{ml^2}\cos(\Omega t)\end{aligned}$$

beschrieben. Hierbei sind g die Erdbeschleunigung, l, die Länge des Pendels und b und a Faktoren, die die Dämpfung beziehungsweise die Amplitude des Eingangs angeben. Da die Interpretation der physikalischen Größen für uns nicht von Interesse ist, setzen wir $|g| = |l|$, $\lambda = \frac{-b}{ml^2}$ und $u(t) = \frac{a}{ml^2}\cos(\Omega t)$. Durch diese Substitutionen erhalten wir,

$$\begin{aligned}\theta'(t) &= \omega(t),\\ \omega'(t) &= -\sin(\theta(t) + \lambda\omega(t) + u(t).\end{aligned} \tag{3.7}$$

Mit $\boldsymbol{x} = [\theta, \omega]^T$ folgt schließlich,

$$\boldsymbol{x}' = f(\boldsymbol{x}, u).$$

die bekannte Darstellung aus Abschnitt 3.1.

Als Nächstes wollen wir einen Ableitungsschätzer trainieren, um die zukünftige Entwicklung des Systems zu schätzen. Ausgänge eines neuronalen Netzes werden, wie in der Literatur üblich, mit einem Dach ($\hat{y}$) gekennzeichnet und alle Messwerte werden mit einer Tilde ($\tilde{y}$) markiert.

Wir unterscheiden zwischen drei Methoden zur Auswertung eines trainierten Netzes:

A: Schätzung vom Anfangswert aus gemäß

$$\hat{\boldsymbol{x}}_{k+1} = \hat{\boldsymbol{x}}_k + \mathcal{N}(\hat{\boldsymbol{x}}_k, \boldsymbol{u}_k),$$

mit $\hat{\boldsymbol{x}}_0 = \boldsymbol{x}_0$.

B: 1-Schritt Schätzung mit Messdaten

$$\hat{\boldsymbol{x}}_{k+1} = \tilde{\boldsymbol{x}}_k + \mathcal{N}(\tilde{\boldsymbol{x}}_k, \boldsymbol{u}_k).$$

C: 1-Schritt Schätzung mit einer Mischung aus Simulations- und Messdaten der Form

$$\begin{bmatrix}\hat{\theta}\\ \hat{\omega}\end{bmatrix}_{k+1} = \begin{bmatrix}\tilde{\theta}\\ \hat{\omega}\end{bmatrix}_k + \mathcal{N}\left(\begin{bmatrix}\tilde{\theta}\\ \hat{\omega}\end{bmatrix}_k, \boldsymbol{u}_k\right).$$

Für die letzten beiden Methoden, B und C, ist eine fortlaufende Messung aller relevanten Zustände für jede weitere Prädiktion notwendig.

Bemerkung. In dieser Arbeit werden wir uns hauptsächlich mit Methode A auseinandersetzen, da wir an Simulationen über größere Zeiträume, ohne Interaktion mit dem System, interessiert sind. Ziel ist ein Black-Box Modell, welches wie ein klassisches mathematisch-physikalisch motiviertes Modell als Simulationsmodell beispielsweise zur Entwicklung von Regelungsstrategien oder für Simulationsstudien genutzt werden kann. Es sind dennoch Situationen denkbar, in denen Simulationen, die nur einen oder einige wenige Zeitschritte in die Zukunft schätzen, benötigt werden.

Als Beispiel für Methode B wäre Folgendes vorstellbar. In einer Fabrik wird die Temperatur von Werkstücken nach einem Arbeitsschritt gemessen. Aus dieser Folge von Messungen schätzt ein neuronales Netz die Temperatur des nächsten Werkstücks und die optimale Einstellung für die Parameter der Strecke. Mit dieser Schätzung könnte man dann den nächsten Arbeitsschritt anpassen. Dabei ist es wichtig, dass kein Verständnis über die genauen Prozesse in der Strecke nötig ist. Allerdings benötigt man eine große Menge an Trainingsdaten die die optimale Temperatur und zugehörigen Parameter für die Produktion enthalten. Methode C kann beispielsweise bei der Ermittlung von Geschwindigkeiten sinnvoll sein. Typischerweise kann zumeist die Position oder der Winkel direkt gemessen werden, nicht aber die (Winkel)Geschwindigkeit. Mit Methode C kann nun auf Basis der gemessenen Positions oder Winkelinformation der gesamte Zustand eines mechanischen (Teil)Systems, also Position beziehungsweise Winkel und die dazugehörige (Winkel)Geschwindigkeit ermittelt werden.

Für das Training erzeugen wir, mit Hilfe eines numerischen Verfahrens, Lösungen von (3.7) im Intervall $T = [0, 30]s$ an 200 äquidistanten Stützstellen. Als Systemeingänge werden random walks verwendet. Um einen random walk zu erzeugen, starten man mit einem zufälligen Startwert und addiert dann einen Zufallswert aus einer Gleichverteilung darauf. So fährt man für die nötige Anzahl an Schritten fort, wobei das Vorzeichen eines Schritts gegebenenfalls geändert werden muss, wenn der nächste Wert außerhalb des zulässigen Wertebereichs liegen würde. Weiters wird angenommen, dass unsere Messdaten exakte Lösungen darstellen und kein Messrauschen vorhanden ist, d.h. $\boldsymbol{x} = \tilde{\boldsymbol{x}}$ gilt. Mit diesen Daten wird ein kleines Long-Short Term Memory neural network zur Ableitungsschätzung (LSTM-dt) trainiert. Die folgende Tabelle enthält die genauen Parameter des Netzes.

Model Parameter	**LSTM**
Hidden Size	5
Number of Cells	1
Window Size	4
Input Size	3
Output Size	2

Tabelle 3.1: Model Parameter für das neuronale Netz, das für die Pendelsimulation genutzt wurde.

Das resultierende Netzwerk ermöglicht Simulationen mit den Methoden A, B und C des Pendels, auch über deutlich längere Zeiträume als in den Trainingsdaten abgebildet. Die Ergebnisse zeigen über den gesamten Betrachtungszeitraum nur geringe Abweichungen zum modellbasierten Simulationsmodell. In diesem Abschnitt, vergleichen wir die Methoden A, und B, ein Vergleich von Methode B mit Methode C erfolgt im nächsten Unterkapitel. Methode A ist eine Schätzung, die nur vom Anfangswert abhängt, während Methode B jeweils nur den nächsten Zeitschritt (*nextstep prediction*) zu einem Messwert berechnet.

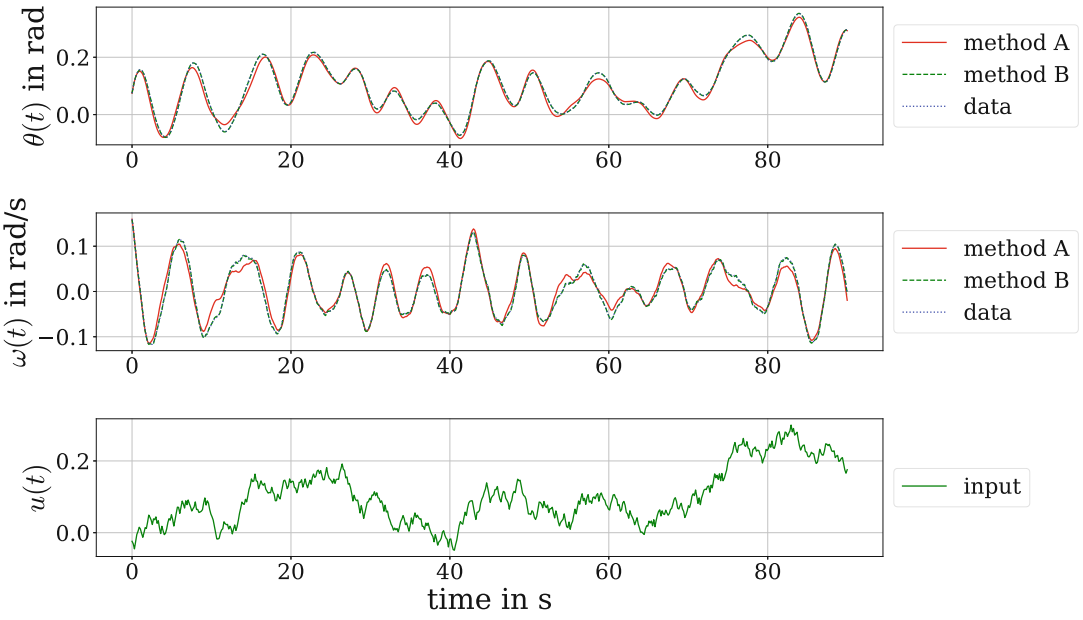

Abbildung 3.3: Pendel Simulation: Random walk als Eingang. Die oberen beiden Teilbilder zeigen die Zustände, Winkel und Drehwinkelgeschwindigkeit. Im untersten Teilbild ist der Systemeingang dargestellt.

Die Variation der Eingänge, welche im untersten Teilbild von 3.3 dargestellt ist, deckt einen großen Wertebereich ab. Die Ergebnisse im Winkel und der Winkelgeschwindigkeit zeigen deutliche kleinere Abweichungen von Methode B im Vergleich zu Methode A. Um die Unterschiede besser hervorzuheben und zu untersuchen, ob das Netzwerk die Dynamik des System gelernt hat, wird eine weitere Simulationsserie mit veränderten Eingängen durchgeführt. Diese Eingangssignale unterscheiden sich deutlich von den in den Trainingsdaten genutzten random walk Zeitreihen. Zur besseren Bewertung der Ergebnisse werden Fehlermaße wie folgt definiert.

Definition 3.4.1. Wir definieren den absoluten Fehler und den kumulativen Fehler zwischen zwei Zeitreihen $y(t_k)$ und $\hat{y}(t_k)$ mit $k = 1 \dots N$ als

$$e_a(y, \hat{y}, t_k) = |y(t_k) - \hat{y}(t_k)|,$$
$$e_c(y, \hat{y}, n) = \sum_{k=1}^{n} |y(t_k) - \hat{y}(t_k)|, \quad n \leq N.$$

Da wir stets den Fehler zwischen Daten y und der Schätzung eines neuronalen Netzes $\hat{y}$ vergleichen werden, schreiben wir abkürzend $e_a(y)$ beziehungsweise $e_c(y)$.

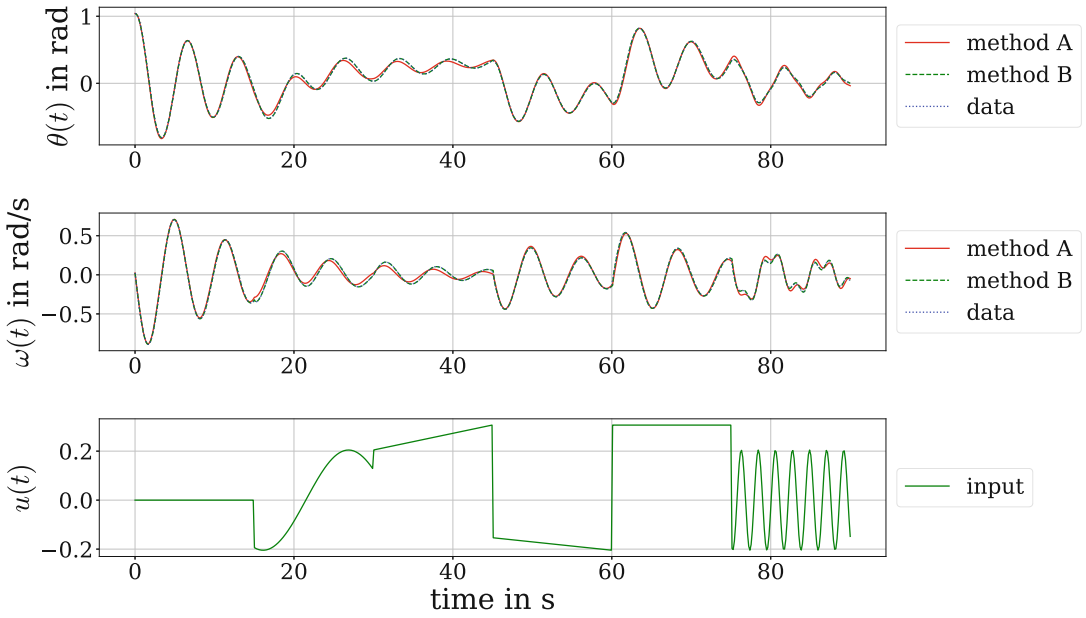

Abbildung 3.4: Pendel Simulation mit einem gemischten Eingangssignal, bestehend aus konstanten und stark oszillierenden Abschnitten und Sprungstellen.

Obwohl sich das Eingangssignal in Abbildung 3.4 stark von einem random walk unterscheidet ist eine sehr genaue Approximation der Daten möglich. Die absoluten Fehler zu Abbildung 3.4, sind in den oberen beiden Teilbildern in Abbildung 3.5 dargestellt. Man sieht, dass die größten Fehler bei Methode B nach den Sprüngen im Systemeingang entstehen. Eine Erklärung dafür ist, dass die Werte an diesen Stellen von den Trainingsdaten unterscheiden. Bei Methode A, zeigen sich die größten Abweichungen zu Zeiten mit konstanter Eingangsgröße, welche naturgemäß nicht durch die Trainingsdaten, bestehend aus random walks, abgedeckt ist. Insgesamt beobachten wir, dass die 1-Schritt Simulation B von Messdaten deutlich genauere Schätzungen liefert. Das ist nicht überraschend, da das Netzwerk für die Schätzung des nächsten Schrittes, für Methode A als auch B, nur die letzten vier Zeitschritte in Betracht zieht. Wobei für Methode A diese Werte bereits Ausgänge des Netzes sind. Eine Abweichung, der vorherigen Ausgänge von den Daten kann daher von Methode A nicht wieder ausgeglichen werden. Davon ist die Methode B nicht betroffen, da vorherige Ausgänge des Netzes die nächsten Schätzungen nicht beeinflussen. Dies spiegelt sich besonders deutlich in der Darstellung der kumulativen Fehler im untersten Teilbild wider.

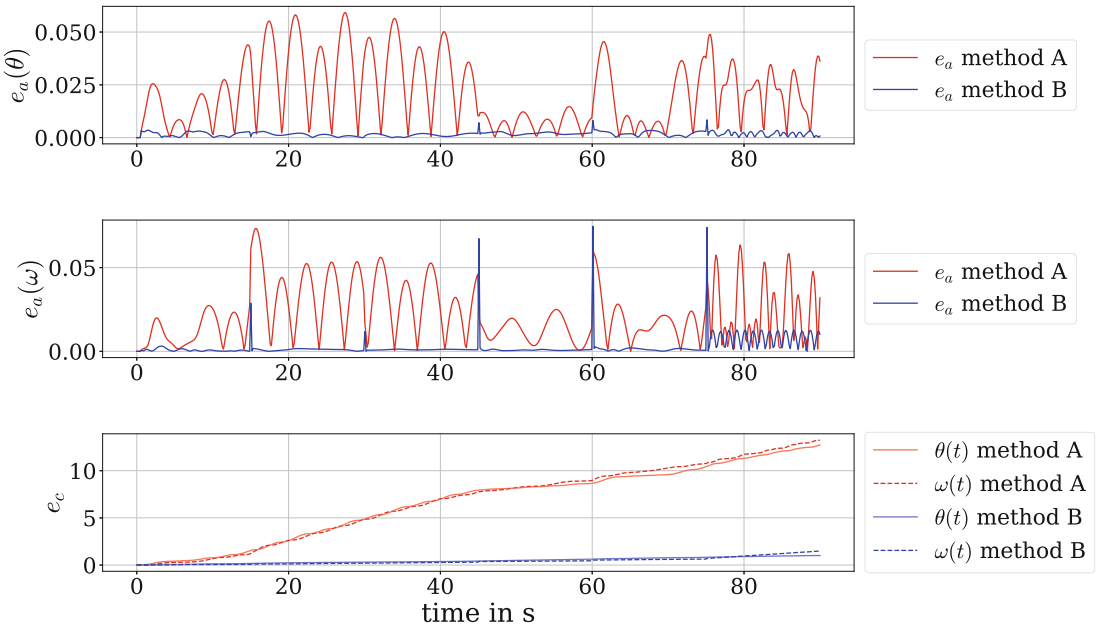

Abbildung 3.5: Fehler bei Pendel Simulation: Hier sind der absolute und der kumulative Fehler zwischen den Daten und dem, mit Methode A und B erzeugten, Netzwerkausgang bei der in Abbildung 3.4 dargestellten Simulation mit gemischtem Eingang, dargestellt.

Um diesen Abschnitt abzuschließen, wird ein Experiment durchgeführt, bei dem die Methoden B und C verglichen werden. Dabei interessiert uns, welche der Methoden besser abschneidet, wenn die verwendeten Daten verrauscht sind. Bei realen Systemen kann man davon ausgehen, dass gemessene Daten immer mit Messrauschen behaftet sind. Wir können das dadurch simulieren, indem die Trainingsdaten mit einem normalverteilten Rauschen der Form

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} \tilde{\theta} + e_1 \\ \tilde{\omega} + e_2 \end{bmatrix},$$

wobei $e_1, e_2 \sim \mathcal{N}(0,\, 1)$ gilt, belegt werden.

Wenn wir die drei oben vorgestellten Methoden zur Approximation vergleichen, ist für die Approximationsgüte zu erwarten, dass Methode B besser als C ist und Methode C wiederum besser als Methode A. Mit Methode B ist besser als Methode C ist gemeint, dass die Methode B bessere Ergebnisse, das heißt, einen im Durchschnitt kleineren MSE liefert, als Methode C. Die oben angegebene Reihung der Methoden gilt, da ein geschätzter Wert des Netzes nie genauer sein kann als der tatsächliche Messwert. Methode C kann auch nicht besser sein als B, solange die Testdaten mit dem gleichen Rauschen wie die Trainingsdaten behaftet sind.

Das folgende Beispiel, soll nun eine Konstellation aufzeigen, in der Methode C einen im Durchschnitt geringeren MSE als Methode B erzielt. Dazu wird ein neuronales Netz

auf Basis von Trainingsdaten, bei denen e_1 und e_2 in der gleichen Größenordnung sind, trainiert. An realen Systemen gilt oft $e_1 < e_2$, da die Geschwindigkeit meist, nicht direkt gemessen werden kann. Diese muss dann, aus einer Positionsmessung berechnet werden, wodurch sich bereits vorhandene Messfehler verstärken. Um diese Auswirkung zu verbildlichen, ist in Abbildung 3.6, die Verteilung der Fehler, für eine Berechnung dargestellt, bei der $e_2 = 5e_1$ gilt, also deutlich von den Rauschverhältnissen in den Trainingsdaten abweicht. Dazu wurden 400 Eingangssignale und zufällige Anfangswerte erzeugt und mit den Methoden B und C über 90 Sekunden ausgewertet.

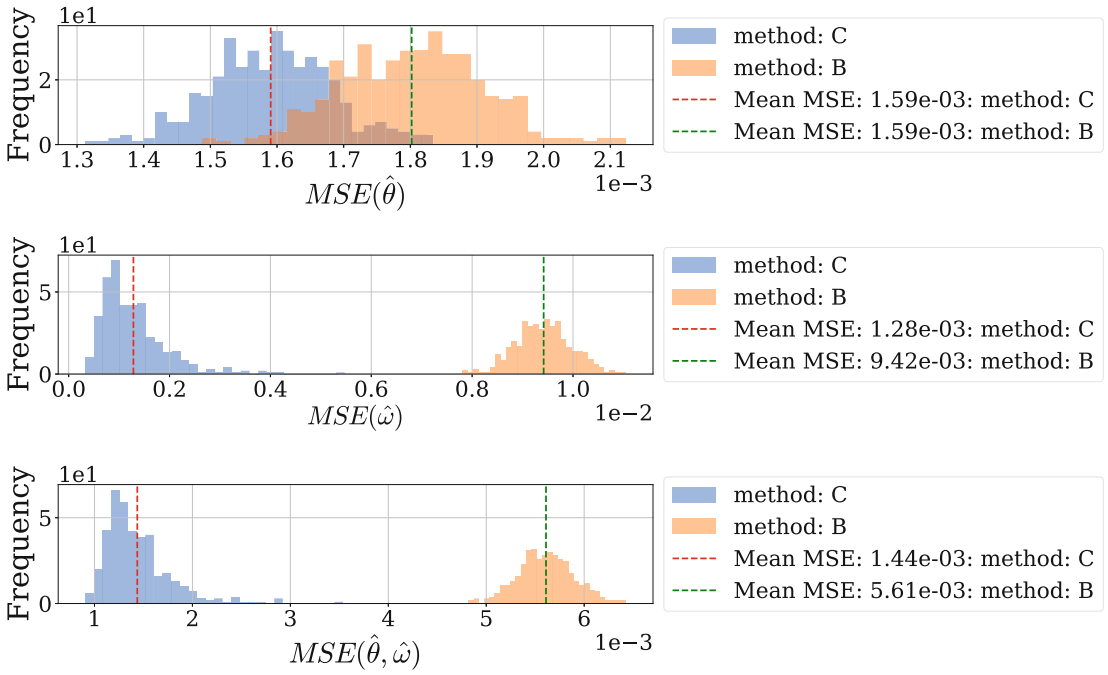

Abbildung 3.6: Histogramm zur Fehlerverteilung von 400 Schätzungen mit unterschiedlichem Rauschen, zwischen den Methoden B und C.

Wir sehen in Abbildung 3.6, dass bei diesem Szenario, der durchschnittliche MSE von Methode C, kleiner als der von Methode B ist. Es gilt also tatsächlich, dass Methode C in diesem Fall besser als Methode B abschneidet. Beim Arbeiten mit Messdaten, die in unterschiedlicher Qualität vorhanden sind, ist es durchaus ratsam, die beiden Methoden zu vergleichen.

Bemerkung. Abschließend sei noch angemerkt, dass in den meisten Fällen Methode B am besten abschneidet. Das vorgestellte Experiment ist so konstruiert, um die Möglichkeit hervorzuheben, dass Methode C manchmal sogar geringere durchschnittliche Fehler erzeugt.

4 Modellierung eines Pneumatikventils mit neuronalen Netzen

Das zentrale dynamische System, das wir in dieser Arbeit modellieren, ist ein Pneumatikventil vom Typ VEVM [41] der Firma Festo SE & Co. KG. Für eine genaue mathematische Beschreibung siehe [42, 43] und [44].

4.1 Beschreibung des Pneumatik Ventils

Das Ventil ist zweistufig ausgeführt, bestehend aus Vor und Hauptstufe, wie in Abbildung 4.1 dargestellt ist. Das Besondere dabei ist, dass die pneumatische Hauptstufe durch eine pneumatische Vorstufe aktuiert wird. In einem Ventilblock sind vier Haupt- und Vorstufen verbaut. Eine Vorstufe für einen Hauptstufenstößel, besteht aus zwei Piezos, die Vorstufen eines Ventilblocks beinhalten acht davon. Diese können durch piezoelektrisch aktuierte Balken geschlossen beziehungsweise geöffnet werden. Die Hälfte der Ventile sind *Normally Closed* (NC) und die andere Hälfte ist *Normally Open* (NO). Jeweils zwei dieser Ventile sind über pneumatische Halbbrücken zusammengeschaltet. Ihr Ausgangsdruck bildet die Stellgröße für eines der vier pneumatisch aktuierten Ventile der Hauptstufe. Über die Auslenkung des Ventilstößels ist der Durchfluss der Hauptstufe zum Arbeitsanschluss einstellbar. Weiters ist das Ventil mit Sensoren versehen, die den Zylinderhub als auch den Druck in einem Ventil der Hauptstufe messen können. Die Motivation hinter dieser Konstruktion ist, die, im Vergleich zu elektromagnetisch angesteuerten Schaltventilen geringe Lärmentwicklung im Betrieb und den im Vergleich zur elektromagnetischen Direktansteuerung eines (Servo)Ventilstößels sehr geringen Stromaufnahme der Piezoelektronik. Der durch die Vorstufe resultierende Luftverbrauch, ist natürlich höher als bei Schaltventilen, aber durch die kontinuierliche Positionierung des Hauptventilstößels in Relation zur meist hohen Leckage von klassischen Servoventilen zu stellen.

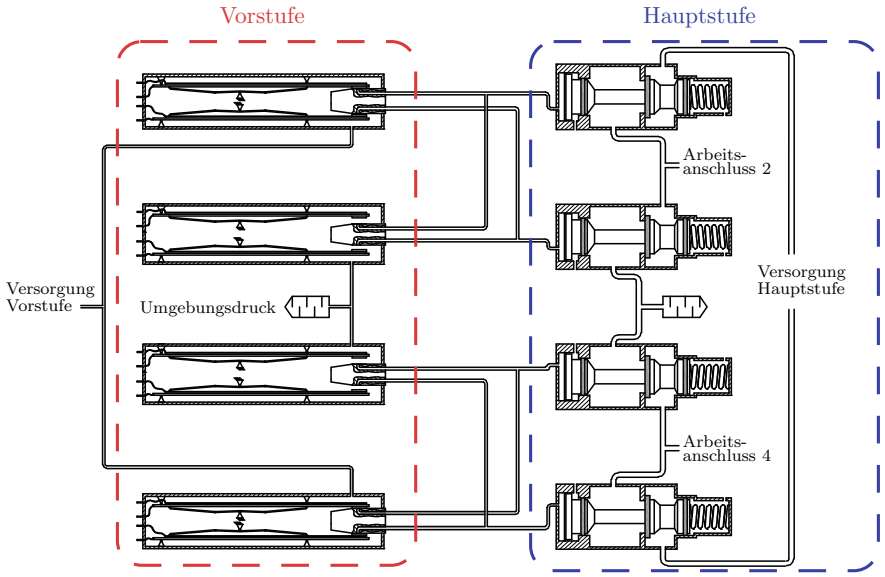

Abbildung 4.1: Übersichtskizze von Vor- und Hauptstufe des Pneumatikventils, siehe [43, Abbildung 2.1]. Links ist die Vorstufe mit den acht piezoelektrisch aktuierten Balken und rechts die Hauptstufe mit vier Ventilstößeln dargestellt.

Wie Abbildung 4.1 zu entnehmen ist, sind die vier Ventilstößel der Hauptstufe gleichartig aufgebaut. Daher beschränken wir uns im Folgenden auf eine Ventileinheit bestehend aus zwei Ventilen der Vorstufe und dem zugehörigen Ventil der Hauptstufe und betrachten diese als geschlossenes System. Die Funktion von der Ansteuerung der Vorstufe bis zum gewünschten Effekt an der Hauptstufe ist wie folgt. An die beiden Piezos werden zwei voneinander unabhängige Spannungen gelegt. Eine Kontraktion der Piezokeramik, die auf den Balken angebracht ist, biegt den jeweiligen Balken und öffnet damit die NC-Piezoventile oder schließt die NO-Piezoventile kontinuierlich. Durch den Versorgungsdruck auf dem NC-Piezoventil entsteht ein Massenstrom in die Vorkammer der Hauptstufe, während eine offenes NO-Piezoventil einen Luftstrom von der Vorkammer zum Abluftauslass erlaubt. Bei insgesamt positiven Volumenstrom in die Vorkammer, steigt der Druck und der Ventilstößel wird gegen die Rückstellfeder gedrückt. Die Auslenkung des Ventilstößels erhöht die Öffnungsfläche des Ventils der Hauptstufe und somit kann am Arbeitsanschluss Luft von der Versorgung über das Ventil zum Arbeitsanschluss, oder bei den in der Mitte abgebildeten Hauptstufen vom Arbeitsanschluss an den Abluftanschluss, strömen.

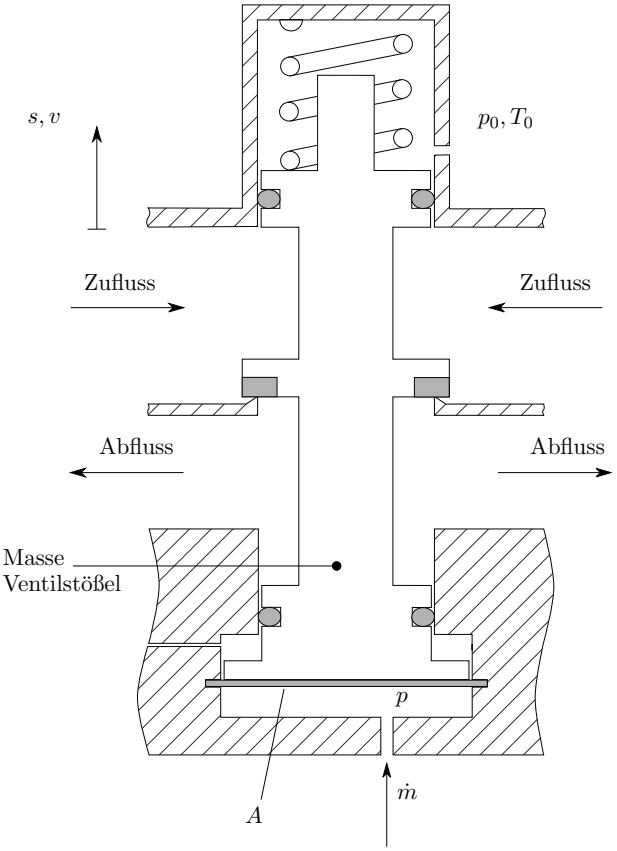

Abbildung 4.2: Darstellung eines Ventils der Hauptstufe, mit Rückstellfeder in entspannter Position. Der Massenstrom $\dot{m}$ in die Vorkammer, sowie der, durch Auslenkung des Ventilstößels, resultierende Luftstrom der Hauptstufe sind durch Pfeile angedeutet. Die Darstellung wurde auf Basis von [43, Abbildung 3.4] erstellt.

Wir beschränken unsere mathematische Beschreibung auf das System an Differentialgleichungen, die die Funktionalität der Hauptstufe, die in Abbildung 4.2 dargestellt ist, beschreiben. Die Bewegungsgleichungen der Hauptstufe sind einerseits leichter darzustellen als die der piezoelektrischen Bauteile und sind für uns wichtiger, da sie das System für unser zentrales Experiment in Kapitel 4.3 darstellen. Die dafür relevanten Größen sind der Druck p als Systemeingang, die Position $s = s(t)$ und die Geschwindigkeit $v = v(t)$ des Ventilstößels. Diese werden durch die Differentialgleichungen

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}s &= v, \\ \frac{\mathrm{d}}{\mathrm{d}t}v &= \frac{1}{m}(A(p-p_0) - c(s-s_0) + f_r(v) + f_k(s,v)), \end{aligned} \tag{4.1}$$

mit $m, A, s_0, p_0 \in \mathbb{R}^+$, beschrieben. Die Reibkraft und die Kontaktkraft, die die Bewegung des Ventilstößels beeinflussen, werden durch $f_r(v)$ und $f_k(s, v)$ abgebildet.

Wird die Vorstufe mit einbezogen, so wird der Druck zu einem Zustand des Systems, der wiederum von dem Massenstrom in die Vorkammer abhängt. Der Massenstrom und der Druckaufbau in der Vorkammer sind durch,

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t} m &= \dot{m}_{in}(u_{NC}) - \dot{m}_{out}(u_{NO}), \\ \frac{\mathrm{d}}{\mathrm{d}t} p &= \frac{\kappa}{V(s)}(-Avp + R_s T_0 \dot{m}), \end{aligned} \tag{4.2}$$

beschrieben. Dabei ist Änderung des Massenstroms durch die Differenz von $\dot{m}_{in}(u_{NC})$ und $\dot{m}_{out}(u_{NO})$ gegeben, wobei die Änderung des Zuflusses $\dot{m}_{in}(u_{NC})$ und die Änderung des abfließenden Gases, jeweils von den Spannungen u_{NC} und u_{NO}, die die Öffnung der Piezoventile regulieren, abhängig. In Kombination mit den Gleichungen (4.1), erhalten wir dadurch die Systemgleichungen für das gesamte System mit den unabhängigen Spannungen u_{NC} und u_{NO} als Eingang und Druck $p = p(t)$, Position $s = s(t)$ und Geschwindigkeit $v = v(t)$ als Zustände.

Auf die genauen Beschreibungen der Formeln für $\dot{m}_{in}(u_{NC}), \dot{m}_{out}(u_{NO}), V(s), f_r(v)$ und $f_k(s, v)$ werden wir nicht eingehen und verweisen auf [42] und [44]. Alle übrigen Parameter sind in Tabelle 4.1 zusammengefasst.

Parameter	**Beschreibung**
κ	Isentropenexponent (Wärmekapazitätsverhältnis)
A	effektive Fläche über welche der Vorkammerdruck auf den Ventilstößel drückt
R_s	spezifische Gaskonstante
T_0	Temperatur des Gases (konstant)
p_0	Umgebungsdruck
c	Federkonstante
s_0	entspannte Position der Feder

Tabelle 4.1: Beschreibung physikalischer Parameter für Gleichungen (4.1)

Bemerkung. An dieser Stelle sei nochmals betont, dass die obige mathematische und physikalische Beschreibung, einzig dem Verständnis des zugrundeliegenden Systems dient. Formal wird das Pneutikventil, sowohl in der Modellierung der Hauptstufe, als auch in der Modellierung des Gesamtsystems, immer als Black-Box Modell behandelt. Unser Vorgehen ist also grundsätzlich unabhängig von der obigen Beschreibung.

Wir erhalten also zwei dynamische Systeme, die in Abbildung 4.3 dargestellt sind. Die Hauptstufe mit dem Systemeingang p und der Position s und der Geschwindigkeit v des Ventilstößels als Systemzustände auf der linken Seite. Auf der rechten ist das gesamte System, mit den Spannungen u_{NC} und u_{NO} als Systemeingang und dem Druck p, der Position s und der Geschwindigkeit v als Systemzustände dargestellt. Der Massenstrom, welcher auch eine Zwischengröße hätte bilden können, wurde weggelassen.

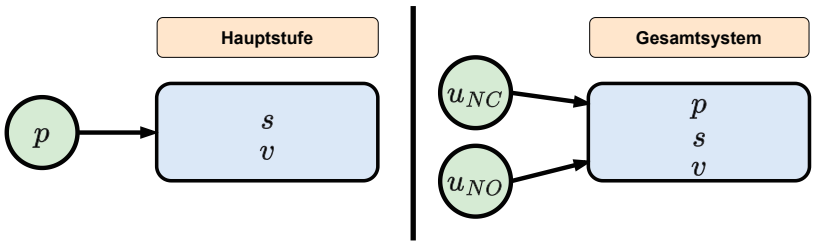

Abbildung 4.3: Überblicksdarstellung der Aufteilung des Pneumatikventils in zwei dynamischen Systeme. Links die Hauptstufe und rechts das gesamte Ventil.

4.2 Training der neuronalen Netze

In diesem Abschnitt werden Einzelheiten zum Training der neuronalen Netze angegeben. Dabei werden die Aufbereitung von Trainingsdaten, technische Details der Implementierung, sowie Soft- und Hardware Spezifikationen beschrieben.

4.2.1 Details zu Trainingsdaten und Beschreibung des Trainingszyklus

Daten für das Training der neuronalen Netze in dieser Arbeit werden auf zwei verschiedene Arten erzeugt. In dieser Arbeit werden Daten, als *synthetisch* bezeichnet, wenn diese durch eine Simulation erzeugt wurden. Das sind Daten, die beispielsweise durch numerische Lösungsverfahren von Differentialgleichungen durch Modelle in MATLAB\SIMULINK erzeugt werden. Alle Daten, welche mit Messgeräten aufgenommen wurde, also Vorgänge an realen System darstellen, werden als *Messdaten* bezeichnet. Der Vorteil synthetischer Daten liegt darin, dass sie in großer Menge, mit relativ geringem Aufwand, im Wesentlichen nur durch die Rechenzeit beschränkt, erzeugt werden können, sofern ein akkurates Modell des Systems verfügbar ist. Die Voraussetzung eines bestehenden Modells ist meist unrealistisch, wenn die Daten zum Training eines neuronalen Netzes verwendet werden sollen, um ein entsprechendes Systemmodell zu lernen. In manchen Fällen kann ein solches neuronales Netz allerdings kürzere Rechenzeiten für Simulationen erreichen. Im Sinne der Blackbox-Modellierung ist es naheliegend, Messdaten eines Systems als Trainingsdaten zu verwenden. Aber auch hier gibt es Hürden, wie beispielsweise, dass deren Erhebung häufig zeit- und kostenintensiv ist und Messfehler, die Qualität der Daten stark beeinträchtigen können. Zudem lassen sich manche physikalischen Größen nur schwer oder gar nicht zuverlässig direkt messen.

Bemerkung. Da Daten aus realen Systemen meist unterschiedliche Einheiten und damit verschiedene Größenordnungen besitzen, ist es sinnvoll, die Daten vor der Verwendung als Trainingsdaten zu normieren. In dieser Arbeit wurde stets die Min-Max-Normalisierung, der Form

$$x^* = \frac{x - x_{min}}{x_{max} - x_{min}},$$

zur Transformation auf ein definiertes Intervall verwendet. Die Größen x_{min} und x_{max} sind dabei Eigenschaften des Systems, typischerweise Systemgrenzen wie der maximale

Zylinderhub oder die minimale Spannung für das, im Abschnitt 4.1 beschriebene, Pneumatikventil. Alternativ können die Maxima und Minima auch aus den Daten ermittelt werden, dies birgt aber die Gefahr, dass das definierte Intervall verlassen wird, wenn nicht der gesamte mögliche Wertebereich in den Daten vorhanden ist.

Die Menge und Qualität der Daten sind entscheidend für den Erfolg des Trainings eines neuronalen Netzes. Besonders bei dynamischen Systemen ist es wichtig, dass eine möglichst große Bandbreite an Systemeingängen, Anfangszuständen und Zustandsverläufen abgedeckt wird. Für die in dieser Arbeit betrachteten Systeme haben sich Eingangssignale, welche mit Random Walk Algorithmen, oder einem Zufallstrajektoriengenerator, der in [45] beschrieben wird, generiert wurden, als zielführend erwiesen. Letzterer erzeugt eine Trajektorie, die aus einer beliebigen Anzahl an konstanten und nichtlinearen Teilstücken besteht. Um einen Datenpunkt für das Training zu erzeugen, werden die Systemzustände $x(t)$, zu einem Einganssignal $u(t)$, aufgezeichnet und zusammengefasst. Die genaue Unterteilung der Trainingsdaten $d(t) = [u(t), x(t)]^T$ in Netzeingang und Zielwert wird später besprochen. Der Pseudocode in Algorithmus 4 bietet eine detailliertere Darstellung des Ablaufs des Trainings und enthält Maßnahmen, die das Konvergenzverhalten verbessern. Dafür wird ein *Learning rate scheduler*, *mini-Batch Learning* und *Gradient Clipping* verwendet.

Algorithm 4: Training Loop

input : data, $\mathcal{N}$, epochs
output: $\mathcal{N}$

```
1  begin
2      Optimizer = Adam
3      LearningRateScheduler = ExponentialDecay()
4      NormalizedData = MinMaxNormalization(data)
5      BatchedData = DataLoader(NormalizedData)
6      LossFunction = MSE()
7      for i = 1 to epochs do
8          for d, y in DataLoader do
9              output = N(x)
10             loss = LossFunction(output, y)
11             loss.backward() #  calculate gradient
12             N.GradientClipping()
13             Optimizer.step()
14         end
15         LearningRateScheduler.step()
16     end
17     return N
18 end
```

Der Learning rate scheduler, verringert nach jeder Epoche die Lernrate. Dafür wird oft ein exponentieller Abfall der Form

$$\lambda_{new} = \lambda_{initial} \cdot \mathrm{e}^{-k \cdot t},$$

für die neue Lernrate zur Epoche t mit dem Skalierungsfaktor k gewählt.

Weiters ist es zielführend, die Daten in sogenannte *Batches*, in diesem Fall Teilmengen gleicher Größe aus zufällig ausgewählten Datenpunkten (Eingang, Zustand, Ausgang), aufzuteilen. Sind die Daten durch Zeitreihen dargestellt, so muss die zeitliche Reihenfolge bei der Bildung der Batches eingehalten werden. Wie in Abschnitt 2.1 beschrieben wird, ist es üblich bei Zeitreihen ein Fenster von Werten als Eingang ins Netz zu verwenden. Dafür wird in dieser Arbeit ein *sliding window* mit Länge ws verwendet, um eine Zeitreihe $d(t)$ in Eingänge und zugehörige Zielwerte $y(t)$ zu unterteilen. Um ein Batch aus unseren Daten auszuwählen, definieren wir zunächst für eine $n-$elementige Menge von Zeitreihen $X = \{d_j(1), \dots, d_j(T),\ j = 1, \dots, n\}$, die Menge

$$\mathcal{B}_X = \left\{ \left(\begin{bmatrix} u(i), \dots, u(i+ws) \\ x(i), \dots, x(i+ws) \end{bmatrix} \mid x(i+ws+1) \right), \quad i = 1, \dots, T-ws-1, \quad \begin{bmatrix} u \\ x \end{bmatrix} \in X \right\}.$$

Ein Batch der Größe n, ist dann eine zufällig ausgewählte n-elementige Teilmenge $B_n \subseteq \mathcal{B}_X$. Dabei ist der Zielwert zu einem Zeitfenster $d(i), \dots, d(i+ws)$, durch den Systemzustand zum nächsten Zeitpunkt, $x(i+ws+1)$, gegeben. Ein ganzes Batch kann parallel berechnet werden und mit dem daraus resultierenden Fehler wird eine Aktualisierung der Parameter durchgeführt. Bei sehr großen Datenmengen ist es ineffizient vor jedem Update der Parameter die gesamten Trainingsdaten auszuwerten, den Gradienten jedoch an nur einem Datenpunkt zu berechnen, liefert hingegen eine zu ungenaue Näherung. Das *mini-Batch Learning* ist ein effizienter Mittelweg, bei dem nach einer Auswertung des Netzes auf einem relativ kleinen Batch, also $n \ll |\mathcal{B}_X|$, eine Aktualisierung der Parameter durchgeführt wird. Dabei werden einerseits Vorteile der stochastischen Approximation des Gradienten, als auch der Parallelisierbarkeit der Berechnungen genutzt.

Bemerkung. Beim Training eines Netzes mit OR wird bereits beim Training, die gesamte Länge der Trajektorie, in Abhängigkeit der Länge T des Systemeingangs $u(t)$ berechnet. Dafür wurden Batches der Form,

$$B_m^{\text{OR}} \subseteq \mathcal{B}_X^{\text{OR}} = \left\{ \left(\begin{bmatrix} u(1), \dots, u(ws), \dots, u(T) \\ x(1), \dots, x(ws), \dots, 0 \end{bmatrix} \mid [x(ws+1), \dots, x(T)] \right), \quad \begin{bmatrix} u \\ x \end{bmatrix} \in X \right\},$$

verwendet. Dem Netz wird also nur ein Anfangswert und die Systemeingänge vorgegeben, da das Netzwerk seine eigenen Ausgänge als zukünftige Eingänge nutzt. Dennoch wird in einem forward pass trotzdem nur ein zukünftiger Zeitschritt berechnet, wobei die Iteration bis zum Zeitpunkt T innerhalb des Netzes geschieht. Daher ist es besonders wichtig, dass die Daten eine große Bandbreite an Anfangswerten enthalten, da diese nicht wie bei den Batches beim Training ohne OR, an zufälligen Stellen gesetzt werden. Die Batch Größe m, ist in der Regel viel kleiner als n, da für ein Datum in einem eben beschriebenen Batch das Netzwerk $(T-ws)$ mal ausgewertet werden muss, während es bei einem Batch B_n für das Training ohne OR nur n mal ausgewertet wird.

Bevor die Aktualisierung der Parameter durchgeführt wird, wird der Gradient beim *Gradient Clipping* normiert. Durch

$$\hat{\nabla}_\theta = \nabla_\theta \frac{c}{\|\nabla_\theta\|},$$

erhalten wir einen Gradienten, der die von uns gewählte Größe c nicht übersteigt.

4.2.2 Beschreibung der verwendeten Software und Hardware Komponenten

In Tabelle 4.2 sind die relevante Software und Hardware Spezifikationen angeführt. Dabei ist besonders die Bibliothek PyTorch hervorzuheben. Diese ermöglicht eine einfache Implementierung diverser Netzwerkarchitekturen und ist neben Keras und TensorFlow eine der populärsten Bibliotheken für die Entwicklung neuronaler Netze. PyTorch verfügt außerdem über die Schnittstelle mit CUDA, die eine einfache Durchführung von Berechnungen auf der GPU erlaubt, wodurch Rechenzeiten deutlich verkürzt wurden. Sämtlicher Code, der für diese Arbeit geschrieben wurde, steht unter [46] zur Verfügung.

Table 4.2: Software und Hardware Spezifikationen

Spezifikationen	**Beschreibung**
Software Spezifikationen	
Progammiersprache	Python 3.12.4
Machinelles Lernen Bibliothek	PyTorch 2.3.1
Numerik Bibliotheken,	NumPy 1.26.4, Pandas 2.14
Betriebssystem	Linux
Hardware Spezifikationen	
Prozessor	Intel Core i7-10700K
GPU	NVIDIA GeForce RTX 2080 Ti
CPU	Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz
CUDA Version	11.5

4.3 Simulationsstudie an der Hauptstufe

In diesem Abschnitt beschreiben wir die durchgeführten Experimente, um festzustellen, welche Art von Netzwerkarchitektur von neuronalen Netzen am besten geeignet ist, ein dynamisches System zu simulieren, und vergleichen zwei Trainingsmethoden.

4.3.1 Aufbau der Experimente

Zu betonen ist, dass die Ergebnisse der folgenden Experimente nur für die gegebenen Konfigurationen, siehe Tabelle 4.3, gelten. Im Allgemeinen ist es ziemlich schwierig, endgültige Schlussfolgerungen aus Experimenten zu ziehen, wenn man neuronale Netzwerke vergleicht, da viele Faktoren, wie beispielsweise.

- die Menge der Trainingsdaten,
- die Qualität der Trainingsdaten,
- die Batch Size,
- die Netzwerkgröße,
- die Anzahl der Layer,
- die Lernrate,
- die Art des Optimierers oder auch
- die Zufälligkeit bei der Initialisierung,

die Ergebnisse beeinflussen können. Dem kann mit einer Optimierung der Hyperparameter der zu vergleichenden Netze entgegenwirken und mit Verwendung von *Benchmark Datasets* beim Testen. Das sind, oft sehr umfangreiche, frei zugängliche Datensätze, auf denen die Genauigkeit von Netzen verglichen werden kann.

Zusätzlich sollte beachtet werden, dass die durchgeführten Experimente nur ein spezifisches dynamisches System betrachten. Daher gibt es keine Garantie, dass die folgenden Ergebnisse auch für andere Systeme gelten. Da jedoch viele dynamische Systeme ein ähnliches Verhalten zeigen, können die Ergebnisse für die beste Netzwerkarchitektur und Trainingsmethode als ein guter Ausgangspunkt betrachtet werden, auch um andere dynamische Systeme zu untersuchen.

Die folgenden Tabellen, 4.3 und 4.4 geben einen Überblick über die in den Experimenten verwendeten Hyperparameter und über einige für das Training relevante Daten. Die hidden Size beschreibt die Dimension, also die Anzahl an Neuronen in einem hidden Layer des Netzes. Die Anzahl der Layer für das MLP entspricht der Anzahl der hintereinander geschalteten LSTM-Zellen beziehungsweise der Level in einem TCN. Anzahl der Kanäle und Größe des Filters sind TCN spezifische Größen. Die Fensterlänge ist eine Größe, die sich bei LSTM und TCN auf die Länge Anzahl der aufeinanderfolgenden Zeitschritte in einem Netzeingang bezieht. Die Angabe, einer Fensterlänge könnte man erhalten, indem man die Dimension des Eingangs durch die Anzahl der unterschiedlichen Eingangsgrößen dividiert, wie in Abbildung 2.4 dargestellt ist.

Model types	**LSTM**	**MLP**	**TCN**
Hidden Size	8	24	-
Number of Layers/Cells/Levels	3	3	4
Channels	-	-	5
Kernel Size	-	-	7
Window Size	16	-	30
Input Size	3	60	3
Output Size	2	2	2

Tabelle 4.3: Spezifikationen von Parametern der neuronalen Netze.

In der Tabelle 4.4 ist die Anzahl der Epochen, sowie die Lernrate und größe der Batches beschrieben. Eine Lernrate von $\lambda = 0.001$ ist eine typische Wahl, wobei die Lernrate ohnehin während des Trainings adaptiert wird, wie in Abschnitt 4.2.1 beschrieben wurde. Die Größe des Batches, bestimmt die Anzahl der Trainingsdaten, die von dem Netz ausgewertet werden, bevor eine Aktualisierung der Parameter, auf Basis des auf dem Batch approximierten Gradienten, durchgeführt wird. Die Anzahl der Epochen bestimmt die Anzahl der Iterationen über den gesamten Datensatz im Training.

Model typen	OR-LSTM-dt / OR-LSTM	OR-MLP-dt / OR-MLP	OR-TCN-dt / OR-TCN
Epochen	2000	2000	2000
Lernrate λ	0.0008	0.001	0.001
Batch größe : m (B_m^{OR})	20	20	20

Tabelle 4.4: Parameterspezifikationen für das Training von neuronalen Netzen mit OR.

Bei den Netzwerken ohne OR im Training wurde zusätzlich ein Pretraining auf einem kleinen Teil der Trainingsdaten durchgeführt. Außerdem wurden Trainingsläufe teilweise verfrüht abgebrochen, um overfitting der Trainingsdaten zu vermeiden. Die Größe der Batches für das Training ist in einer anderen Größenordnung wie jene beim Training der Netze mit OR, wie wir in Abschnitt 4.2.1 ausgeführt haben. Es wurde allgemein versucht, die Parameter recht ähnlich zu wählen, um ein gewisses Maß an Vergleichbarkeit zu erhalten.

Model Typen	LSTM-dt / LSTM	MLP-dt / MLP	TCN-dt / TCN
Epochen	2000	2000	2000
Lernrate λ	0.001 / 0.0008	0.001	0.001
pretrained	✓/ -	✓/ -	✓/ -
Batch größe : n (B_n)	2000	1000	2000

Tabelle 4.5: Parameterspezifikationen für das Training von neuronalen Netzen ohne OR.

Für jedes dieser Netzwerke vergleichen wir die folgenden Kategorien

A: Ableitungsschätzer mit OR,

B: direkte Schätzer mit OR,

C: Ableitungsschätzer und

D: direkte Schätzer.

4.3.2 Vergleich der Trainingsfehler

In diesem Abschnitt werden die Konvergenzen der verschiedenen Kategorien A, B, C und D aus Abschnitt 4.3.1 diskutiert. Dazu wurde nach jeweils 200 Epochen des Trainings eine Simulationsstudie mit 50 Trajektorien aus dem Testdatensatz durchgeführt. Die Trajektorien im Testdatensatz sind etwa fünfmal länger, als die in den Trainingsdaten. Die Ergebnisse sind in Abbildung 4.4 dargestellt. Das dargestellte Fehlermaß ist der durchschnittliche MSE zwischen den Ergebnissen der Simulation und den Daten. Es zeigt sich, dass, egal ob Ableitungsschätzung oder direkte Schätzung verwendet wurde, die Netze mit OR deutlich besser konvergieren. Die schnellste Konvergenzrate zeigt sich bei dem TCN Ableitungsschätzer mit Output Recurrence (OR-TCN-dt). Bei den Netzen ohne OR ist das Konvergenzverhalten durch Ausreißer geprägt. Zwar erzeugen diese Netze in den meisten Fällen gute Approximationen, jedoch sind die Mittelwerte von einzelnen Ausreißern stark verzerrt. Einige dieser Ausreißer wurden weggelassen, um eine übersichtliche Darstellung zu ermöglichen.

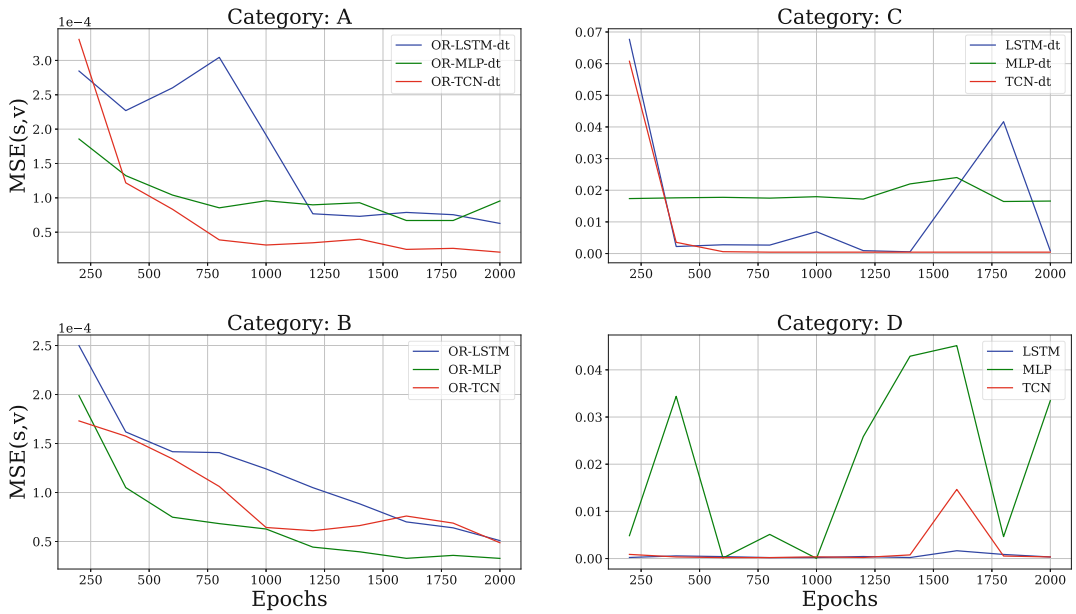

Abbildung 4.4: Konvergenz der unterschiedlichen Kategorien A-D über 2000 Epochen während des Trainings. Die linken beiden Teilbilder, in denen die Kategorien A und B dargestellt sind, zeigen ein deutlich stabileres Konvergenzverhalten als die Netze der Kategorien C und D unabhängig von der Netzwerkarchitektur.

4.3.3 Simulationsergebnisse für ausgewählte Eingangssignale

In diesem Abschnitt wird die Qualität der Vorhersagen der Netzwerke auf Testdaten, also Daten außerhalb der Trainingsdaten diskutiert. Dazu werden zunächst die Vorhersagen der unterschiedlichen Netze und der Kategorien A-D auf einzelnen Trajektorien verglichen. Die, für diese Versuche betrachteten, Konfigurationen bestehen dabei entweder aus allen Netzarchitekturen mit fixierter Kategorie, zu sehen in Abbildungen 4.5, 4.6, 4.7 und 4.8, oder aus allen Kategorien bei festgehaltener Netzwerkarchitektur, die in den Abbildungen 4.9, 4.10 und 4.11 dargestellt sind. Im Anschluss werden die Fehlerverteilungen über 100 Testtrajetkorien diskutiert.

Im ersten Vergleich wird eine Trajektorie betrachtet, die mit 56 ms fünfmal länger als die Zeitreihen der Trainingsdaten ist. Ziel dieses ersten Vergleichs, dargestellt in Abbildung 4.5, ist eine Bewertung, ob die Netze auch über längere Zeitspannen verlässliche Prädiktionen produzieren können.

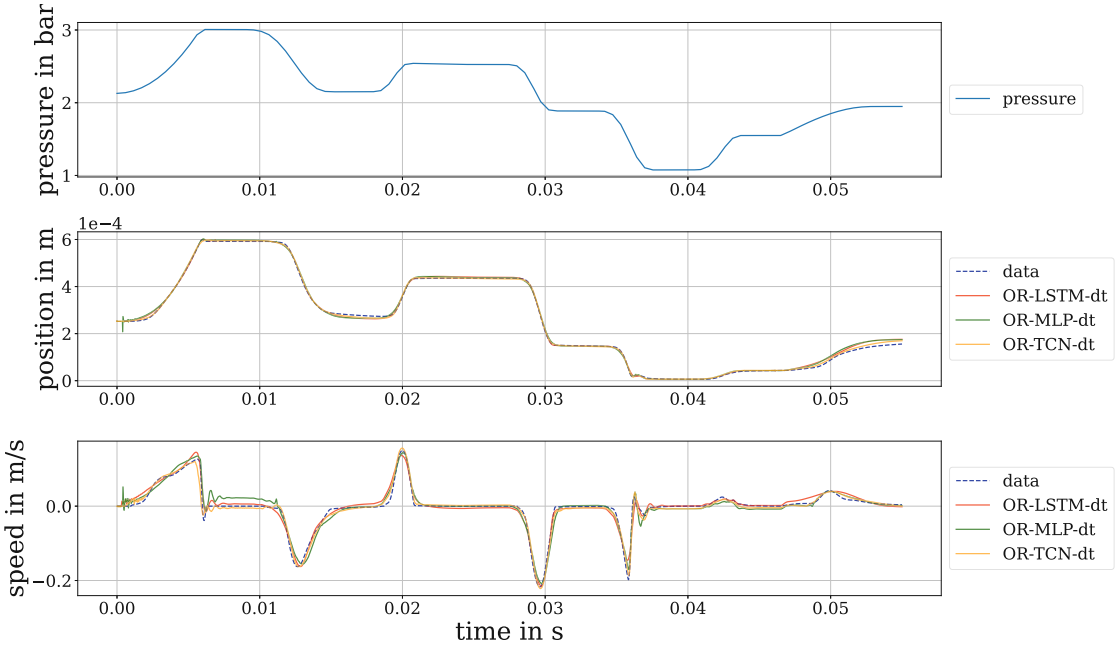

Abbildung 4.5: Simulationsergebnisse der, mit OR trainierten, Ableitungsschätzer für MLP, LSTM und TCN mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

Die Abbildung 4.5 zeigt die Ergebnisse für drei Ableitungsschätzer, die alle mit OR trainiert wurden. Im obersten Teilbild ist der Druckverlauf als Eingang des Systems dargestellt. Hierbei handelt es sich um eine Trajektorie aus dem Zufallstrajektoriengenerator. Darunter sind die Systemzustände der Schätzer zusammen mit den Testdaten, also den Zielgrößen für die Prädiktionen abgebildet. Alle Netzwerkarchitekturen produzieren Prädiktionen, die sehr nah an den realen Daten liegen. Auffällig ist eine Abweichung des MLPs, vor allem in den ersten 10 ms, die aber über die Dauer des Tests abnimmt. Das Netz mit den geringsten Abweichungen bei diesem Test ist das OR-TCN-dt. Es zeigt sich weiters, dass die Prädiktion der Geschwindigkeit stets größere Fehler, als die der Position aufweist. Dieses Phänomen wird sich auch bei den folgenden Abbildungen zeigen. Auch wenn dies hier nicht direkt als Erklärung genutzt werden kann, so kann über eine physikalische Interpretation des Systems dieses Phänomen untersucht werden. Aus den Gleichungen (4.1) zeigt sich, dass die Position, bis auf Unsicherheiten durch Reibkräfte, stationär direkt und eindeutig einem Druck zugeordnet werden kann. Die gilt jedoch nicht für die Geschwindigkeit. Somit können alle stationären Phasen in den Testdaten zum Training der Abbildung zwischen Druck und Position genutzt werden.

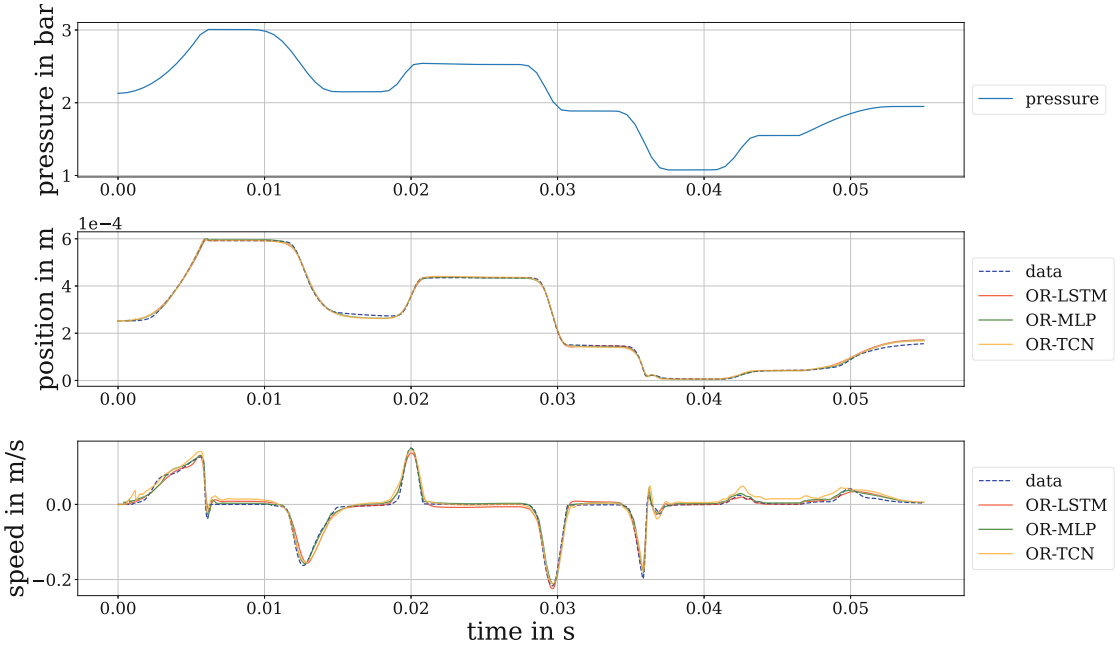

Abbildung 4.6: Simulationsergebnisse der, mit OR trainierten, direkten Schätzer für MLP, LSTM und TCN, mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

Die Auswertung desselben Eingangssignals mit den direkten Schätzern, welche mit OR trainiert wurden, ist in Abbildung 4.6 dargestellt. Die Ergebnisse weisen eine ähnlich geringe Fehler, wie jene der Ableitungsschätzer, auf. Der Vergleich von Abbildung 4.6 und 4.5 zeigt, dass bei dieser Konfiguration, keines der Netze einen signifikanten Vorteil gegenüber den anderen hat. Alle drei Netze können, aufgrund ihres Trainings mit eigenen Ausgängen, leichte Ungenauigkeit in den ersten Zeitschritten gut selbstständig ausgleichen. Nur das TCN direkter Schätzer mit Output Recurrence (OR-TCN) weißt in der Geschwindigkeit, zwischen 40 ms und 50 ms, leichte Abweichungen von den Daten auf. Ansonsten ist auch hier keine Verschlechterung der Genauigkeit der Prädiktionen bei fortgeschrittener Zeit zu erkennen.

Im Vergleich zu den mit OR trainierten Netzwerken, hat sich die Genauigkeit des LSTM und des MLP in Abbildung 4.7, besonders bei der Approximation der Geschwindigkeit, verschlechtert. Die Schätzung des TCN oszilliert stark und kann das langfristige Verhalten der Daten nicht abbilden. Besonders bei konstanten Teilstücken in Position und Geschwindigkeit ist es für das Netz nicht möglich diese richtig abzubilden. MLP und LSTM erreichen in der Position eine hohe Genauigkeit, allerdings ist bei manchen hohen Geschwindigkeitsänderungen, wie etwa bei 5 ms und 12 ms, keine genaue Approximation der Maxima und Minima der Geschwindigkeit möglich.

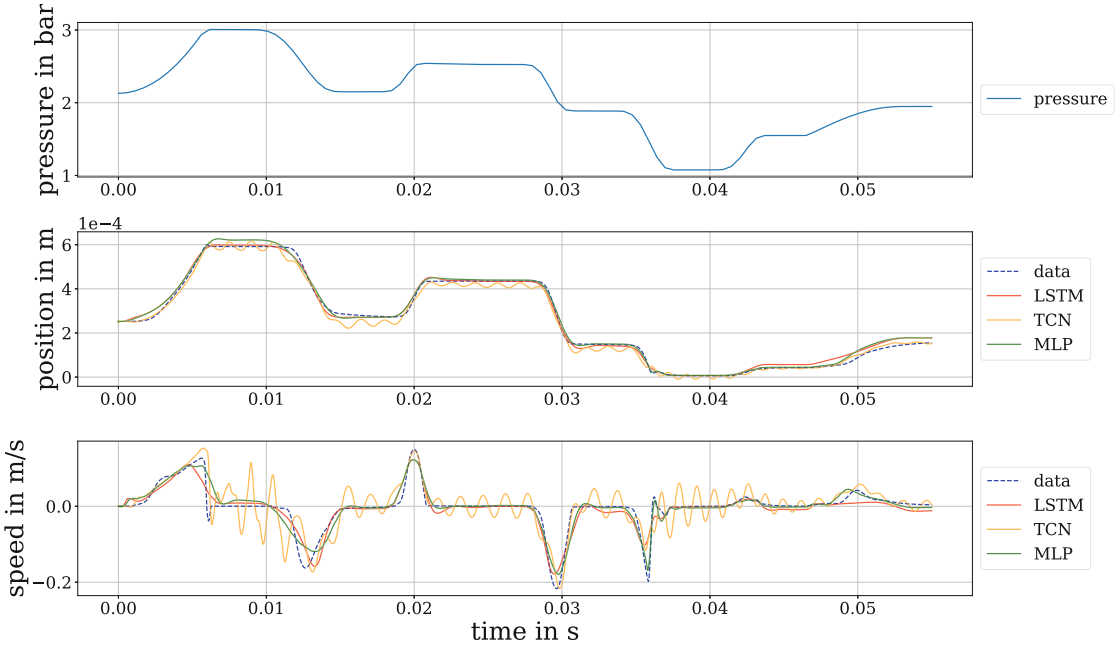

Abbildung 4.7: Simulationsergebnisse der direkten Schätzer ohne OR für die drei Netzwerkarchitekturen mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

Bei der, in Abbildung 4.8 dargestellten, Konfiguration fällt sofort die große Abweichung des MultiLayer Perceptron zur Ableitungsschätzung (MLP-dt) auf. Sowohl in Position als auch in der Geschwindigkeit ist keine zufriedenstellende Näherung möglich. Auch das LSTM-dt schwingt in der Position deutlich über die konstanten Teilstücke hinaus. Überraschend ist jedoch, dass das Temporal Convolutional Neural Network zur Ableitungsschätzung (TCN-dt) bis auf eine leichte Abweichung in Position und Geschwindigkeit bei nach etwa 6 ms eine hohe Genauigkeit erzielt.

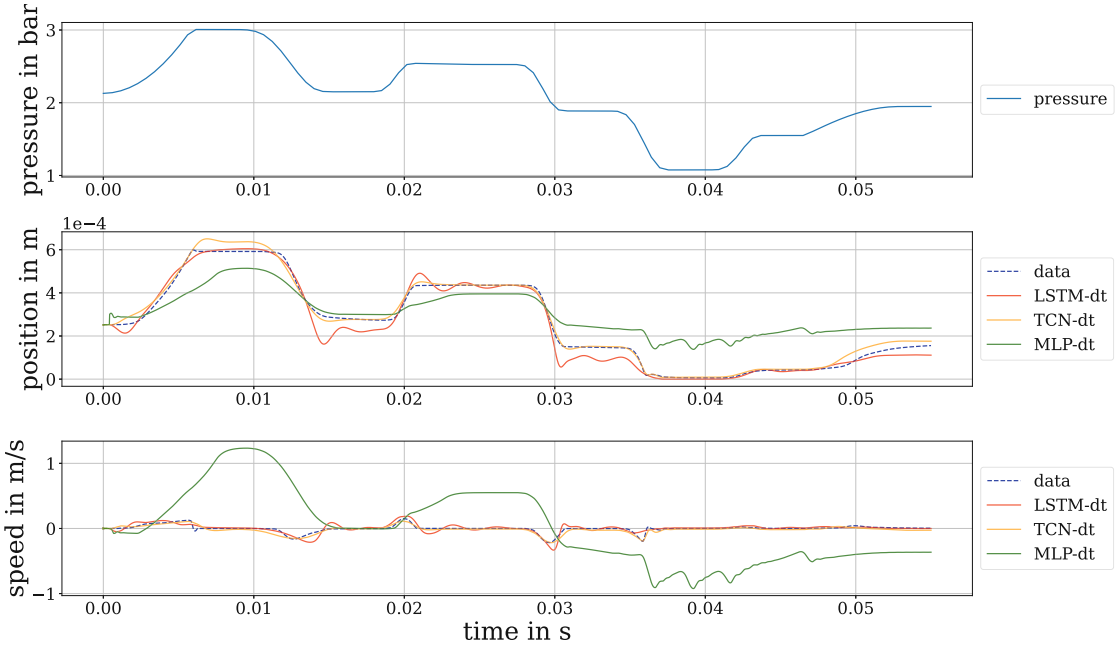

Abbildung 4.8: Simulationsergebnisse der Ableitungsschätzer für die drei Netzwerkarchitekturen mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

Aus den bislang diskutierten Ergebnisses ist zu erkennen, dass die Netze, die ohne OR trainiert wurden, allgemein größere Fehler aufweisen. Da wir bis jetzt aber nur Simulationsbeispiele betrachtet haben, können wir die Netze ohne OR noch nicht verwerfen. Keine der Netzwerkarchitekturen LSTM, MLP oder TCN hat deutlich genauere Prädiktionen als die anderen erzeugt, wobei das LSTM bei den Kategorien A-D immer eine akzeptable Genauigkeit aufweist und daher eine sichere Wahl einer Architektur darstellt.

Als Nächstes fixieren wir eine Netzwerkarchitektur und variieren die Kategorien A-D. Diese Konfigurationen sollen eine Anschauung dafür geben, wie groß die Unterschiede zwischen den einzelnen Kategorien innerhalb einer Netzwerkarchitektur sind. Der Druckeingang und die daraus resultierenden Systemzustände entsprechen denen, die in den obigen Versuchen verwendet wurden.

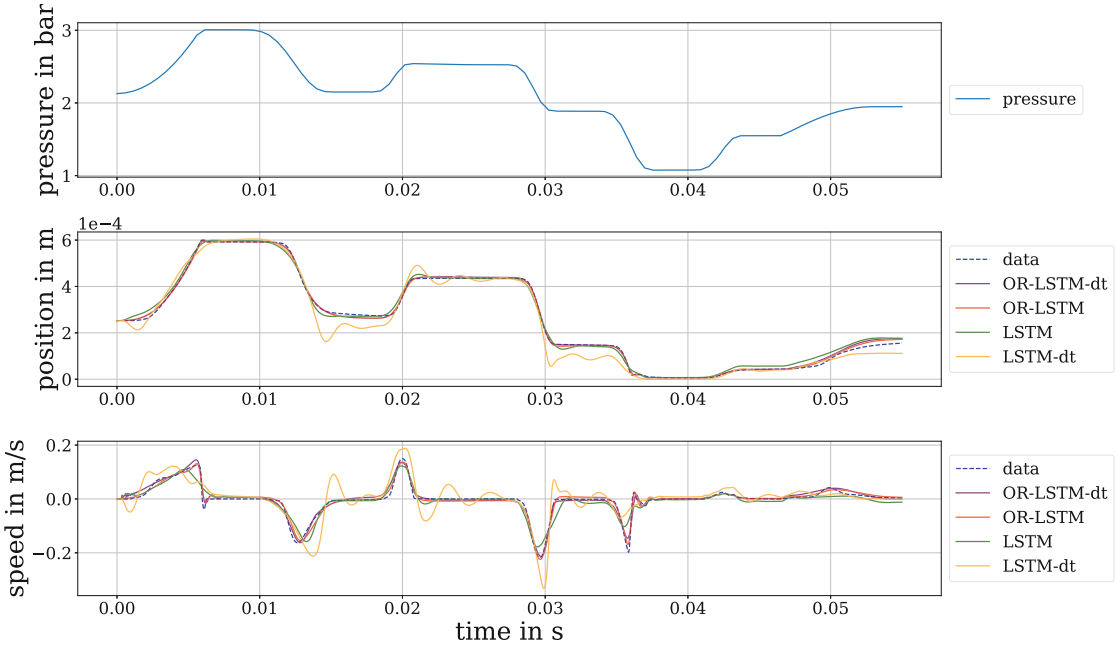

Abbildung 4.9: Simulationsergebnisse der unterschiedlichen Kategorien für das LSTM, mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

In Abbildung 4.9 wird die oben beschriebene Beobachtung, dass das LSTM unabhängig von der Kategorie, eine hohe Präzision erreicht, bestätigt. Dennoch ist klar zu erkennen, dass das LSTM-dt die geringste Genauigkeit hat, wie oben bereits beschrieben wurde. Es ist nicht überraschend, dass Netze, mit der LSTM Architektur, im Allgemeinen gut abgeschnitten haben, da sie, wie aus der Literatur bekannt ist, eine der weitverbreitetsten Netzwerkarchitekturen, für die Verarbeitung von Zeitreihendaten ist.

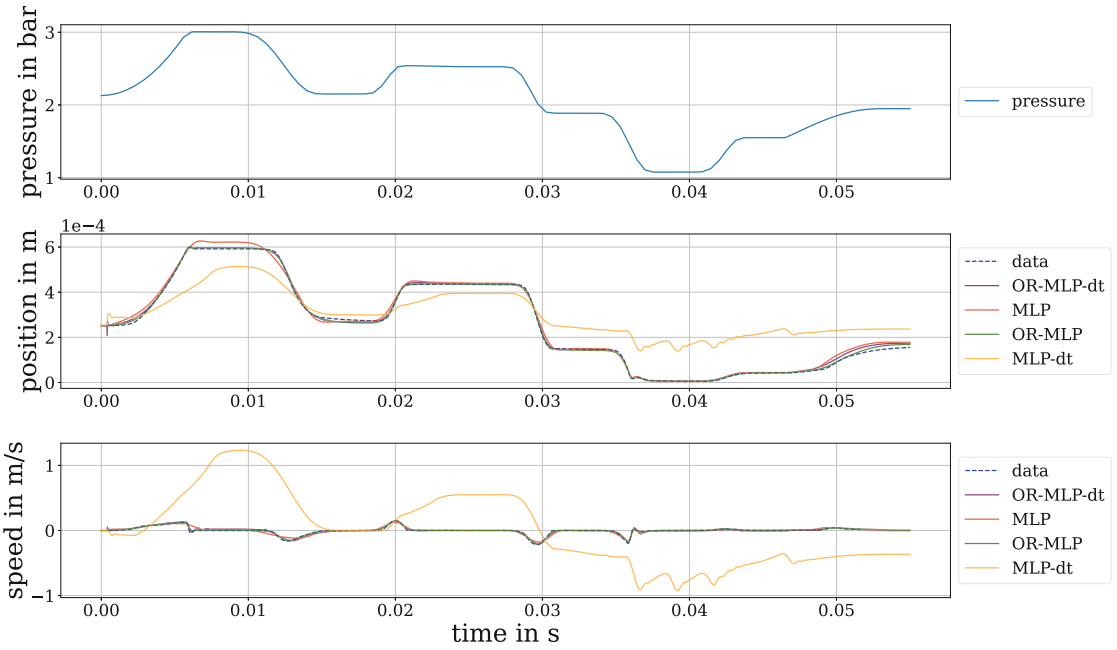

Abbildung 4.10: Simulationsergebnisse der unterschiedlichen Kategorien für das MLP, mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

Das MLP liefert in den meisten Kategorien Schätzungen mit adäquater Genauigkeit wie in Abbildung 4.10 zu erkennen ist, wobei der Ausreißer gravierender ausfällt, als beim LSTM. Wie schon beim LSTM ist dies auch beim MLP der Ableitungsschätzer ohne OR. Ein großer Punkt für das MLP ist, dass die Rechenzeit für die Simulation einer Trajektorie deutlich geringer ist.

In Abbildung 4.11 sind die verschiedenen Kategorien für die TCNs dargestellt. Dabei ist der Unterschied zwischen Netzen mit und ohne OR im Training, klar ersichtlich. Im Gegensatz dazu ist die Präzision des, OR-TCN-dt bis auf anfängliche minimale Abweichungen von den Daten, in der Prädiktion der Geschwindigkeit, mit einem MSE zwischen Prädiktion des Netzes und den Daten in einer Größenordnung von 10^{-5}, äußert genau.

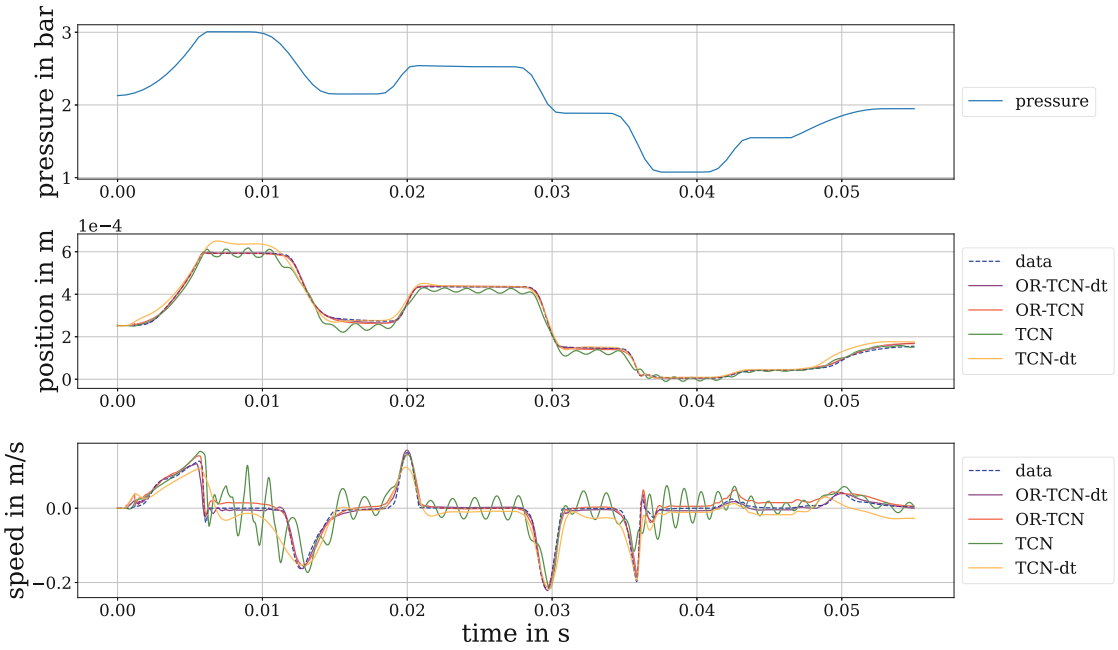

Abbildung 4.11: Simulationsergebnisse der unterschiedlichen Kategorien für das TCN, mit einer Beispieltrajektorie des Druckeingangs, die mithilfe des Zufallstrajektoriengenerators, generiert wurde. Im obersten Teilbild ist der Druckeingang und darunter die Systemzustände mit den Zielgrößen in blau strichliert dargestellt.

4.3.4 Analyse der Prädiktionsgüte über den Testdatensatz

In diesem Abschnitt wird die Verteilung der Prädiktionsfehler der MSE der Netzwerke diskutiert. Als ideal gilt eine geringe Varianz, da dann die Qualität der Schätzungen nicht von den Anfangsbedingungen oder dem externen Eingang abhängt. Insgesamt bestätigt die folgende Untersuchung der Verteilung der Prädiktionsfehler, dass die Beobachtungen, die wir bei den Beispieltrajektorien gemacht haben, auch für eine Vielzahl an Kombinationen von Trajektorien und Anfangsbedingungen ihre Gültigkeit behalten. Für die Darstellung der Verteilung der Fehler wurden die unterschiedlichen Netze auf jeweils 100 Trajektorien ausgewertet. Anschließend wurde der MSE zwischen der Prädiktion der Position s und der Geschwindigkeit v des Netzes und den Daten berechnet. Zusätzlich wurde noch Mittelwert dieser Fehler berechnet.

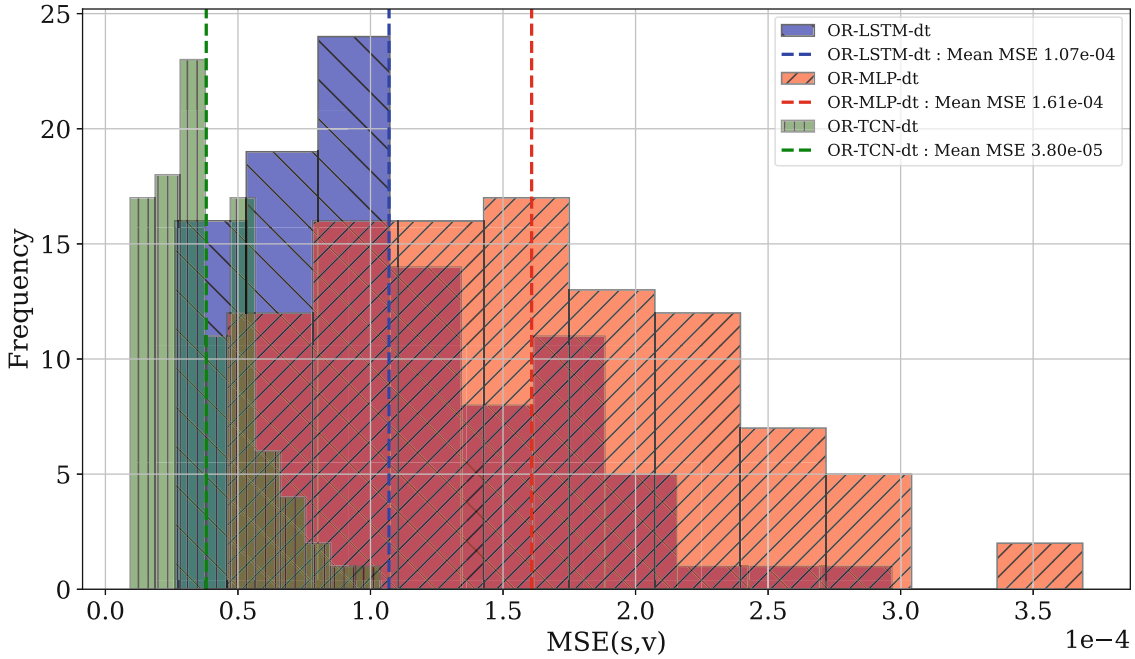

Abbildung 4.12: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der Ableitungsschätzer mit OR im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Es zeigt sich ein deutlich besseres Abschneiden des OR-TCN-dt sowohl im Hinblick auf den Mittelwert als auch bei der Varianz in Vergleich zu den anderen beiden Netzen.

Aus Abbildung 4.12 erkennen wir, dass das OR-TCN-dt den geringsten durchschnittlichen Fehler hat. Die Ergebnisse zeigen, dass das OR-LSTM-dt etwa um den Faktor zwei und das MLP Ableitungsschätzer mit Output Recurrence (OR-MLP-dt) etwa um den Faktor drei schlechter abschneiden. Neben dem mittleren Fehler ist aber auch die Varianz wichtig, um die Praktikabilität einer Methode zu bewerten. Es zeigt sich, dass das OR-TCN-dt eine deutlich geringere Varianz im Fehler aufweist als die anderen beiden Netze. Auch ist zu erkennen, dass mit dem OR-TCN-dt lediglich Ergebnisse zu erwarten sind, welche einen geringeren Fehler aufweisen als mit dem OR-LSTM-dt im Mittel erreicht werden. Die meisten Ergebnisse von OR-TCN-dt sind gar besser als die besten von OR-MLP-dt.

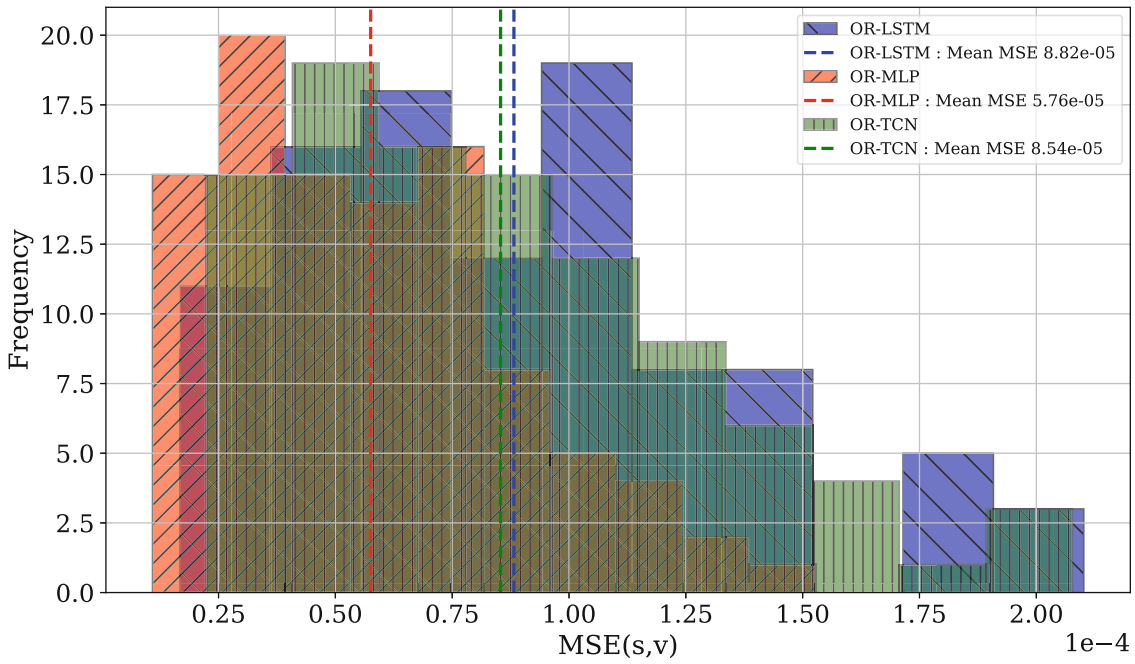

Abbildung 4.13: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der direkten Schätzer mit OR im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Alle drei Netze haben einen ähnlichen durchschnittlichen Fehler, wobei das MLP geringfügig besser abschneidet.

Bei der Fehlerverteilung der direkten Schätzer in Abbildung 4.13 zeigt sich, dass das OR-TCN eine höhere Varianz als das OR-TCN-dt aufweist. Beispielsweise sind die größten Fehler für das OR-TCN um etwa 50% größer, als die für das OR-TCN-dt. Die Mittelwerte von LSTM direkter Schätzer mit Output Recurrence (OR-LSTM) und MLP direkter Schätzer mit Output Recurrence (OR-MLP) sind sogar etwas geringer. An der x-Achse sehen wir außerdem, dass die Streubreite der Fehler von OR-LSTM und OR-MLP im Vergleich zu denen von OR-LSTM-dt und OR-MLP-dt in Abbildung 4.12, bei dieser Konfiguration geringer ausfällt. Die Abbildungen 4.12 und 4.13 bestätigen unsere Beobachtung aus dem vorherigen Abschnitt, dass die Netze mit OR im Training eine sehr hohe Präzision, bei der langfristigen Simulation von Trajektorien, ermöglichen.

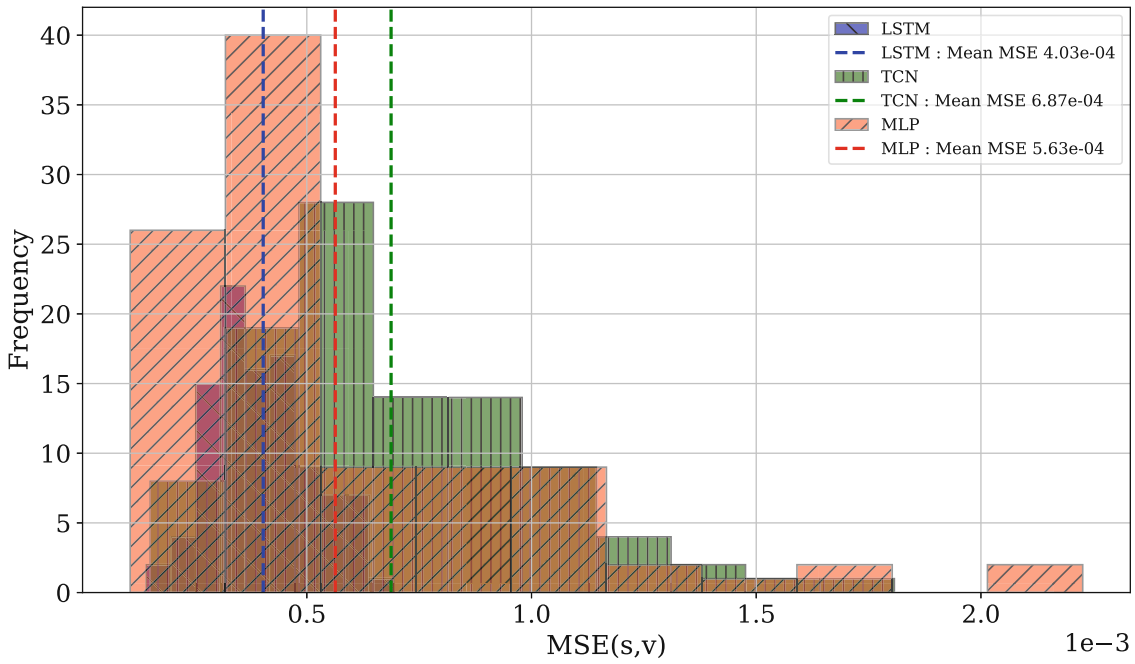

Abbildung 4.14: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der direkten Schätzer onhe OR im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Sowohl im Mittelwert, wie auch bei der Varianz sind hier die Ergebnisse des LSTM am besten.

Die direkten Schätzer ohne OR, in Abbildung 4.14, weisen, man beachte die Skalierung der x-Achse, höhere Mittelwerte auf, wodurch sich unsere Beobachtung, dass Netze ohne OR im Training, ungenauere Prädiktionen erzeugen, bestätigt. Das LSTM erzielt den besten Mittelwert und hat eine verhältnismäßig geringe Varianz, was an der engen Gruppierung der Balken um den Mittelwert zu erkennen ist. Allerdings sind die Mittelwerte der Fehler, für alle Netze, fast um einen Faktor 10 größer im Vergleich zu ihren jeweiligen Gegenstücken mit OR in Abbildung 4.13.

Der letzte Vergleich dieser Art ist in Abbildung 4.15 für die Ableitungsschätzer ohne OR dargestellt. Es ist klar zu erkennen, dass das MLP-dt im Vergleich zu den anderen Netzen keine zufriedenstellende Genauigkeit erzielen kann. Tatsächlich haben alle Netze in dieser Konfiguration den höchsten durchschnittlichen Fehler, was sich mit unseren Beobachtungen aus dem vorherigen Abschnitt deckt. Insgesamt sind die Netze der Kategorie C durchwegs die schlechtesten der vier Kategorien.

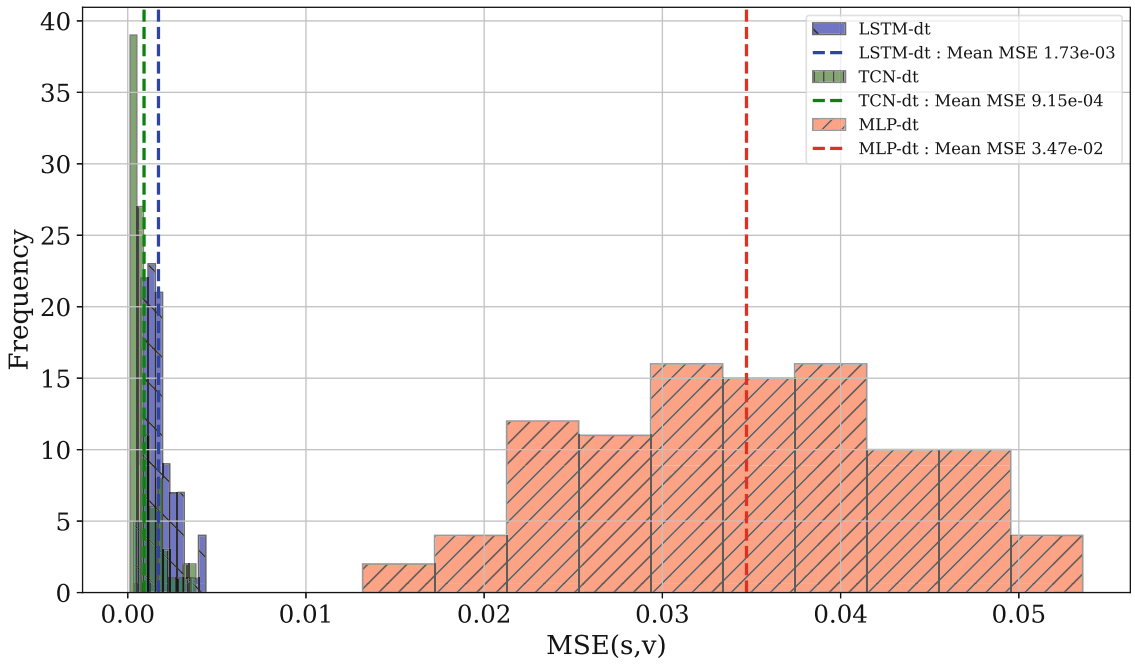

Abbildung 4.15: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der direkten Schätzer ohne OR im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Das Ergebnis zeigt ein deutlich schlechteres Abschneiden des MLP-dt im Vergleich zu den beiden anderen Netzwerkarchitekturen, die sich sowohl im Mittelwert als auch in der Verteilung ähneln.

Insgesamt zeigen die Abbildungen 4.12 und 4.13, dass alle Netzwerke der Kategorie A und B ähnlich gute Schätzungen liefern, wobei das OR-TCN-dt den kleinsten mittleren Fehler und die geringste Varianz aufweist. Weiters bestätigt sich, dass die Netzwerke mit OR im Training im Durchschnitt bessere Schätzungen liefern.

Abschließend betrachten wir die Verteilung zwischen den vier Kategorien bei fixierter Netzwerkarchitektur. Wobei alle Varianten des LSTM in Abbildung 4.16, alle des MLP in Abbildung 4.17 und alle des TCN in Abbildung 4.18 dargestellt sind. Beim Vergleich dieser Abbildungen wird der Unterschied zwischen Training mit und ohne OR nochmals deutlich sichtbar. Besonders auffällig ist, dass in den Abbildungen 4.16, 4.17 und 4.18, der Fehler der Ableitungsschätzer ohne OR im Training, deutlich höher ist als der, der anderen Kategorien, unabhängig von der Art der Netzwerkarchitektur. Außerdem sehen wir in den Abbildungen 4.16, 4.17 sowie 4.18 nochmals deutlich, dass, für alle drei Netzwerkarchitekturen, die Netze mit OR besser abschneiden. Sie haben stets die geringsten mittleren Fehler und kleinere Varianzen als die Netze ohne OR.

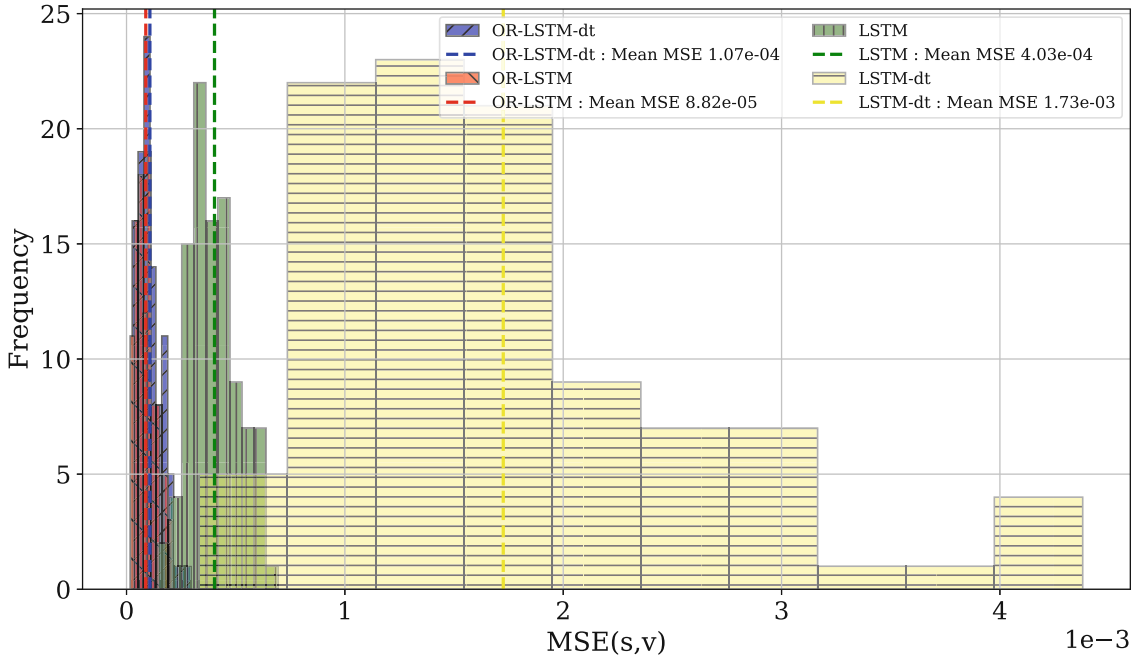

Abbildung 4.16: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der LSTMs für die Kategorien A-D im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Klar zu erkennen ist das deutlich bessere Abschneiden der Varianten mit OR, im Hinblick auf Mittelwert und Varianz.

Die Fehlerverteilung der LSTM-dt, welche in Abbildung 4.16 dargestellt ist, deckt sich mit der Beobachtung aus Abbildung 4.9. Das LSTM-dt erreicht eine grobe Näherung, der zeitliche Evolution der Systemzustände, ist jedoch im Vergleich zu den anderen Kategorien deutlich ungenauer. Der Unterschied der Mittelwerte der Kategorien A, B und D fällt dafür im Vergleich zu den anderen beiden Netzwerkarchitekturen am geringsten aus. Dennoch ist zu erkennen, dass die Varianten des LSTM mit Ableitungsschätzer bezüglich des Mittelwerts stets schlechter abschneiden als die jeweiligen direkten Schätzer mit beziehungsweise ohne OR.

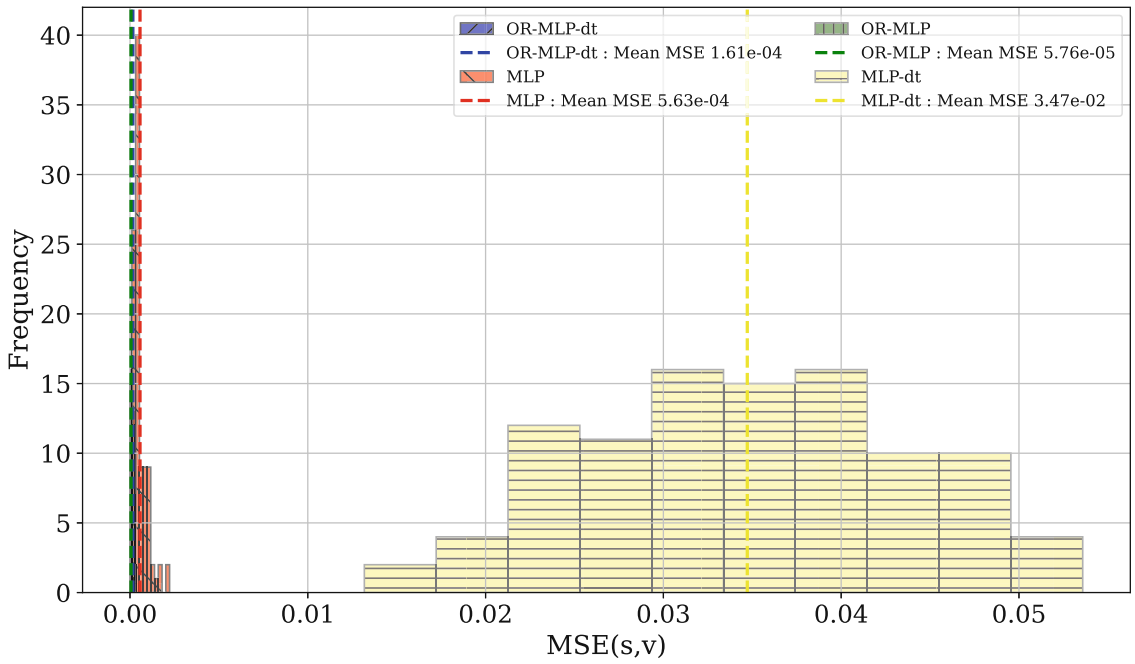

Abbildung 4.17: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der MLPs für die Kategorien A-D im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Wie bereits beim LSTM schneidet die Ableitungschätzung ohne OR mit Abstand am schlechtesten ab.

In Abbildung 4.17 ist der Fehler MLPs für alle Kategorien dargestellt. Dabei ist die Skalierung der x-Achse besonders zu beachten. Da der Fehler des MLP-dt im Durchschnitt soviel höher ausfällt, sieht es so aus, als ob die Fehlerverteilungen der übrigen Kategorien sehr ähnlich sind. Tatsächlich unterscheiden sich diese, wie wir an den Unterschieden der Mittelwerte, die in der Legende angegeben sind, erkennen können, stärker als bei den LSTM Kategorien.

Die Fehlerverteilungen der Kategorien der TCNs, unterscheiden sich, im Vergleich zu den anderen Netzwerkarchitekturen, nicht so stark, jedoch ist doch ein Faktor 10 zwischen den Netzwerken mit und ohne OR. Außerdem ist interessant, dass der mittlere Fehler des direkten Schätzer TCN, tatsächlich geringer ist als der, des TCN-dt. Dies stimmt zwar nicht mit unserer Beobachtung aus Abbildung 4.11 überein, jedoch zeigt Abbildung 4.11 nur die Prädiktion einer einzelnen Trajektorie, während in Abbildung 4.18 eine Schar von 100 Trajektorien in das Ergebnis einfließen.

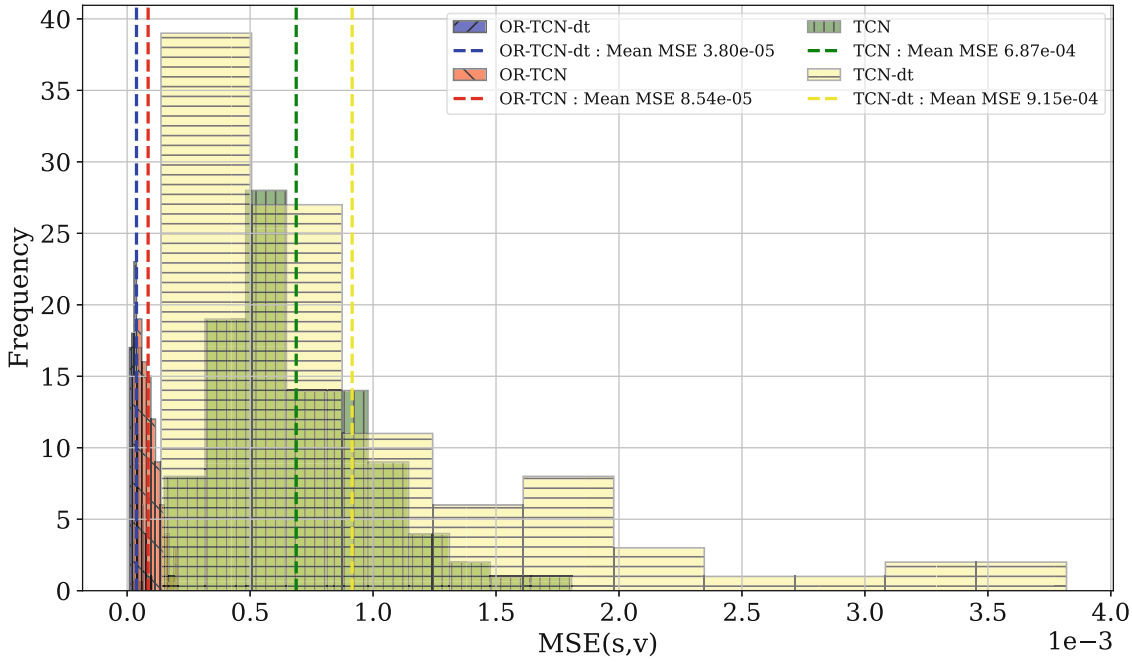

Abbildung 4.18: Verteilung und Mittelwerte des MSE der Prädiktion von Position s und Geschwindigkeit v der TCNs für die Kategorien A-D im Training. Zur besseren Übersicht, vor allem der überlappenden Flächen, sind die einzelnen Netzwerke durch unterschiedliche Schraffierungen gekennzeichnet. Die Mittelwerte sind durch horizontale strichlierte Linien gekennzeichnet, um einen leichteren Vergleich der durchschnittlichen Leistung der Netze zu ermöglichen. Auffällig ist, wie verhältnismäßig nahe, die Ergebnisse beieinander liegen.

Mit den in diesem Abschnitt besprochenen Ergebnissen ist klar, dass neuronale Netze mit OR im Training bessere Approximationen erzielen, wobei sich die Wahl der konkreten Netzwerkarchitektur nur geringfügig auf die Genauigkeit auswirkt. Der Unterschied zwischen Ableitungsschätzern und direkten Schätzern ist zwar gering, jedoch schließen letztere für alle Netzwerkarchitekturen im Durchschnitt geringfügig besser ab. Ob es trotzdem einen Grund gibt, Ableitungsschätzer zu bevorzugen, untersuchen wir im nächsten Abschnitt.

Bemerkung. Abschließend wollen wir noch anmerken, dass es auch weitere Gütekriterien gibt, die wir bei der Interpretation der Ergebnisse in diesem Abschnitt außer Acht gelassen haben. Eines davon ist die Rechenzeit. Zwar sind die Netze vergleichbar in der Anzahl ihrer Parameter, jedoch benötigt das MLP deutlich weniger Zeit, um eine Beispieltrajektorie zu berechnen. Zu große Rechenzeiten können besonders bei Echtzeitanwendungen zu Problemen führen.

4.3.5 Vorteil von Ableitungsschätzung beim Training mit spärlichen Daten

Wie in Abschnitt 3.2 beschrieben, besteht die Motivation hinter dem Lernen einer ableitungsartigen Größe des Systems darin, dass das resultierende Netzwerk widerstandsfähiger gegenüber Änderungen der Anfangsbedingungen ist. Dies ist wichtig, da die verfügbaren Daten in der Praxis eventuell nur einen Teil möglicher Anfangszustände abdecken.

Um dies zu überprüfen, trainieren wird sowohl den direkten als auch den Ableitungsschätzer des TCN mit OR Netzwerks erneut. Dabei behalten wir alle Einstellungen gleich. Lediglich die Trainingsdaten werden durch Daten ersetzt, bei denen der Druck nie einen Wert von 0.5 überschreitet. Alle Daten sind normalisiert, um zwischen 0 und 1 zu liegen, wie in Abschnitt 4.2.1 beschrieben. Die Testdaten bleiben unverändert.

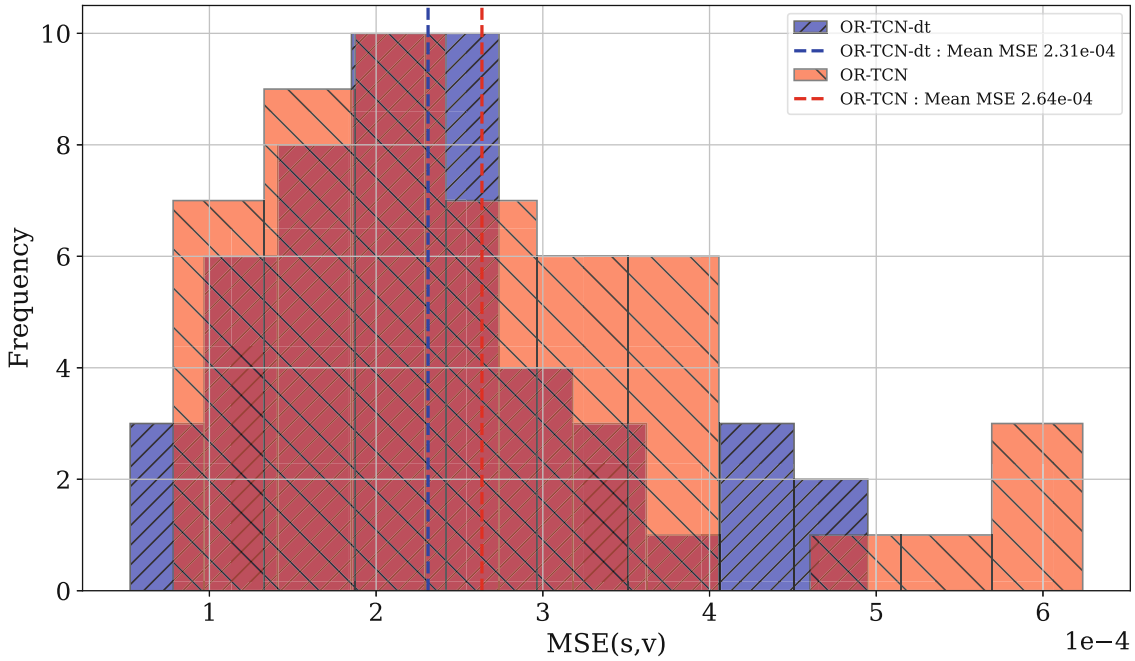

Abbildung 4.19: Verteilung und Mittelwerte der MSE der Prädiktion von Position s und Geschwindigkeit v für ein OR-TCN und ein OR-TCN-dt, die mit Daten trainiert wurden, in denen die Druckeingänge 0.5 nicht übersteigen.

Abbildung 4.19 zeigt deutlich, dass es keinen signifikanten Unterschied zwischen Ableitungsschätzung und direkter Schätzung in diesem Experiment gibt. Mit diesem Szenario, bei welchem absichtlich ein bestimmter Wertebereich des Eingangs nicht in den Trainingsdaten abgebildet ist, sehr wohl aber in den Testdaten vorkommt, dreht sich das Ergebnis zwischen Ableitungsschätzung und direkter Schätzung. Mittelwert und Varianz des OR-TCN-dt sind zwar etwas geringer, aber der Unterschied ist nicht so ausgeprägt, dass eine klare Bevorzugung eines Ableitungsschätzers gegenüber eines direkten Schätzers zu empfehlen ist. Mit der in diesem Abschnitt vorgenommenen Untersuchung kann allerdings keine definitive Antwort auf die Frage getroffen werden, ob Ableitungsschätzer resistenter als direkte Schätzer sind, wenn gewisse Teile des möglichen Wertebereichs in den Trainingsdaten nicht vorhanden sind.

Um eine eindeutige Antwort zu erhalten, wäre eine breiter angelegte Untersuchung notwendig.

4.3.6 Folgerungen aus den Ergebnissen der Experimente

Zunächst können wir aus den Experimenten schließen, dass das Verwenden von OR im Training ein stabileres Training ermöglicht und dass die daraus resultierenden Netze in allen Fällen, unabhängig von der zugrundeliegenden Netzwerkarchitektur, die besten Ergebnisse erzielen. Zwischen Ableitungsschätzern und den direkten Schätzern konnte kein signifikanter Unterschied festgestellt werden. Dennoch ist das Lernen der Ableitung, dass sich im Aufbau des Netzes in Form einer residual connection realisiert wird, aus theoretischer Sicht durchaus sinnvoll, wie wir in Abschnitt 3.2 beschrieben haben. Man sollte jedoch im Hinterkopf behalten, dass es aufgrund der zahlreichen Faktoren, die die Ergebnisse von Experimenten dieser Art beeinflussen, fast unmöglich ist, allgemein gültige Schlüsse zu ziehen.

4.4 Simulationsstudie am gesamten Pneumatikventil

In diesem Abschnitt betrachten wir das gesamte, in Abschnitt 4.1 beschriebene, Pneumatikventil. Das OR-LSTM hat sich in den Experimenten, in Abschnitt 4.3, als eine der am besten geeignetsten Architekturen herausgestellt und wird daher für die folgende Untersuchung verwendet. Das OR-TCN-dt hat zwar geringfügig bessere Ergebnisse geliefert, aber die Suche geeigneter Hyperparameter hat sich als deutlich aufwändiger herausgestellt.

Die Modellierung des gesamten Ventils unterscheidet sich von der Hauptstufe, da die Gleichungen, die das System beschreiben, wesentlich komplizierter werden und neue physikalische Effekte, besonders im Hinblick auf die piezoelektronischen Bauteile, beschrieben werden müssen.

Wir teilen unser Experiment in drei Teile. Zunächst erzeugen wir mit einem ausgereiften MATLAB\SIMULINK-Modell, dessen Funktionsweise in [42, 44, 43] beschrieben ist, synthetische Trainingsdaten. Dabei muss noch angemerkt werden, dass das MATLAB\SIMULINK-Modell, mit dem die Trainingsdaten generiert wurden, nicht auf das Ventil am Prüfstand parametriert wurde. Die adäquaten Parameter des Ventils am Prüfstand können aufgrund von Serienstreuung von denen im MATLAB\SIMULINK-Modell abweichen. Mit den Trainingsdaten wird ein neuronales Netz trainiert, um zu sehen, ob Prädiktionen mit einer

ähnlichen Präzision, wie in Abschnitt 4.3, erreichbar sind. Anschließend erzeugen wir an einem Prüfstand Messdaten, dabei werden die beiden Eingangsspannungen für die piezoelektrisch aktuierten Ventile vorgegeben und die Position des Ventilstößels, sowie der Druck in der Vorkammer der Hauptstufe, aufgezeichnet. Mit diesen Messdaten trainieren wir ein neuronales Netz und vergleichen es mit dem MATLAB\SIMULINK-Modell. Abschließend wollen wir anmerken, dass die Geschwindigkeit des Ventilstößels von den zur Verfügung stehenden Sensoren nicht direkt gemessen werden kann. Daher wird, sobald Messdaten verarbeitet werden, die Geschwindigkeit nicht mehr für die neuronalen Netze verwendet. Sämtliche Eingangsdaten, die für das Generieren von Trainingsdaten genutzt wurden, sind entweder random walks oder wurden durch den Zufallstrajektoriengenerator, der bereits in Abschnitt 4.2.1 erwähnt wurde, erzeugt. Die Systemeingänge sind zwei unabhängige Spannungen, wie in Abschnitt 4.3 dargestellt ist. Die Eingänge u_{NC}, u_{NO} sind Paare von Zeitreihen der Form,

$$(u_{NC}, u_{NO}) = \begin{cases} (rw_1, rw_2), \\ (rw_1, tg_1), \\ (tg_1, rw_2), \\ (tg_1, tg_1), \\ (tg_1, H(tg_1)), \\ (tg_1, tg_2), \end{cases}$$

wobei rw_1 und rw_2 random walks sind und tg_1 und tg_2 durch den Zufallstrajektoriengenerator erzeugt wurden. Die Funktion $H(u(t)) = 200 - u(t)$ stellt eine Spiegelung der Zeitreihe u um 100 V dar. Die verschiedenen Kombinationen aus random walks und Trajektorien des Zufallstrajektoriengenerators, sind in gleichem Ausmaß in den Trainingsdaten vorhanden. Die Trajektorien der Trainingsdaten, bestehen aus 500 Abtastpunkten für einen Zeitraum von etwa 20 ms. Im Vergleich dazu ist die Zeitreihe der Testdaten etwa 7-mal so lang.

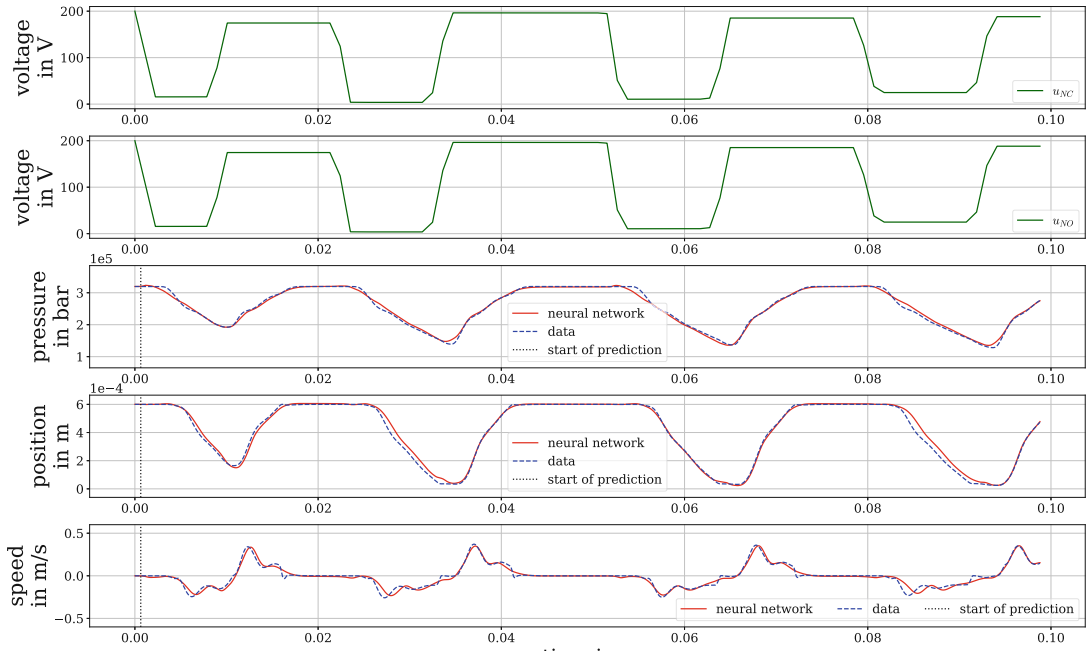

Abbildung 4.20: Simulationsergebnisse eines OR-LSTM-dt für ein mit dem Zufallstrajektoriengenerator erzeugtes Eingangssignal. Der Systemeingang, bestehend aus zwei Spannungen, ist in den oberen beiden Teilbildern dargestellt. Die übrigen Teilbilder zeigen die Systemzustände, Druck, Position und Geschwindigkeit, wobei die synthetischen Daten blau strichliert und die Approximation des Netzes rot durchgezogen, dargestellt sind.

Abbildung 4.20 zeigt, dass eine hinreichend gute Approximation, trotz der komplizierteren Dynamik des Systems, möglich ist, obwohl die Testdaten wieder deutlich längere Zeitverläufe als die Trainingsdaten darstellen. Die Spannungseingänge sind zwar gleich, jedoch können sie im Allgemeinen, wie in der Aufzählung am Beginn dieses Abschnitts beschrieben ist, unabhängig voneinander gewählt werden. Da ein solches Netzwerk, Trajektorien, die durch das Simulationsmodell erzeugt wurden, mit hoher Genauigkeit annähern kann, ist es naheliegend, dass auch Approximation von Messdaten des realen Systems möglich ist. Wie bereits am Beginn des Abschnitts erwähnt wurde, werden diese Messdaten allerdings die Geschwindigkeit nicht mehr enthalten.

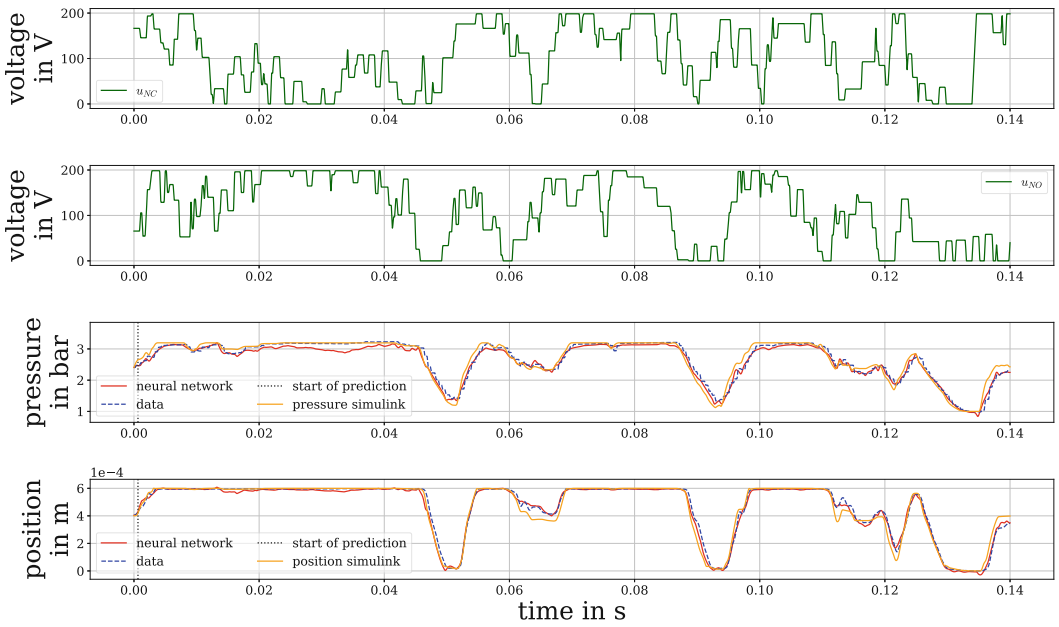

Abbildung 4.21: Vergleich der Simulationsergebnisse mit Messdaten vom MATLAB\SIMULINK-Modell und dem OR-LSTM-dt mit random walks als Spannungseingänge. Die Messdaten, in den unteren beiden Teilbildern, sind strichliert dargestellt.

In Abbildung 4.21 sieht man, dass das neuronale Netz vergleichbare und stellenweise sogar kleinere Prädiktionsfehler im Vergleich zur Messung liefert. Bei Positionsänderungen, wie etwa bei 60 ms ist das neuronale Netzer sogar deutlich näher an den Messdaten als das MATLAB\SIMULINK-Modell. Allerdings hat das OR-LSTM-dt anscheinend Probleme, einen konstanten Druck, wie sich etwa zwischen 20 ms und 40 ms einstellt, abzubilden.

Auch bei den glatteren, mit dem Zufallstrajektoriengenerator erzeugten Eingangssignalen, die in Abbildung 4.22 verwendet wurden, ist die Approximation des Netzes mit der des MATLAB\SIMULINK-Modells vergleichbar. Besonders bei positiver Druckänderung ist zu erkennen, dass das MATLAB\SIMULINK-Modell eine schnellere Steigerung des Drucks abbildet als die Messdaten. Eine mögliche Erklärung wäre, dass das Ventil, auf dem die Messdaten aufgezeichnet wurden, eine leichte Leckage aufweist, die in dem MATLAB\SIMULINK-Modells nicht abgebildet ist. Dafür wäre eine, mitunter aufwändige Neuparametrierung des MATLAB\SIMULINK-Modells notwendig. Solche Eigenschaften eines spezifischen Ventils, lassen sich eventuell mit einem neuronalen Netz, dass nur auf Basis dieses bestimmten Ventils trainiert wurde, leichter darstellen.

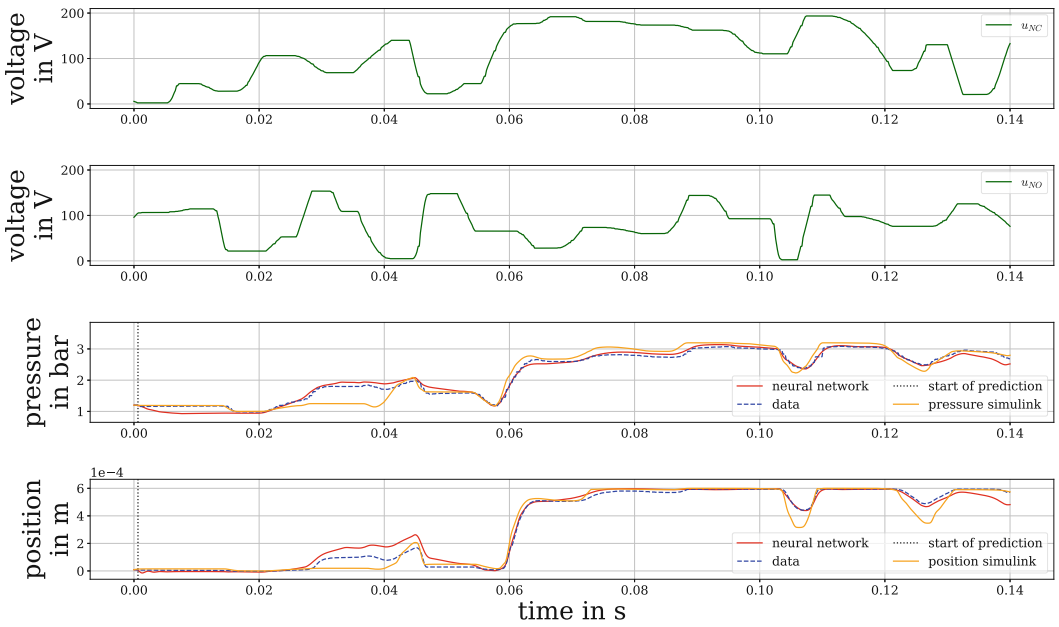

Abbildung 4.22: Vergleich der Simulationsergebnisse zwischen dem MATLAB\SIMULINK-Modell und OR-LSTM-dt mit zwei unterschiedlichen, vom Zufallstrajektoriengenerator erzeugten, Eingangsspannungen. Die Messdaten, in den unteren beiden Teilbildern, sind strichliert dargestellt, während für die Ergebnisse der Simulationen durchgezogene Linien verwendet wurden.

Wie in Abschnitt 4.3 beschrieben, betrachten wir auch für dieses Experiment die Verteilung der MSE in einem Histogramm. Wir sehen in Abbildung 4.23, dass das neuronale Netz tatsächlich im Durchschnitt einen geringeren MSE als das MATLAB\SIMULINK-Modell produziert und sogar eine geringere Varianz in der Verteilung der Ergebnisse aufweist. Anzumerken ist hierbei, dass, obwohl alle Kombinationen von Eingangssignalen, die wir oben beschrieben haben, in den Testdaten vertreten sind, aufgrund des Aufwandes der Messaufzeichnung insgesamt nur 30 Trajektorien am Prüfstand aufgezeichnet wurden. Diese verhältnismäßig kleine Datenmenge wurde dann für die Vergleiche herangezogen. In den Spannungseingängen der Testdaten, waren alle Kombinationen, die am Beginn des Abschnitts beschrieben sind, zu gleichen Teilen vorhanden.

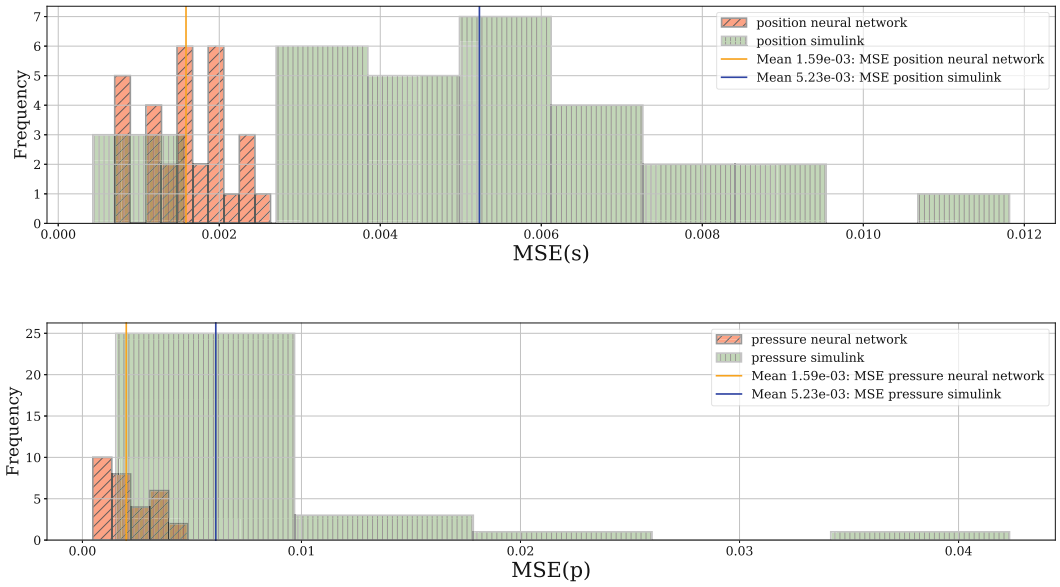

Abbildung 4.23: Verteilung und Mittelwerte des MSE von MATLAB\SIMULINK-Modell und OR-LSTM-dt. Im oberen Teilbild ist der Fehler in der Position und im unteren der Fehler im Druck dargestellt.

Bemerkung. Die Ergebnisse dieses Abschnitts sind erfreulich, da sie zeigen, dass neuronale Netze ein ernstzunehmendes Werkzeug in der Simulation dynamischer Systeme darstellen. Obwohl in datenbasierten Modellen keine physikalischen Zusammenhänge berücksichtigt sind, erreicht man dennoch eine Prädiktionsgenauigkeit, die sich mit einem ausgereiften MATLAB\SIMULINK-Modell vergleichen lässt.

5 Zusammenfassung

Diese Arbeit befasste sich mit der Simulation dynamischer Systeme mit Hilfe von neuronalen Netzen. Dafür wurden neuronale Netze als Black-Box-Modelle genutzt, um Systemzustände, in Form von diskreten Zeitreihen, zu approximieren. Um einen theoretischen Rahmen zu schaffen, wurden zunächst die relevanten Aspekte der Theorie der dynamischen Systeme und der neuronalen Netze aufbereitet. Anschließend wurden Implementierungen der unterschiedlichen Netzwerkarchitekturen vorgenommen und Simulationsstudien durchgeführt.

Ein Ergebnis der durchgeführten Experimente ist, dass die repräsentativ ausgewählten Netzwerkarchitekturen alle gute Simulationsergebnisse erzielten. Allerdings konnte kein klarer Favorit unter den drei betrachteten Netzwerkarchitekturen ausgemacht werden. Weiters wurde ein Vergleich zwischen Ableitungsschätzern und direkten Schätzern durchgeführt. Dabei wurde analysiert, ob Ableitungsschätzer besser für das Training mit Daten, die nicht alle möglichen Systemzustände enthalten, geeignet sind. Die Ableitungsschätzer haben zwar geringfügig besser abgeschnitten, aber eine definitive Aussage ließ sich dennoch nicht treffen. Dafür wäre eine umfangreichere Untersuchung bei der eine größere Anzahl dynamischer Systeme mit unterschiedlichen Restriktionen der Trainingsdaten, notwendig.

Die zentrale Erkenntnis dieser Arbeit ist, dass die Verwendung von Output Recurrence, im Training die beste Strategie für das Training von neuronalen Netzen zur Modellierung von dynamischen Systemen dargestellt hat. Sowohl die mit OR trainierten Ableitungsschätzer als auch die direkten Schätzer waren unabhängig von der Netzwerkarchitektur besser als die Netze ohne OR im Training. Diese Resultate sind ein wichtiger erster Schritt, jedoch sind weiterführende Untersuchungen notwendig, um die Ergebnisse zu validieren und zu prüfen, ob diese sich bestätigen und erweitern lassen.

Insgesamt haben sich neuronale Netze, wie in vielen wissenschaftlichen Disziplinen, als nützliches Werkzeug herausgestellt und die gewonnenen Erkenntnisse bieten eine solide Grundlage für zukünftige Untersuchungen in diesem Bereich.

6 Ausblick

Um diese Arbeit abzuschließen, werden einige weiterführende Ideen und Problemstellungen angegeben, die im Zuge des Erstellens dieser Arbeit aufgetreten sind. Um der Antwort auf die Frage nach einer zu bevorzugenden Netzwerkarchitektur zur Simulation näherzukommen, sind weitere Arbeitsschritte notwendig. Eine umfassendere Studie, bei der nicht nur modernere Netzwerkarchitekturen, wie zum Beispiel die Transformer Architektur, sondern auch diverse dynamische Systeme systematisch im Hinblick auf ein allgemeines Vergleichsmaß analysiert werden, wäre dafür zielführend. Dabei sollten auch weitere wichtige Größen, wie die Menge an notwendigen Trainingsdaten sowie die Komplexität der dynamischen Systeme beziehungsweise die Anzahl und Komplexität der notwendigen Berechnungen, welche sich in der Rechenzeit widerspiegeln, beachtet werden.

Weiters könnte man mit Hilfe von physically informed neural networks PINNs, die in Abschnitt 2.2.5 vorgestellt wurden, auf einen Gray-Box-Modellierungsansatz übergehen. Dabei könnten bekannte physikalische Eigenschaften des Systems, welche man zu modellieren gedenkt, in den Trainingsprozess des Netzes einfließen. Die Dynamik der betrachteten Systeme war in dieser Arbeit zwar bekannt, aber war nie Teil des eigentlichen Modellierungsprozesses. In dieser Arbeit wurde außerdem bewusst mit Zeitreihendaten mit konstanter Abtastrate gearbeitet. Bei einigen realen Systemen entspricht dies allerdings nicht der gängigen Praxis. Die in Abschnitt 2.2.4 beschriebene NCDE Architektur, könnte dieses Problem lösen, da sie, um ein stetiges Signal zu erhalten, eine Interpolation der Netzeingänge durchführt. Ein logischer nächster Schritt in diese Richtung wäre daher, die in dieser Arbeit durchgeführten Versuche auf PINNs und auf Netze der NCDE Architektur auszuweiten.

Schließlich wäre es interessant, eine mögliche Integration von neuronalen Netzen in Regelungssysteme zu untersuchen. Besonders eine Art adaptiver Regelung, bei der Änderungen des Systems, wie beispielsweise Alterungseffekte berücksichtigt werden, lässt sich in natürlicher Weise auf neuronale Netze übertragen. Dazu wäre es von besonderem Interesse, die Netze im laufenden Betrieb effizient, durch nachträgliches Training anpassen, zu können.

Abbildungsverzeichnis

Literaturverzeichnis

[1] A. Saviolo, G. Li, and G. Loianno, "Physics-Inspired Temporal Learning of Quadrotor Dynamics for Accurate Model Predictive Trajectory Tracking," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 256–10 263, 2022, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1109/LRA.2022.3192609

[2] M. Sangiorgio and F. Dercole, "Robustness of LSTM neural networks for multi-step forecasting of chaotic time series," *Chaos, Solitons Fractals*, vol. 139, p. 110045, 2020, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.chaos.2020.110045

[3] Y. Xue, J. Jiang, and L. Hong, "A LSTM based prediction model for nonlinear dynamical systems with chaotic itinerancy," *International Journal of Dynamics and Control*, vol. 8, pp. 1117–1128, 2020, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1007/s40435-020-00673-4

[4] J. M. Maroli, "Generating discrete dynamical system equations from input–output data using neural network identification models," *Reliability Engineering & System Safety*, vol. 235, pp. 109 198: 1–11, Jul. 2023, zuletzt aufgerufen: 03.06.2024. [Online]. Available: https://doi.org/10.1016/j.ress.2023.109198

[5] B. Kapusuzoglu and S. Mahadevan, "Information fusion and machine learning for sensitivity analysis using physics knowledge and experimental data," *Reliability Engineering & System Safety*, vol. 214, pp. 107 712: 1–18, Oct. 2021, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.ress.2021.107712

[6] Z. Zhao, J. Zhang, Z. Liu, C. Mu, and K.-S. Hong, "Adaptive Neural Network Control of an Uncertain 2-DOF Helicopter With Unknown Backlash-Like Hysteresis and Output Constraints," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 12, pp. 10 018–10 027, 2023, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1109/TNNLS.2022.3163572

[7] D. C. Gordon, A. Winkler, J. Bedei, P. Schaber, S. Pischinger, J. Andert, and C. R. Koch, "Introducing a Deep Neural Network-Based Model Predictive Control Framework for Rapid Controller Implementation," in *Proceedings of the 2024 American Control Conference (ACC)*, 2024, pp. 5232–5237, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.23919/ACC60939.2024.10644830

[8] R. Chen, X. Jin, S. Laima, Y. Huang, and H. Li, "Intelligent modeling of nonlinear dynamical systems by machine learning," *International Journal of Non-Linear Mechanics*, vol. 142, p. 103984, 2022, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.ijnonlinmec.2022.103984

[9] K. Yeo and I. Melnyk, "Deep learning algorithm for data-driven simulation of noisy dynamical system," *Journal of Computational Physics*, vol. 376, pp. 1212–1231, 2019, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.jcp.2018.10.024

[10] M. Forgione and D. Piga, "dynoNet: A neural network architecture for learning dynamical systems," *International Journal of Adaptive Control and Signal Processing*, vol. 35, no. 4, pp. 612–626, 2021, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1002/acs.3216

[11] K. S. Narendra and K. Parthasarathy, "Neural networks and dynamical systems," *International Journal of Approximate Reasoning*, vol. 6, no. 2, pp. 109–131, 1992, zuletzt besucht : 03.05.2024. [Online]. Available: https://doi.org/10.1016/0888-613X(92)90014-Q

[12] S. N. Kumpati and P. Kannan, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on neural networks*, vol. 1, no. 1, pp. 4–27, 1990, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1109/72.80202

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, zuletzt aufgerufen: 12.10.2024. [Online]. Available: http://www.deeplearningbook.org

[14] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2007, vol. 5.

[15] C. C. Aggarwal, *Neural Networks and Deep Learning*. Springer, 2018.

[16] A. Sivaram, L. Das, and V. Venkatasubramanian, "Hidden representations in deep neural networks: Part 1. Classification problems," *Computers Chemical Engineering*, vol. 134, pp. 106 669: 1–17, 2020, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.compchemeng.2019.106669

[17] C. Fefferman, S. Mitter, and H. Narayanan, "Testing the Manifold Hypothesis," *arXiv:1310.0425v2 [math.ST]*, 2013, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.1310.0425

[18] A. Gharehbaghi, *Deep Learning in Time Series Analysis*. CRC Press, 2023, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1201/9780429321252

[19] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[20] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean, "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 10 2017, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1162/tacl_a_00065

[21] W. Xiong, L. Wu, F. Alleva, J. Droppo, X. Huang, and A. Stolcke, "The Microsoft 2017 Conversational Speech Recognition System," in *Proceedings of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 5934–5938, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1109/ICASSP.2018.8461870

[22] J. Schmidhuber, "Deep Learning: Our Miraculous Year 1990-1991," *arXiv:2005.05744 [cs.NE]*, 2020, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.2005.05744

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs.CV]*, pp. 770–778, 2016, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.1512.03385

[24] S. Bai, J. Z. Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling," *arXiv:1803.01271v2 [cs.LG]*, 2018, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.1803.01271

[25] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural Ordinary Differential Equations," *arXiv:1806.07366v5 [cs.LG]*, 2019, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.1806.07366

[26] P. Kidger, J. Morrill, J. Foster, and T. Lyons, "Neural Controlled Differential Equations for Irregular Time Series," *arXiv:2005.08926v2 [cs.LG]*, 2020, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.2005.08926

[27] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.jcp.2018.10.045

[28] A. M. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comptes rendus de l'Académie des sciences*, pp. 536–538, 1847, zuletzt aufgerufen: 12.10.2024, Übersetzung von Richard J. Pulskamp. [Online]. Available: https://cs.uwaterloo.ca/~y328yu/classics/cauchy-en.pdf

[29] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1007/BF02551274

[30] M. E. Valle, W. L. Vital, and G. Vieira, "Universal Approximation Theorem for Vector-and Hypercomplex-Valued Neural Networks," *arXiv:2401.02277v2 [cs.LG]*, 2024, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.02277

[31] W. Rudin, *Functional Analysis*, ser. International series in pure and applied mathematics. McGraw-Hill, 1991.

[32] W. Rudin, *Real and Complex Analysis*. McGraw-Hill, 1987.

[33] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980v9 [cs.LG]*, 2017, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.48550/arXiv.1412.6980

[34] H. Broer, F. Takens, and B. Hasselblatt, *Handbook of Dynamical Systems*. Elsevier Science, 2010, vol. 3.

[35] H. Khalil, *Nonlinear Systems*. Prentice Hall, 2002.

[36] D. Hinrichsen and A. J. Pritchard, *Mathematical Systems Theory I Modelling, State Space Analysis, Stability and Robustness*, 1st ed., ser. Texts in Applied Mathematics. Springer Berlin, Heidelberg, 2005, vol. 48, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1007/b137541

[37] C. Legaard, T. Schranz, G. Schweiger, J. Drgoňa, B. Falay, C. Gomes, A. Iosifidis, M. Abkar, and P. Larsen, "Constructing Neural Network Based Models for Simulating Dynamical Systems," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–34, Feb. 2023, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1145/3567591

[38] A. Levin and K. Narendra, "Control of nonlinear dynamical systems using neural networks. II. Observability, Identification, and control," *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 30–42, 1996, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1109/72.478390

[39] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, and A. Juditsky, "Nonlinear black-box modeling in system identification: a unified overview," *Automatica*, vol. 31, no. 12, pp. 1691–1724, 1995, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/0005-1098(95)00120-8

[40] T. Söderström and P. Stoica, *System Identification*, ser. Prentice-Hall Software Series. Prentice Hall, 1989, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/0005-1098(92)90167-E

[41] Produktbeschreibung Festo Ventil VEVM. Zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://www.festo.com/at/de/p/ventil-id_VEVM/?q=%7E%3AsortByCoreRangeAndNewProduct

[42] T. Glück, D. Büchl, C. Krämer, A. Pfeffer, A. Risle, L. Hägele, and A. Kugi, "Modeling and control of a novel pneumatic two-stage piezoelectric-actuated valve," *Mechatronics*, vol. 75, pp. 102 529 : 1–11, 2021, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.1016/j.mechatronics.2021.102529

[43] C. Krämer, "Regelung von pneumatischen Lasten mithilfe eines piezoelektrischen Stetigventils," Diplomarbeit, Technische Universität Wien, Wien, Jun 2016, zuletzt aufgerufen: 12.10.2024. [Online]. Available: http://hdl.handle.net/20.500.12708/158627

[44] D. Büchl, "Modellierung und Regelung eines pneumatischen, piezoelektrischen Stetigventils," Diplomarbeit, Technische Universität Wien, Wien, Jan 2015, zuletzt aufgerufen: 12.10.2024. [Online]. Available: http://hdl.handle.net/20.500.12708/158627

[45] C. Geischläger, "Analyse nichtlinearer Regelkreise mittels computerunterstützter Methoden," Diplomarbeit, Technische Universität Wien, Wien, Februar 2023, zuletzt aufgerufen: 12.10.2024. [Online]. Available: https://doi.org/10.34726/hss.2023.100620

[46] P. Strasser, "Code repository - Masterarbeit Paul Strasser." [Online]. Available: https://github.com/Gandalf3141/ventil_lstm