Master's Thesis

Constraint Optimization for Reservoir Learning of Multivariate Time Series

Yannic Lieder

April 6, 2021

Institute of Robotics and Process Control

Prof. Dr. Jochen J. Steil

Supervisors:
Prof. Dr. Jochen J. Steil
Heiko Donat, M.Sc.

Statement of Originality

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, April 6, 2021

Abstract

Embedding prior knowledge into recurrent neural networks can be useful, especially if only a small amount of training data is available or the training data is very noisy. Reservoir computing is a recurrent neural network framework that has been proven to be highly efficient in learning temporal dynamic systems. Due to its specific structure consisting of a high-dimensional, non-linear reservoir combined with a simple read-out layer, the reservoir computing framework is predestined to be extended by an approach for the explicit learning of prior knowledge. In this thesis, we introduce a framework which enables the formalization of prior knowledge as mathematical constraints. The framework covers a wide range of different constraints, including lower and upper bounds to the network output and constraints concerning the periodicity, monotonicity and curvature behavior of the network output with respect to the time. The framework is applied to the Echo State Network, one of the major reservoir computing approaches. An algorithm for training Echo State Networks under constraints is presented. Furthermore, it is shown that the algorithm can be extended to Echo State Networks with output feedback. With the forecasting of satellite images of woodland areas, the framework is applied to a practical learning task.

Contents

1 Introduction **1**
- 1.1 Related Work . . . 2
- 1.2 The Constrained Extreme Learning Machine . . . 3
- 1.3 Evaluation Metrics . . . 5
- 1.4 Outline . . . 6

2 The Constrained Echo State Network **7**
- 2.1 The Basic Echo State Network . . . 7
- 2.2 Constraints for the ESN . . . 8
- 2.3 Embedding Constraints into the Network . . . 15

3 Practical Realization **21**
- 3.1 Algorithmic Implementation . . . 21
- 3.2 Illustrative Example: Frequency-varying Sine Wave . . . 24

4 Application: Satellite Image Forecasting **33**
- 4.1 Planet Labs Inc. . . . 33
- 4.2 Data and Data Preparation . . . 33
- 4.3 Experimental Setup . . . 36
- 4.4 Results and Evaluation . . . 38

5 Extension: Output Feedback **43**

6 Conclusion and Discussion **49**

Bibliography **53**

Appendices

A1 ESN Derivatives **59**

A2 Linear Least Squares as Quadratic Program **61**

A3 Practical Implementation **63**

A4 Gridsearch parameters **67**

A5 Master's Thesis Scope **71**

List of Figures

1.1 ELM Structure . . . 3
1.2 Example of the CELM . . . 4

2.1 ESN Structure . . . 8
2.2 Failing ESN constraints . . . 12
2.3 Cause of failing ESN constraints . . . 13
2.4 Reduction of ESN training problem to QP . . . 16
2.5 Unfolding in time . . . 20

3.1 Sine wave with variable frequency . . . 25
3.2 Rotation on unit circle . . . 25
3.3 Illustrative example 1: Result plots . . . 28
3.4 Illustrative example 2: Training data plot . . . 29
3.5 Illustrative example 2: Result plots . . . 32

4.1 Planet Basemap tile example . . . 34
4.2 Planet Basemap of the Harz highland area . . . 35
4.3 Satellite image pixel mask . . . 35
4.4 Noisy Planet Basemap tiles . . . 36
4.5 Colorspace of woodland area satellite images . . . 37
4.6 Satellite data application: Result plots over training dataset size . . . 39
4.7 Satellite data application: Result plots over time . . . 41
4.8 Forecasted image examples . . . 42
4.9 Input data image examples . . . 42

5.1 Structure of ESN with output feedback . . . 43
5.2 Unfolding in time for ESN with output feedback . . . 44
5.3 Illustrative example of CESN with output feedback . . . 46
5.4 ESN with output feedback: Result plot . . . 47
5.5 ESN with output feedback: Iterative progress . . . 48

List of Tables

2.1 QP constraints in standard form 20

3.1 Illustrative example 1: Absolute and relative errors 27
3.2 Illustrative example 2: Absolute and relative errors 31

4.1 Satellite data application: Absolute errors and error ratios 39

A4.1 Gridsearch: Illustrative example 1 (ESN) 67
A4.2 Gridsearch: Illustrative example 1 (CESN) 67
A4.3 Gridsearch: Illustrative example 2 (ESN) 67
A4.4 Gridsearch: Illustrative example 2 (ESN (mean)) 68
A4.5 Gridsearch: Illustrative example 2 (ESN (mean extended)) 68
A4.6 Gridsearch: Illustrative example 2 (CESN (training data only)) 68
A4.7 Gridsearch: Illustrative example 2 (CESN) 68
A4.8 Gridsearch: Satellite image forecasting application 69

Acronyms

AOI Area of Interest. 33, 36

BPDC Backpropagation-Decorrelation. 2

CELM Constrained Extreme Learning Machine. 3, 4, 8, 9, 50

CESN Constrained Echo State Network. 6, 15, 21, 22, 24, 26–28, 30, 31, 33, 36–42, 48, 50, 51, 63, 65, 67, 68

CLLS Constrained Linear Least Squares. 17–20, 61

ELM Extreme Learning Machine. 3, 4, 7, 8, 10

ESN Echo State Network. 1, 2, 7–17, 20–22, 24, 26–28, 30, 31, 37–46, 49, 51, 64, 67, 68

LLS Linear Least Squares. 15–18

LSM Liquid State Machine. 2

QP Quadratic Program. 19, 20, 23, 27, 45, 61, 63–65

RMSE Root Mean Square Error. 16, 27, 38, 40

RNN Recurrent Neural Network. 1, 2, 10, 21, 49, 51

List of Symbols

α	ESN leaking rate
β	Regularization parameter
$\mathcal{C}_t$	Set of constraints applied at time step t
D, $D_{\mathbf{u}}$	Component-wise differential operator with respect to $\mathbf{u}$
$D_{\mathbf{x}}$	Component-wise differential operator with respect to $\mathbf{x}$
ϵ	Error
L	ESN/ELM constraint
N_{co}	Length of input signal for applying constraints
N_u	Number of ELM/ESN input units
N_{tr}	Length of training input and output signal
N_x	Number of ESN reservoir nodes/ELM hidden nodes
$N_{\tilde{x}}$	Size of extended reservoir output
N_y	Number of ELM/ESN output units
ρ	Spectral radius of weight matrix $\mathbf{W}$
σ	Neuron activation function
t	Discrete time variable (time step)
t_0	Transient time
$\mathbf{u}$, u	Input vector of a neural network (multi- or one-dimensional)
$\mathbf{U}$	Input vector of a neural network collected over time series
$\mathbf{u}^{\text{co}}$	Input signal for applying constraints
$\mathbf{u}^{\text{co}}$	Input signal for applying constraints collected over time series
$\mathbf{W}$	ESN reservoir weight matrix
$\mathbf{W}^{\text{in}}$	ELM/ESN input weight matrix
$\mathbf{W}^{\text{fb}}$	ESN output feedback weight matrix
$\mathbf{W}^{\text{out}}$	ELM/ESN output weight matrix
$\hat{\mathbf{W}}^{\text{out}}$	Reshaped ESN output weight vector
$\mathbf{x}$	ESN reservoir state/reservoir output
$\tilde{\mathbf{x}}$	Extended reservoir output
$\tilde{\mathbf{X}}$	Extended reservoir output collected over time series
$\hat{\tilde{\mathbf{X}}}$	Block-diagonal matrix of $\tilde{\mathbf{X}}$
$\mathbf{x}^{\text{cont}}$	Continuous ESN reservoir equation
$\mathbf{x}^{\text{prev}}$	Variable for ESN reservoir state in continuous ESN equation
$\mathbf{y}$, y	Output vector of a neural network (multi- or one-dimensional)
$\mathbf{y}^{\text{cont}}$	Continuous ESN equation
$\mathbf{y}^{\text{target}}$, y^{target}	ESN training target output (multi- or one-dimensional)
$\mathbf{Y}^{\text{co}}$	ESN output of constraint input signal collected over time series
$\mathbf{Y}^{\text{target}}$	ESN training target output collected over time series
$\hat{\mathbf{Y}}^{\text{target}}$	Reshaped ESN training target output collected time series

1 Introduction

Usually, artificial neural networks are trained by processing a set of training samples, each of which consists of an input and an expected result. By internalizing the relation between the input and the output data, the network learns to produce results close to the expected ones. The concept's key idea is to learn a model that generalizes to new data and may output good results also for new data not available during the training process. This often works quite well if the training dataset is large enough and represents the data domain well. When dealing with model learning for time series like speech recognition, text analysis, or stock market prediction, recurrent neural networks (RNNs) are state-of-the-art. In contrast to feedforward neural networks, RNNs exhibit a kind of memory and thus can learn temporal dynamic behavior. Due to its significantly higher complexity, training an RNN is much more expensive and requires an even larger set of training data than feedforward neural networks.

But how can we deal with tasks for which only very noisy or a very small amount of training data is available? Often, the raw training data is not the only information available about the task. It is common to use additional knowledge about the expected inputs and expected outputs to improve the results by preprocessing the training data or postprocessing the network predictions. One such example is outlier detection: Knowledge about the smoothness of the target function is used to remove data points that do not fit into the expected schema. In this thesis, we consider an alternative approach of using prior knowledge in neural network learning. By formalizing this prior knowledge in the form of mathematical constraints, these constraints can be embedded straight into the neural network during the training process. As a result, no further pre- or postprocessing of the data is necessary, since the neural network is not only able to generalize the relation between the input and the output training data to new input data, but also to generalize the constraints. Such constraints can express a wide range of different properties about the learning function, e.g., about lower and upper bounds, the curvature, the smoothness, the periodicity or the monotonicity of the function. Embedding suitable constraints into a neural network can help to fill the gap of missing information in the training data.

Reservoir Computing is an umbrella term for RNNs with a specific network architecture. The input is mapped to a high-dimensional, recurrent system, the so-called reservoir. This system is fixed and induces a high non-linearity caused by its high dimensionality and a suitable activation function. A read-out layer harvests the reservoir state and is trained to compute the desired target outputs out of it. One reservoir computing approach is the Echo State Network (ESN) [17], which, due to its architecture, is predestined to be extended by the constraint learning approach.

1.1 Related Work

In the early 2000s, Jaeger [16, 17], Maass et al. [31] and Natschläger et al. [32] proposed two RNN structures, both combining a high-dimensional reservoir with the training of a single read-out layer, simultaneously. The introduced concepts of the Echo State Network (ESN) (Jaeger) and the Liquid State Machine (LSM) (Maas et al., Natschläger et al.) were the first major reservoir computing approaches. While both networks are basically similar in their ideas, they differ in their neuron concepts. The ESN uses the more conventional neuron paradigm known from multiplayer perceptron networks. On the other hand, the LSM is based on a spiking neuron model. A third reservoir computing concept, Backpropagation-Decorrelation (BPDC), was suggested in 2004 by Steil [39]. By combining a backpropagation RNN learning algorithm [1] with the concept of reservoir computing, BPDC turned out to be an efficient online learning rule for RNNs. Subsequent papers analyzes the stabilitiy of BPDC [41] and suggests further improvements concerning memory and regularization [40]. These three approaches were first unified in the reservoir computing framework in 2007 [44]. Later, further reservoir computing concepts were suggested, e.g., Evolino, which is a Long Short-Term Memory RNN with a linear read-out [37]. Reviews to the reservoir computing framework in general and overviews to the different approaches are given in [26, 27, 38].

The ESN, introduced in 2001, was a game-changer in the field of RNNs. RNNs always lacked efficient training algorithms, which changed with the introdcution of the ESN. Since then, much research has been done on ESNs. Practical guides to applying ESNs are given by Lukoševičius [25] and Jaeger [18]. Several papers analyze the architecture of the ESN or the ESN's reservoir and suggest improvements and extensions in the reservoir structure. In [36], the minimum complexity and memory capacity of ESNs is analyzed. Gallichio et al. [8, 11] suggested the so-called deepESN, which applies the ESN approach to the deep learning framework. Other variations includes, for example, ESNs with leaky integrator neurons [19, 29], ESNs with multiple subreservoirs [12, 35] and convolutional ESNs [30]. Several papers tackled the issues of the training of the read-out weights [45, 46] and a good initialization of the ESN reservoir [5, 6, 24, 42].

The embedding of prior knowledge into neural networks is issued in several papers, mainly concerning feedforward neural networks. The reliable embedding of specific kinds of prior knowledge is issued in serveral papers, for example monotonicity constraints [7, 21] or gain constraints [13]. A wider range of constraints without reliability verification are supported by regression methods suggested by Sun et al. [43] and Lauer and Block [22]. Finally, the Constrained Extreme Learning Machine introduced by Neumann et al. [33, 34] allows the embedding and reliability verification of a wide range of formal constraints into the feedforward neural networks. Concerning RNNs, Chen et al. [4] proposed an approach to embed first order logic rules into RNNs. Another paper presents an application of ESNs, where prior knowledge is used to determine the reservoir topology [49].

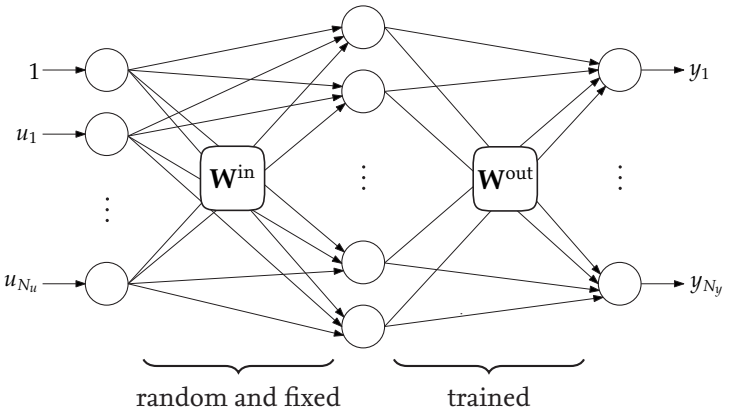

Figure 1.1: The ELM is a feedforward neural network with randomly assigned input weights and a trainable linear read-out layer.

1.2 The Constrained Extreme Learning Machine

This work builds on the Constrained Extreme Learning Machine (CELM) approach introduced by Neumann, Rolf and Steil in 2013 [34] and thus a short introduction to the CELM is given. An Extreme Learning Machine (ELM) [14, 15] is a feedforward neural network with one input layer of size N_u, one hidden layer of size N_x and one output layer of size N_y (see Figure 1.1). Mathematically, the network can be expressed as

$$\mathbf{y}(\mathbf{u}) = \mathbf{W}^{\text{out}}\,\sigma\left(\mathbf{W}^{\text{in}}\begin{bmatrix}1\\ \mathbf{u}\end{bmatrix}\right), \tag{1.1}$$

where

$$\begin{bmatrix}1\\ \mathbf{u}\end{bmatrix} = \begin{bmatrix}1\\ u_1\\ \vdots\\ u_{N_u}\end{bmatrix} \tag{1.2}$$

is a vertical vector concatenation. $\mathbf{W}^{\text{in}} \in \mathbb{R}^{N_x \times N_u+1}$ and $\mathbf{W}^{\text{out}} \in \mathbb{R}^{N_y \times N_x}$ are the input and output weights, respectively, and $\sigma : \mathbb{R} \to \mathbb{R}$ an element-wise applied neuron activation function, typically a sigmoid function like the logistic function. $\mathbf{u} \in \mathbb{R}^{N_u}$ is the network input and $\mathbf{y} \in \mathbb{R}^{N_y}$ the corresponding network output. One can see that the outputs result as linear combinations of the hidden layer neuron activations with read-out weights $\mathbf{W}^{\text{out}}$ as coefficients. While the input-to-hidden layer weights $\mathbf{W}^{\text{in}}$ are assigned randomly and are fixed, only the output weights $\mathbf{W}^{\text{out}}$ are optimized during the training process. Hence, optimizing the network in the sense of minimizing a training error ϵ is equivalent to solving a linear regression problem and can be accomplished using a suitable algorithm like ridge regression. A further detailed explanation is described in Section 2.3. This approach is much faster than the computationally expensive backpropagation algorithm, usually used to train feedforward neural networks.

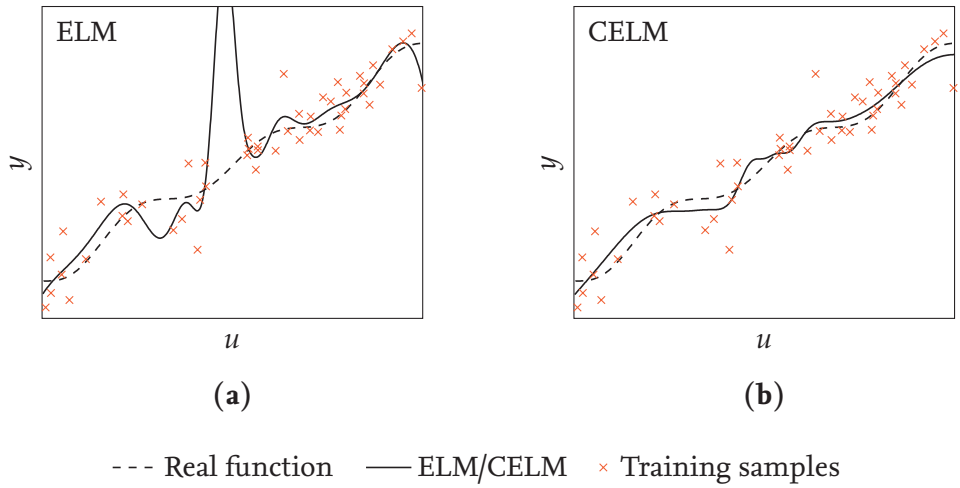

Figure 1.2: **(a)** The ELM without constraints tends to strong overfitting due to the few and noisy training data. **(b)** The CELM with monotonicity constraints is much closer to the real function and leads to a good generalization capability. Figures adapted from [33].

Neumann et al. [34] introduce a framework that exploits the linearity of the ELM's read-out weights and allows to force constraints to the network output. These constraints are defined by linear combinations of partial derivatives of arbitrary order with respect to arbitrary input variables. As a result, the framework supports a wide range of constraints regarding partial derivatives of the learned function $\mathbf{y}$. Some examples are

- $y_1(\mathbf{u})$ *is monotonously increasing with respect to the first input variable* u_1,
- $y_3(\mathbf{u})$ *is larger than or equal to* $c \in \mathbb{R}$ or
- *The slope of* y_2 *in the direction* $\mathbf{d} \in \mathbb{R}^{N_u}$ *is larger than or equal to* $c \in \mathbb{R}$.

A constraint can be defined on the entire input space or only on particular subdomains. An example of the impact of such constraints is visualized in Figure 1.2. A monotonicity constraint is defined on the one-dimensional output y with respect to the one-dimensional input u. Since only few and noisy training data is available, the standard ELM without constraints tends to strong overfitting (Figure 1.2a). In contrast, the CELM enforces the monotonicity of the output function and provides a much better generalization compared to the ELM without constraints (Figure 1.2b).

Formally, a constraint L is defined as a function of an input $\mathbf{u} \in \mathbb{R}^{N_u}$ as

$$L(\mathbf{u}) = \sum_{i=1}^{N_y} \gamma_i D^{\mathbf{m}^i} y_i(\mathbf{u}) - c \tag{1.3}$$

with a bound $c \in \mathbb{R}$, coefficients $\gamma_i \in \mathbb{R}$, vectors $\mathbf{m}^i = [m_1^i \dots m_{M^i}^i] \in \{1, \dots, N_u\}^{M^i}$ defining the input variables to which the differentiations are carried out and

$$D^{\mathbf{m}^i} = \frac{\partial^{M^i}}{\partial u_{m_1^i} \dots \partial u_{m_{M^i}^i}} \tag{1.4}$$

the component-wise differential operator. The constraint is forced as $L(\mathbf{u}) \leq 0$ which is equivalent to $\sum_{i=1}^{N_y} \gamma_i D^{\mathbf{m}^i} y_i(\mathbf{u}) \leq c$. For example, the monotonicity constraint applied in Figure 1.2b can be expressed in this framework as

$$L(\mathbf{u}) = -1 \cdot D^{[1]} y_1(\mathbf{u}) = -1 \cdot \frac{\partial}{\partial u_1} y_1(\mathbf{u}) = -1 \cdot \frac{\partial}{\partial u} y(u), \tag{1.5}$$

where $c = 0$ can be omitted and $\gamma_1 = -1$. The constraint is evaluated as $\partial/\partial u\, y(u) \geq 0$. Formally, the input and the output of the neural network are both vectors $\mathbf{u} = [u_1 \dots u_{N_u}]^T$ and $\mathbf{y} = [y_1 \dots y_{N_y}]^T$. But as already done in the previous example, if the vectors are one-dimensional, we will write them as scalars u and y in this work.

1.3 Evaluation Metrics

By embedding constraints into a neural network, we deliberately influence and restrict the range of possible outputs of the network. As a result, mathematically, the training error cannot be smaller with constraints than without. It is even likely that the training error becomes much bigger, especially if the training target outputs do not satisfy the constraints. In the case the training samples are good representatives of the entire data domain, the same will also hold for the test error. This behavior is excepted, since the primary goal is no longer to minimize the training error, but to minimize the training error while satisfying the constraints. This is a fact we always have to keep in mind when evaluating machine learning models dealing with constraints since the quality of a model can no longer necessarily be determined by the test error only. There are different use cases of the constraint embedding, which result in different evaluation possibilities:

- Suppose only a small amount of training data is available or the training data is very noisy. In that case, constraints can help to learn a function that is closer to the real one and thus provide a better generalization capability. The test error should be smaller, if suitable constraints are used and the performance of the constrained model can be evaluated on a test dataset as usual.

- The second use case is similar to the first one. Here, the problem is not too few or too noisy training data in the usual sense, but the training data does not represent the desired network output. So-called ground-truth data is available, which is data representing the expected network outputs well, especially the satisfaction of the constraints. Using this data in a final evaluation, the test error should be a good measurement of the quality of the model. Here, we assume that this data is not available for training, for example because only very few ground-truth samples are available.

- In contrast to the first use cases, it is hardly possible to evaluate a model in the sense of a test error if the constraints are embedded just because they are required, but not to assist the training process. An example is that the model is only one part of a

data pipeline and the following step requires specific constraints to its input data. In this case, the primary evaluation metric should deal with the satisfaction of the constraints. Evaluation metrics regarding the test error of the model can only be compared between models that all satisfy the constraints.

1.4 Outline

In this thesis, the Constrained Echo State Network (CESN), a recurrent neural network embedding prior knowledge in the form of constraints, is introduced based on the work of Neumann, Rolf and Steil [34]. After introducing the topic in the first chapter, the basic CESN model is introduced and elaborated theoretically in Chapter 2. The practical realization, including the algorithmic implementation and an illustrative example, is described in Chapter 3. In Chapter 4, an application of the CESN is presented, which deals with the forecasting of satellite image time series of woodland areas. Finally, an extension of the CESN is presented in Chapter 5, tackling the task of recursive predictions.

2 The Constrained Echo State Network

In the following, an introduction to the basic ESN approach is given. A framework to define constraints for ESNs is elaborated and finally, it is described theoretically how these constraints can be embedded into the network during the training process.

2.1 The Basic Echo State Network

Echo State Networks (ESNs) are recurrent neural networks from the field of Reservoir Computing. They are applied to supervised machine learning tasks with temporal dynamic behavior and, due to their similar network structure, can be considered as the recurrent counterpart to the ELM.

When dealing with temporal data, we no longer talk about single inputs $\mathbf{u}$ and outputs $\mathbf{y}$, but about input and output signals $\mathbf{u}(t)$ and $\mathbf{y}(t)$, which change with respect to a time t. In the case of ESNs, these signals are defined on a discrete time scale $t = 1, 2, \ldots, T$ where $T \in \mathbb{N}$ is the number of data points in the signals.

Like the ELM, an ESN consists of one layer collecting the N_u-dimensional input $\mathbf{u}(t) \in \mathbb{R}^{N_u}$ at time step t, one layer that produces the N_y-dimensional output $\mathbf{y}(t) \in \mathbb{R}^{N_y}$ at time step t and one hidden layer, the so-called reservoir. In between, the input layer is fully-connected to the high-dimensional, recurrent reservoir, which induces a high-dimensional, non-linear reservoir output $\mathbf{x}(t)$. We use the notation from Lukoševičius [25], where the reservoir equation is defined as

$$\mathbf{x}(t) = (1-\alpha)\underbrace{\mathbf{x}(t-1)}_{\text{previous reservoir state}} + \underbrace{\alpha\,\sigma\left(\mathbf{W}^{\text{in}}\begin{bmatrix}1\\ \mathbf{u}(t)\end{bmatrix} + \mathbf{W}\mathbf{x}(t-1)\right)}_{\text{reservoir update}}. \tag{2.1}$$

Here, the activation function $\sigma(\cdot)$ of the reservoir update is applied element-wise. We use the hyperbolic tangent $\tanh(\cdot)$, but other sigmoid activation functions can, of course, also be used. $\mathbf{W}^{\text{in}} \in \mathbb{R}^{N_x \times N_u + 1}$ and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ are the input and reservoir weight matrices, respectively and $\mathbf{x}(t-1)$ is the previous reservoir state, responsible for the recurrence of the network. The reservoir output $\mathbf{x}(t) \in \mathbb{R}^{N_x}$ is computed as a convex combination of the previous reservoir state $\mathbf{x}(t-1)$ and the reservoir update. This so-called leaky-integration [19] can be used to influence the network's temporal characteristics. A small leaking rate α can simulate slow temporal dynamics. The final read-out layer consists of additive units, fully-connected with the reservoir layer. The network output is then computed as a linear combination of the input $\mathbf{u}(t)$, the reservoir output $\mathbf{x}(t)$ and a bias,

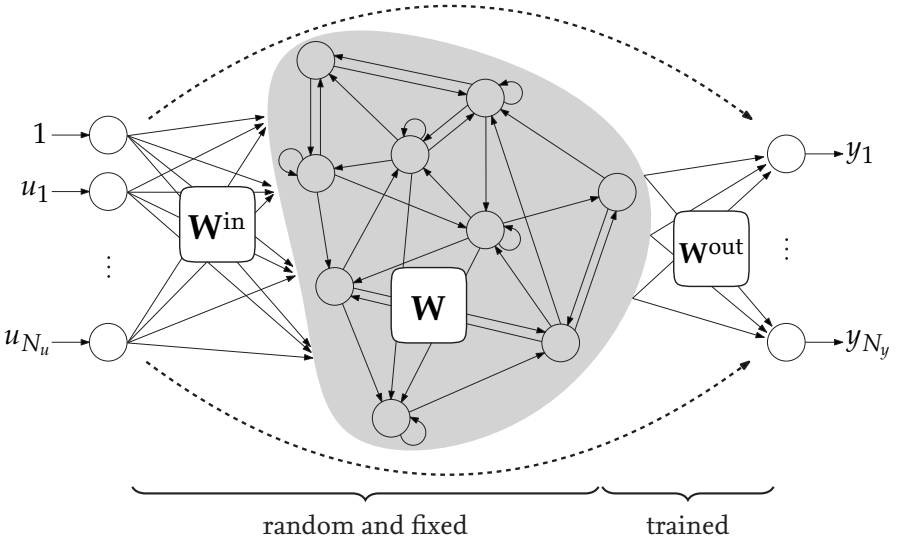

Figure 2.1: The input layer is fully-connected to the reservoir layer, which in turn is fully-connected to the read-out layer. The reservoir is basically fully self-connected, but its sparsity results from the sparsity of the weight matrix $\mathbf{W}$. While only the read-out weights $\mathbf{W}^{\text{out}}$ are optimized, all other weights are randomly assigned and fixed.

where the coefficients are provided by the output weight matrix $\mathbf{W}^{\text{out}} \in \mathbb{R}^{N_y \times N_u + N_x + 1}$:

$$\mathbf{y}(t) = \mathbf{W}^{\text{out}} \begin{bmatrix} 1 \\ \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix}. \tag{2.2}$$

An illustrative network diagram, visualizing the ESN structure, is given in Figure 2.1. Like the ELM, the ESN takes advantage of the read-out layer's linearity. The recurrence of the network can be decoupled entirely from the training process since the reservoir weights are fixed and assigned independently from the training process. Hence, as with the ELM, the ESN training can also be regarded as a linear regression problem, as described in detail later in this chapter.

2.2 Constraints for the ESN

The substantial similarity in the network structure between the feedforward ELM and the recurrent ESN suggests adapting the CELM approach to the recurrent ESN. But even though the two approaches are similar in their structures, two major differences must be tackled when transferring the concept of constraint learning to the ESN. On the one hand, the time as an additional dimension needs to be taken into account and on the other hand, the discreteness of the network needs to be considered. We will discuss different constraint types and analyze which of them are suited to deal with these aspects of the ESN.

Continuous vs. Discrete Time

The CELM is defined by a continuous, differentiable network function $\mathbf{y}$ over a continuous input domain $\mathbf{u}$. This is the reason, why the constraints can be defined as linear combinations of partial derivatives of $\mathbf{y}$ with respect to $\mathbf{u}$ as stated in Equation 1.3. In contrast, the output function $\mathbf{y}$ of the ESN is defined as a discrete function over a discrete time sequence $t = 1, \ldots, T$. In order to discuss the definition of constraints for the ESN in the following subsection, we make use of an alternative, continuous definition of $\mathbf{y}$. While the actual network equation can be regarded as a continuous function mathematically, the discreteness of the system is caused by the reservoir state update. The reservoir state is updated on each discrete network input (or after each discrete time step respectively) and occurs in the network equation as $\mathbf{x}(t-1)$. To circumvent this discrete state update in a continuous version of $\mathbf{y}$, we consider the continuous reservoir equation $\mathbf{x}^{\text{cont}}$ and the continuous ESN equation $\mathbf{y}^{\text{cont}}$ as functions of the input $\mathbf{u} \in \mathbb{R}^{N_u}$ and the internal reservoir state $\mathbf{x}^{\text{prev}} \in \mathbb{R}^{N_x}$. They are defined as

$$\mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) = (1-\alpha)\mathbf{x}^{\text{prev}} + \alpha\sigma\left(\mathbf{W}^{\text{in}}\begin{bmatrix} 1 \\ \mathbf{u} \end{bmatrix} + \mathbf{W}\mathbf{x}^{\text{prev}}\right), \tag{2.3}$$

$$\mathbf{y}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) = \mathbf{W}^{\text{out}}\begin{bmatrix} 1 \\ \mathbf{u} \\ \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix}. \tag{2.4}$$

Here, the discreteness is eliminated by regarding the reservoir state as a continuous function argument instead of using it implicitly. The relation between the discrete and the continuous network equation can be expressed as

$$\mathbf{y}(t) = \mathbf{y}^{\text{cont}}(\mathbf{u}(t), \mathbf{x}(t-1)),$$

where $\mathbf{u}(t)$ is the t-th input of the discrete time series and $\mathbf{x}(t-1)$ the internal reservoir state after $t-1$ time steps.

Input-Dependent Constraint

Using this definition of a continuous learning function $\mathbf{y}^{\text{cont}}$ as introduced in Equation 2.4, the adaption of the CELM constraints defined in Equation 1.3 to the ESN case is straightforward: For a given input $\mathbf{u}$ and an internal reservoir state $\mathbf{x}$, a constraint is defined as

$$L(\mathbf{u}, \mathbf{x}) = \sum_{i=1}^{N_y} \gamma_i D^{\mathbf{m}^i} y_i^{\text{cont}}(\mathbf{u}, \mathbf{x}) - c \tag{2.5}$$

with c, γ_i, $\mathbf{m}^i$ and $D^{\mathbf{m}^i}$ defined as above. The partial derivative of y_i^{cont} with respect to $u_{m_1^i}, \ldots, u_{m_{M^i}^i}$ is obtained as

$$D^{\mathbf{m}^i}\mathbf{y}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) = \mathbf{W}^{\text{out}}\begin{bmatrix} 0 \\ D^{\mathbf{m}^i}\mathbf{u} \\ D^{\mathbf{m}^i}\mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix}, \tag{2.6}$$

where the derivatives of $\mathbf{u}$ and $\mathbf{x}^{\text{cont}}$ result as

$$D^{\mathbf{m}^i}\mathbf{u} = \begin{cases} \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^T & \text{if } M^i = 1, \\ \begin{bmatrix} 0 & \dots & 0 & 0 & 0 & \dots & 0 \end{bmatrix}^T & \text{otherwise,} \end{cases} \tag{2.7}$$

$\uparrow m_1^i$-th position

and

$$D^{\mathbf{m}^i}\mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) = \alpha \cdot \sigma^{(M^i)}\left(\mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u}(t) \end{bmatrix} + \mathbf{W}\mathbf{x}^{\text{prev}}\right) \cdot \mathbf{W}^{\text{in}}_{\cdot m_1^i} \cdots \mathbf{W}^{\text{in}}_{\cdot m_{M^i}^i}, \tag{2.8}$$

as shown in Appendix A1. Here, $\sigma^{(M^i)}$ is the the M^i-th derivative of σ and $\mathbf{W}^{\text{in}}_{\cdot j}$ the j-th column of $\mathbf{W}^{\text{in}}$.

Although the constraint definition in Equation 2.5 is mathematically completely valid and can be implemented, some discrepancies arise on a logical level. These issues are explained by a more detailed look at the constraint interpretation. In both the ELM and the ESN case, constraints are defined about partial derivatives of the learning function with respect to the input. The (partial) derivative of a function measures the sensitivity of the function value with respect to the input value. Loosely speaking, a constraint specifies how a small change in the input $\mathbf{u}$ affects the output $\mathbf{y}$. For the ELMs case, where the network output depends only on the current input, it makes perfect sense. But when dealing with ESNs, the network output is computed not only from the current input, but also from the internal reservoir state $\mathbf{x}^{\text{prev}}$, derived from the input history. With this in mind, a constraint gets a different meaning: It specifies how a small change in the input $\mathbf{u}$ affects the output $\mathbf{y}$, if and only if the internal reservoir state (and so the input history) stays the same. Consider the following example: Given two time series $\mathbf{u}(t)$ and $\mathbf{u}'(t)$ with $\mathbf{u}(1) = \mathbf{u}'(1), \dots, \mathbf{u}(i) = \mathbf{u}'(i)$, which means that both time series share the same input history up to the i-th data point. A possible constraint supported by the framework could look like this: If $\mathbf{u}(i+1)$ is smaller than or equal to $\mathbf{u}'(i+1)$ then $\mathbf{y}(i+1) \leq \mathbf{y}'(i+1)$. Even if it works, it seems quite unlikely that this kind of constraint makes sense for any time series machine learning task.

In order to get a more useful constraint framework, we introduce a new kind of constraint: Discrete time-dependent constraints, discussed in the following section.

Discrete Time-Dependent Constraints via Continuous Network Function

As already mentioned above, a constraint as defined so far influences the effect of a small change in the input values to the network output. But when dealing with RNNs, we also deal with an additional dimension, the time. Thus, this *small change in the input* can not only be interpreted as a small change in the input domain, but also as a small change in time. As can be seen later, it even looks quite natural to define constraints regarding the behavior of the network with respect to the time. Since ESNs are defined on time series with a discrete time scale, we consider the network behavior between two or more

consecutive time steps, denoted as Δt. Since the term *derivative* is usually only defined for continuous, differential functions, we have to introduce the term *discrete derivative* first. By a discrete derivative of n-th order we mean the n-th order difference quotient (a more detailed introduction to difference quotients is given in the following section). For now, we consider only the first-order backward difference quotient for $\Delta t = 1$ of the form

$$\frac{\Delta}{\Delta t} f(t) = f(t) - f(t-1). \tag{2.9}$$

To express constraints on this discrete derivative while remaining as close as possible in the context of the original constraint definition, the discrete derivative with respect to the time can be approximated using the chain rule $(f \circ g)' = (f' \circ g) \cdot g'$ as

$$\begin{aligned}
\frac{\Delta}{\Delta t}\mathbf{y}(t) &= \frac{\Delta}{\Delta t}\mathbf{y}^{\text{cont}}(\mathbf{u}(t), \mathbf{x}(t-1)) \\
&\approx \underbrace{\frac{\partial \mathbf{y}^{\text{cont}}}{\partial \mathbf{u}} \frac{\Delta \mathbf{u}(t)}{\Delta t}}_{*} + \underbrace{\frac{\partial \mathbf{y}^{\text{cont}}}{\partial \mathbf{x}} \frac{\Delta \mathbf{x}(t-1)}{\Delta t}}_{**} \\
&= \sum_{i=1}^{N_u} \frac{\partial}{\partial u_i} \mathbf{y}^{\text{cont}}(\mathbf{u}(t), \mathbf{x}(t-1)) \cdot (u_i(t) - u_i(t-\Delta t)) + \\
&\quad \sum_{i=1}^{N_x} \frac{\partial}{\partial x_i} \mathbf{y}^{\text{cont}}(\mathbf{u}(t), \mathbf{x}(t-1)) \cdot (x_i(t-1) - x_i(t-1-\Delta t)),
\end{aligned} \tag{2.10}$$

where $\partial \mathbf{y}^{\text{cont}} / \partial \mathbf{u}$ is the element-wise partial derivative with respect to each element of $\mathbf{u}$. The constraint definition adapted in Equation 2.5 in its original form cannot represent this expression. The first part Equation 2.10$*$ is a linear combination of partial derivatives of $\mathbf{y}$ with respect to $\mathbf{u}$ with $\Delta \mathbf{u}(t) / \Delta t$ as coefficients and thus fits in the constraint framework presented above. But the framework does not allow to define expressions regarding derivatives with respect to the reservoir state $\mathbf{x}$ as required in Equation 2.10$**$. Here, we need to extend the framework accordingly by adding a term taking the $\mathbf{x}$-derivatives into account:

$$L(\mathbf{u}, \mathbf{x}) = \sum_{i=1}^{N_y} \gamma_i D_{\mathbf{u}}^{\mathbf{m}^i} y_i^{\text{cont}}(\mathbf{u}, \mathbf{x}) + \underbrace{\hat{\gamma}_i D_{\mathbf{x}}^{\hat{\mathbf{m}}^i} y_i^{\text{cont}}(\mathbf{u}, \mathbf{x})}_{\text{extension}} - c, \tag{2.11}$$

with coefficients $\gamma_i, \hat{\gamma}_i \in \mathbb{R}$, a bound $c \in \mathbb{R}$ and $\mathbf{m}^i = [m_1^i \dots m_{M^i}^i] \in \{1, \dots, N_u\}^{M^i}$ and $\hat{\mathbf{m}}^i = [\hat{m}_1^i \dots \hat{m}_{\hat{M}^i}^i] \in \{1, \dots, N_x\}^{\hat{M}^i}$. $D_{\mathbf{u}}^{\mathbf{m}^i}$ and $D_{\mathbf{x}}^{\hat{\mathbf{m}}^i}$ are the component-wise differential operator as defined above, but with the explicit distinction of the vector to which $\mathbf{m}^i$ and $\hat{\mathbf{m}}^i$, respectively, refer. The partial derivative of $\mathbf{y}^{\text{cont}}$ with respect to $\mathbf{x}$ is obtained similar to the one with respect to $\mathbf{u}$ as given in Equation 2.6 and is shown in Appendix A1.

Using Equation 2.11, we can finally approximate derivatives of the ESN output functions with respect to the time t. But having a look at Equation 2.10, one can see that it is still an approximation caused by the continuous partial derivatives of the continuous function $\mathbf{y}^{\text{cont}}$ used to calculate the discrete derivative of the discrete function $\mathbf{y}$. The problem here is that we assume that the derivative of $\mathbf{y}^{\text{cont}}$ at $(\mathbf{u}(t), \mathbf{x}(t))$ is similar to the discrete

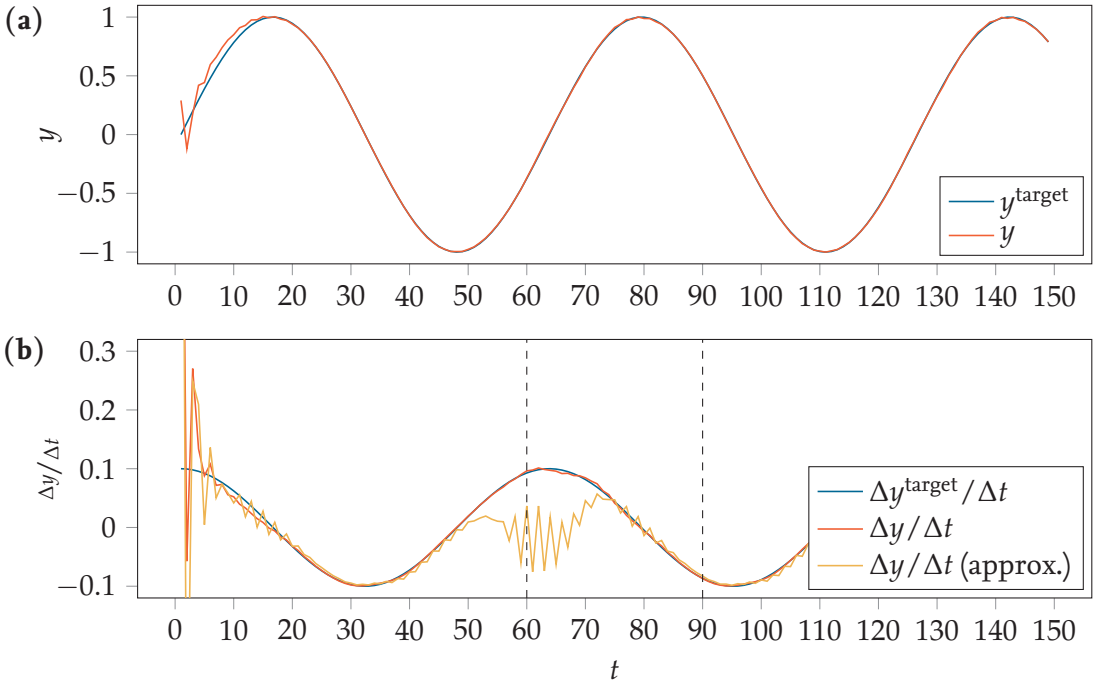

Figure 2.2: **(a)** After an initial washout time, the predicted signal y matches the target signal y^{target} quite well. **(b)** Nevertheless, the approximated discrete derivative deviates strongly from the "real" discrete derivative at some points (e.g., $t = 60$), while it approximates the discrete derivative well at other points (e.g., $t = 90$).

derivative of **y** at time step t. That this assumption does not hold in general and may yield completely wrong results in the worst case could be reproduced in the following example.

Consider the simple learning task of passing through the input signal with one time step delay. The input signal is generated from a sine wave as

$$u(t) = \sin\left(\frac{t}{10}\right) \tag{2.12}$$

for $t = 1, 2, 3, \ldots$ and the target signal is obtained as

$$y^{\text{target}}(t) = u(t-1). \tag{2.13}$$

We create an ESN with a very small reservoir of only ten reservoir neurons and train it with the given training data. Without embedding any constraint into the learning process, we analyze the approximation of the first-order discrete derivative of the learned function as described in Equation 2.10 (in the following called *approximation*). We compare this approximation to the first-order difference quotient of y

$$\frac{\Delta y}{\Delta t} = y(t) - y(t-1), \tag{2.14}$$

which matches the definition of the discrete derivative and thus is a good baseline model for comparison. The result is visualized in Figure 2.2b. Both the discrete derivative of the

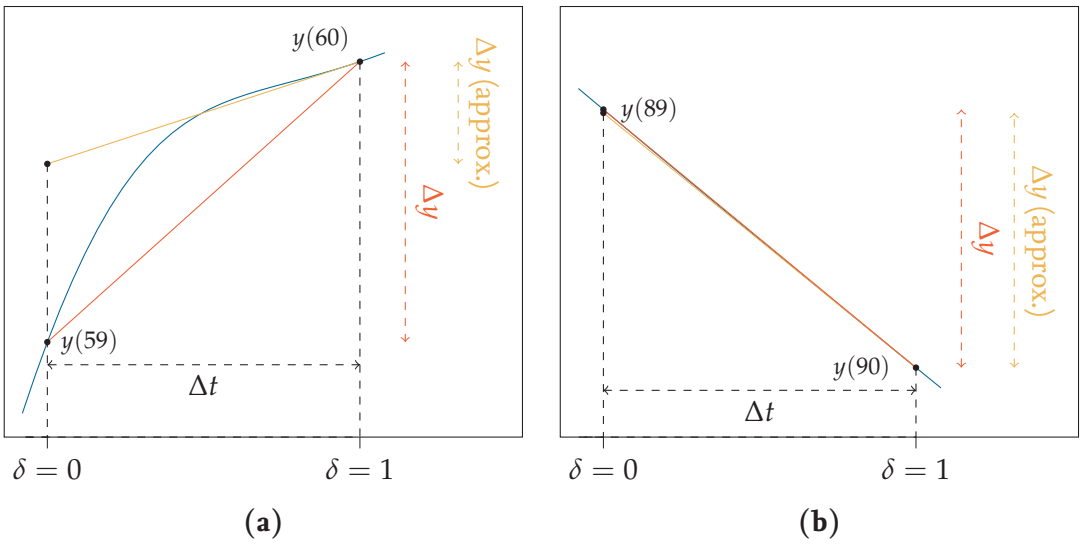

Figure 2.3: **(a)** y_{cont} in the direction $(u(t)-u(t-1),\ \mathbf{x}(t-1)-\mathbf{x}(t-2))$ (——) behaves non-linear between $t=60$ and $t=61$. **(b)** Between $t=90$ and $t=91$, it is almost linear. Since the derivation of y_{cont} at $t=60$ or $t=90$ respectively is used to approximate the discrete derivative of y, the approximation works well only if y_{cont} is nearly linear.

ESN output $\Delta y/\Delta t$ and the discrete derivative of the target signal $\Delta y^{\text{target}}/\Delta t$ are very similar. On the contrary, the approximation deviates strongly from these at some points. To better understand the cause, we consider the behavior at time steps $t=60$, where the result differs strongly from the expected one and $t=90$, where the result is very similar to the expected one. In the following, we discuss the behavior using the continuous ESN equation y_{cont}. The "real" discrete derivative is the difference between $y(t)=y_{\text{cont}}(u(t),\mathbf{x}(t-1))$ and $y(t-1)=y_{\text{cont}}(u(t-1),\mathbf{x}(t-2))$. The approximated one is computed by generalizing the directional partial derivative of $y_{\text{cont}}(u(t),\mathbf{x}(t-1))$ along the vector

$$\begin{bmatrix} u(t)-u(t-1) \\ \mathbf{x}(t-1)-\mathbf{x}(t-2) \end{bmatrix} \tag{2.15}$$

to the whole interval between these time steps. This is visualized in Figure 2.3, where the directional output function

$$y_{\text{cont}}((1-\delta)u(t)+\delta u(t-1),\ (1-\delta)\mathbf{x}(t-1)+\delta\mathbf{x}(t-2)) \tag{2.16}$$

is plotted with respect to δ. For $\delta=0$, the function value equals $y(t-1)$, for $\delta=1$ it equals $y(t)$. In between, the values of u and $\mathbf{x}$ are interpolated linearly. Furthermore, the slope at $\delta=1$ equals the directional derivative used in the approximation. One can see that this continuous output function y_{cont} behaves non-linear between $t=59$ and $t=60$, but almost linear between $t=89$ and $t=90$. As a result, the approximation $\Delta y/\Delta t$ (approx.) is very similar to the actual $\Delta y/\Delta t$ in the latter case, but very different in the first one. As a result, the approximation works well only in the latter case.

This example is constructed so that the issue becomes visible in the best possible manner. On the one hand, the reservoir size is selected very small and on the other hand, the hyperparameters are not optimized in the sense of a validation error, but are selected in the sense of a good exposition of the issue. Nevertheless, the small reservoir and the hyperparameter selection are sufficient to yield a small test error, as visualized in Figure 2.2a. But even though this approximation seems to fail only in particular cases, the example shows that this approach is not reliable to embed time-dependent constraints to the ESN at all. Thus, we introduce another kind of constraint that is able to describe difference quotients of the output function $\mathbf{y}$ directly and therefore, the approximation used here can be omitted.

Discrete Time-Dependent Constraints via Discrete Network Function

The difference quotient of a function $f(x)$ describes the ratio of the change of the function value f to a change of of the argument x, provided that f depends on x. In numerical mathematics, the difference quotient is used to approximate the derivative of differentiable functions and can also be used to define discrete derivatives on discrete functions. The difference quotient can be defined in three different ways:

- Forward difference quotient

$$\frac{\Delta f}{\Delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{2.17}$$

- Backward difference quotient

$$\frac{\Delta f}{\Delta x} = \frac{f(x) - f(x-\Delta x)}{\Delta x} \tag{2.18}$$

- Central difference quotient

$$\frac{\Delta f}{\Delta x} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{2.19}$$

While this standard (first-order) difference quotient approximates only the first-order derivative of a function, there exist higher-order (central) difference quotients, approximating higher-order derivatives. For example, the (central) difference quotient of second-order is defined as

$$\frac{\Delta^2 f}{\Delta x^2} = \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2}. \tag{2.20}$$

The goal is to define a framework, which allows defining constraints regarding difference quotients of the ESN output of arbitrary order with respect to the time. Applied to the ESN, the function f is one network output function y_i and the independent variable x is the discrete time t. Regarding the structure of a difference quotient, it can already be seen in the examples above that they are built as linear combinations of y_i at different

time steps. For technical simplicity, we shift all difference quotients by so many time steps that we have to deal only with backward difference quotients. For example, a constraint at time step t, which involves a central second-order difference quotient, is shifted by Δt time steps. The resulting constraint is defined at time step $t + \Delta t$ and involves a difference quotient of the form

$$\frac{y_i(t) - 2y_i(t - \Delta t) + y_i(t - 2\Delta t)}{2\Delta t}. \tag{2.21}$$

Both expressions are equivalent. This provides us the same expression power, but we have to deal with the network history only and no longer with values beyond time step t. With this in mind, we define the new constraints with respect to the discrete time t as

$$L(t) = \sum_{i=1}^{N_y} \sum_{h=0}^{H} \gamma_{ih}\, y_i(t - h) - c \tag{2.22}$$

with coefficients $\gamma_{ih} \in \mathbb{R}$, a bound $c \in \mathbb{R}$ and $H \in \mathbb{N} \cup \{0\}$ the number of time steps accessed backward. This new constraint definition allows us to define constraints involving the difference quotients introduced above and all higher-order difference quotients as well. Thus, this kind of constraint can influence the behavior of the ESN with respect to the discrete time. Up to this point, we just discussed the constraints' general structure. The following section describes how the actual embedding of the constraints into the ESN works.

2.3 Embedding Constraints into the Network

The objective of the training process of a recurrent network is usually to minimize an error between the training target output $\mathbf{y}^{\text{target}}$ and the actual network output $\mathbf{y}$. Considering the CESN, we furthermore have to regard the special case that we need to embed predefined constraints into the network. Figure 2.4 visualizes the idea behind the training process. First, the training task can be considered as a two-dimensional Linear Least Squares (LLS) regression problem, which in turn can be reduced to a one-dimensional one. At this point, the constraints come into the picture by considering the one-dimensional LLS as a Constrained Linear Least Squares regression problem. This problem again can be rephrased as a non-linear optimization problem and in the end solved using a suitable solver. In the following, these single reduction steps are explained mathematically without going into further details about the actual algorithmic implementation, which is discussed separately in Chapter 3.

Two-Dimensional Linear Least Squares

For simplicity, we assume in the following that the training data consists of exactly one training input time series $\mathbf{u}(1), \ldots, \mathbf{u}(T)$ with $T \in \mathbb{N}$ time steps and one corresponding

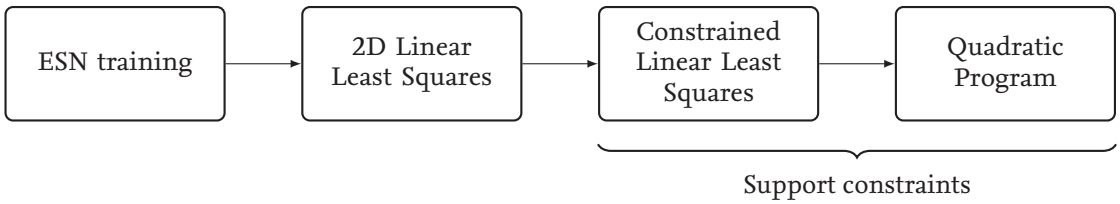

Figure 2.4: The training of the ESN in the sense of minimizing the training error is equivalent to solving a two-dimensional linear least squares problem, which in turn can be reduced to a linear least squares problem supporting constraints. This can be rephrased as a quadratic program and in the end solved by a suitable solver.

target output time series $\mathbf{y}^{\text{target}}(1), \ldots, \mathbf{y}^{\text{target}}(T)$. Furthermore, we substitute

$$\tilde{\mathbf{x}}(t) := \begin{bmatrix} 1 \\ \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} \in \mathbb{R}^{N_{\tilde{x}}}, \tag{2.23}$$

where $N_{\tilde{x}} = N_u + N_x + 1$. With this, the reservoir read-out equation from Equation 2.2 can be expressed as

$$\mathbf{y}(t) = \mathbf{W}^{\text{out}} \tilde{\mathbf{x}}(t). \tag{2.24}$$

The training objective of the ESN is to minimize the error ϵ between the training target output signal $\mathbf{y}^{\text{target}}$ and the actual network output $\mathbf{y}$, using a suitable error metric. In our case, this error metric is the Root Mean Square Error (RMSE), defined as

$$\begin{aligned} \epsilon_{\text{RMSE}} &= \frac{1}{N_y} \sum_{i=1}^{N_y} \frac{1}{T} \sum_{t=1}^{T} \left(y_i(t) - y_i^{\text{target}}(t) \right)^2 \\ &= \frac{1}{N_y} \sum_{i=1}^{N_y} \frac{1}{T} \sum_{t=1}^{T} \left(\mathbf{W}_i^{\text{out}} \tilde{\mathbf{x}}(t) - y_i^{\text{target}}(t) \right)^2. \end{aligned} \tag{2.25}$$

After initializing the reservoir with randomly selected and fixed input weights $\mathbf{W}^{\text{in}}$ and reservoir weights $\mathbf{W}$ and selecting suitable hyperparameters, the only parameters being optimized during the training process are the read-out weights $\mathbf{W}^{\text{out}}$. Collecting $\tilde{\mathbf{x}}(t)$ and $\mathbf{y}^{\text{target}}(t)$ over all training time steps in matrices $\tilde{\mathbf{X}} \in \mathbb{R}^{N_{\tilde{x}} \times T}$ and $\mathbf{Y}^{\text{target}} \in \mathbb{R}^{N_y \times T}$ respectively, the definition of the RMSE can be squashed to

$$\epsilon_{\text{RMSE}} = \left|\left| \tilde{\mathbf{X}}^T (\mathbf{W}^{\text{out}})^T - (\mathbf{Y}^{\text{target}})^T \right|\right|_F^2, \tag{2.26}$$

where $||\cdot||_F$ is the Frobenius norm, which is the matrix counterpart to the euclidean vector norm and defined as the square root of the sum of the squared matrix elements. Now, minimizing ϵ_{RMSE} with respect to $\mathbf{W}^{\text{out}}$ is equivalent to solving the two-dimensional LLS

$$\mathbf{W}^{\text{out}} = \arg\min_{\mathbf{W}^{\text{out}}} \left|\left| \tilde{\mathbf{X}}^T (\mathbf{W}^{\text{out}})^T - (\mathbf{Y}^{\text{target}})^T \right|\right|_F^2. \tag{2.27}$$

Adding a regularization term may help keep the weights in $\mathbf{W}^{\text{out}}$ small and lead to more stable solutions. This so-called Ridge Regression extends the LLS to

$$\mathbf{W}^{\text{out}} = \arg\min_{\mathbf{W}^{\text{out}}} \left|\left|\tilde{\mathbf{X}}^T(\mathbf{W}^{\text{out}})^T - (\mathbf{Y}^{\text{target}})^T\right|\right|_F^2 + \beta \left|\left|(\mathbf{W}^{\text{out}})^T\right|\right|_F^2 \tag{2.28}$$

with the regularization parameter $\beta \in \mathbb{R}$. The larger β is selected, the more large output weights are penalized. A solution for the ridge regression problem in Equation 2.28 is obtained by solving

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\tilde{\mathbf{X}}^T \left(\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T + \beta \mathbf{I}_{N_{\tilde{x}}}\right)^{-1} \tag{2.29}$$

with the identity matrix $\mathbf{I}_{N_{\tilde{x}}} \in \mathbb{R}^{N_{\tilde{x}} \times N_{\tilde{x}}}$ and the multiplicative matrix inverse $(\cdot)^{-1}$ [25]. However, this solution is not able to learn constraints during the training process. Nevertheless, it is much faster than the constrained linear least squares solution presented in the next section and should be preferred whenever no constraints are involved.

Constrained Linear Least Squares

In order to embed constraints as defined in Equation 2.22 into the ESN, we regard the training process as a constrained optimization problem, which is solving a linear least squares problem while satisfying a set of predefined constraints. The so-called Constrained Linear Least Squares (CLLS) problem is formulated generically as

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & ||\mathbf{A}\mathbf{x} - \mathbf{b}||_2^2 \\ \text{subject to} \quad & \mathbf{G}\mathbf{x} \preceq \mathbf{h} \end{aligned} \tag{2.30}$$

for a design matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, observations $\mathbf{b} \in \mathbb{R}^m$ and constraints defined by $\mathbf{G} \in \mathbb{R}^{p \times n}$ and $\mathbf{h} \in \mathbb{R}^p$, where the $\preceq$ operator is defined as component-wise inequality between both operands[1]. Since the optimized variable $\mathbf{x}$ in the CLLS is one-dimensional, we show first, that the two-dimensional ridge regression problem can be reduced to a one dimensional LLS problem as required in Equation 2.30 and second that the constraints defined above fit into the resulting CLLS problem.

Reduction of Two-Dimensional Ridge Regression to Linear Least Squares

The two-dimensional ridge regression problem defined in Equation 2.28 can be reduced to a one-dimensional linear least squares problem as described in [33] in four steps:

[1] $\mathbf{x}$ is a common variable name in the formulation of a generic optimization problem and refers not to the reservoir output in this particular case, as in the rest of this work.

- Rephrase the design matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{N_{\tilde{x}} \times T}$ to a block-diagonal matrix as

$$\hat{\tilde{\mathbf{X}}} := \mathbf{I}_{N_y} \otimes \tilde{\mathbf{X}}^T = \begin{bmatrix} \tilde{\mathbf{X}}^T & 0 & 0 & \cdots & 0 \\ 0 & \tilde{\mathbf{X}}^T & 0 & \cdots & 0 \\ 0 & 0 & \tilde{\mathbf{X}}^T & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \tilde{\mathbf{X}}^T \end{bmatrix} \in \mathbb{R}^{(N_y \cdot T) \times (N_y \cdot N_{\tilde{x}})}, \tag{2.31}$$

  where $\otimes$ denotes the Kronecker product.

- Flatten the two-dimensional variable $\mathbf{W}^{\text{out}}$ to

$$\hat{\mathbf{W}}^{\text{out}} := \begin{bmatrix} (\mathbf{W}_1^{\text{out}})^T \\ (\mathbf{W}_2^{\text{out}})^T \\ \vdots \\ (\mathbf{W}_{N_{\tilde{x}}}^{\text{out}})^T \end{bmatrix} \in \mathbb{R}^{N_y \cdot N_{\tilde{x}}} \tag{2.32}$$

- Flatten the observation matrix $\mathbf{Y}^{\text{target}} \in \mathbb{R}^{N_y \times T}$ to

$$\hat{\mathbf{Y}}^{\text{target}} := \begin{bmatrix} (\mathbf{Y}_1^{\text{target}})^T \\ (\mathbf{Y}_2^{\text{target}})^T \\ \vdots \\ (\mathbf{Y}_{N_y}^{\text{target}})^T \end{bmatrix} \in \mathbb{R}^{N_y \cdot T} \tag{2.33}$$

- Construct a one-dimensional LLS problem that takes the regularization term into account as

$$\underset{\hat{\mathbf{W}}^{\text{out}}}{\text{minimize}} \quad \left\| \begin{bmatrix} \hat{\tilde{\mathbf{X}}} \\ \sqrt{\beta} \cdot \mathbf{I}_{N_y \cdot N_{\tilde{x}}} \end{bmatrix} \cdot \hat{\mathbf{W}}^{\text{out}} - \begin{bmatrix} \hat{\mathbf{Y}}^{\text{target}} \\ \mathbf{0}_{N_y \cdot N_{\tilde{x}}} \end{bmatrix} \right\|_2^2, \tag{2.34}$$

  where $\mathbf{0}_{N_y \cdot N_{\tilde{x}}}$ is the zero vector of size $N_y \cdot N_{\tilde{x}}$.

This resulting one-dimensional LLS with a one-dimensional optimization variable $\hat{\mathbf{W}}^{\text{out}}$ is equivalent to the two-dimensional one. The actual $\mathbf{W}^{\text{out}}$ is obtained by reshaping $\hat{\mathbf{W}}^{\text{out}}$ back to the initial shape of $\mathbf{W}^{\text{out}}$ according to the order in which $\hat{\mathbf{W}}^{\text{out}}$ was constructed from $\mathbf{W}^{\text{out}}$.

Constraint Reformulation

The constraints in the CLLS are defined as $\mathbf{G}\mathbf{x} \preceq \mathbf{h}$, where each row $\mathbf{G}_i\mathbf{x} \leq h_i$ represents one constraint. Applied to the currently defined LLS, a constraint is defined as

$$\mathbf{G}_i \hat{\mathbf{W}}^{\text{out}} \leq h_i \quad \Leftrightarrow \quad \mathbf{G}_i \cdot \begin{bmatrix} (\mathbf{W}_1^{\text{out}})^T \\ (\mathbf{W}_2^{\text{out}})^T \\ \vdots \\ (\mathbf{W}_{N_{\tilde{x}}}^{\text{out}})^T \end{bmatrix} \leq h_i \,. \tag{2.35}$$

In order to fit this formulation, the constraints defined in Equation 2.22 must be reformulated so that the optimization variable $\mathbf{W}^{\text{out}}$ is separated from the rest of the constraint expression. Due to the linearity of the read-out layer and the fact that the read-out can be completely decoupled from the reservoir, such a reformulation can be achieved as

$$\begin{aligned}
& \sum_{i=1}^{N_y} \sum_{h=0}^{H} \gamma_{ih}\, y_i(t-h) \leq c \\
\Leftrightarrow \quad & \sum_{i=1}^{N_y} \sum_{h=0}^{H} \gamma_{ih}\, \mathbf{W}_i^{\text{out}} \tilde{\mathbf{x}}(t-h) \leq c \\
\Leftrightarrow \quad & \sum_{i=1}^{N_y} \mathbf{W}_i^{\text{out}} \cdot \left(\sum_{h=0}^{H} \gamma_{ih}\, \tilde{\mathbf{x}}(t-h) \right) \leq c \\
\Leftrightarrow \quad & \sum_{i=1}^{N_y} \underbrace{\left(\sum_{h=0}^{H} \gamma_{ih}\, \tilde{\mathbf{x}}(t-h) \right)^T}_{\eta_i(t)} \cdot (\mathbf{W}_i^{\text{out}})^T \leq c.
\end{aligned} \tag{2.36}$$

This reformulated constraint can be embedded into the CLLS straightforwardly: Assuming a constraint shall be satisfied for all time steps t in the range $[t_1, t_2]$ with $0 \leq t_1 \leq t_2 \leq T$. We add the following rows to the CLLS:

$$\begin{bmatrix}
\eta_1(t_1) & \eta_2(t_1) & \cdots & \eta_{N_y}(t_1) \\
\eta_1(t_1+1) & \eta_2(t_1+1) & \cdots & \eta_{N_y}(t_1+1) \\
\vdots & \vdots & \ddots & \vdots \\
\eta_1(t_2-1) & \eta_2(t_2-1) & \cdots & \eta_{N_y}(t_2-1) \\
\eta_1(t_2) & \eta_2(t_2) & \cdots & \eta_{N_y}(t_2)
\end{bmatrix} \cdot \hat{\mathbf{W}}^{\text{out}} \leq \begin{bmatrix} c \\ c \\ \vdots \\ c \\ c \end{bmatrix}. \tag{2.37}$$

Since $\mathbf{W}^{\text{out}}$ is only required to compute the final output $\mathbf{y}$, but not to run the reservoir internally, all $\tilde{\mathbf{x}}(t)$ can be computed in advance without knowing the final read-out weights $\mathbf{W}^{\text{out}}$. Figure 2.5 illustrates how *unfolding in time* is used to unfold the reservoir up to H time steps backward to memorize and access the reservoir history easily. Furthermore, the figure visualizes the decoupling of the read-out layer from the rest of the network and shows why $\mathbf{W}^{\text{out}}$ is not required to compute the reservoir outputs. Hence, all η_i, and thus all constraints can be computed in advance before the actual optimization of $\mathbf{W}^{\text{out}}$ begins.

Quadratic Programming

Quadratic programming is a well-researched field of mathematical optimization. As a type of non-linear optimization, quadratic programming deals with solving mathematical optimization problems involving quadratic objective functions under linear constraints. Formally, such an optimization problem, a so-called Quadratic Program (QP), is formulated as

$$\begin{aligned}
\underset{\mathbf{x} \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & \mathbf{G}\mathbf{x} \preceq \mathbf{h}
\end{aligned} \tag{2.38}$$

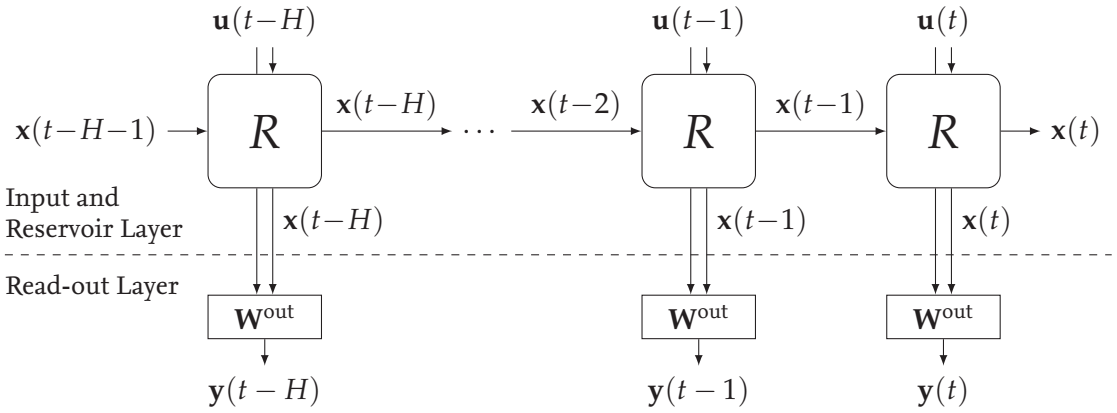

Figure 2.5: The input and reservoir layer are fully decoupled from the read-out layer. By using *Unfolding in time*, the reservoir history can be made accessible explicitly.

with $\mathbf{c} \in \mathbb{R}^n$, a symmetric matrix $\mathbf{Q} \in \mathbb{R}^{n \times n}$, $\mathbf{G} \in \mathbb{R}^{m \times n}$ and $\mathbf{h} \in \mathbb{R}^m$.[2]

One application of quadratic programming is the CLLS (Equation 2.30) which can be formulated as a specialization of a QP by setting

$$\mathbf{Q} := \mathbf{A}^T\mathbf{A} \quad \text{and} \quad \mathbf{c} := -2\mathbf{b}^T\mathbf{A} \tag{2.39}$$

as shown in Appendix A2. The constraints are defined equally as $\mathbf{Gx} \preceq \mathbf{h}$ in both problem formulations. With this in mind, we can rephrase the ESN training problem such that it can be solved as a QP using a suitable QP solver. For the sake of simplicity, we show only the reduction from the generic CLLS formulation to the generic QP formulation and do not transfer it to the ESN training problem.

By default, a constraint is defined in the so-called standard form $\mathbf{G}_i\mathbf{x} \leq h_i$. Other constraint types such as equality constraints or constraints involving absolute values, can easily be reduced to this standard form as listed in Table 2.1. In the following chapters, we use these reductions to use a broader range of constraint formulations.

Constraint	Standard form
$\mathbf{G}_i\mathbf{x} \geq h_i$	$-\mathbf{G}_i\mathbf{x} \leq -h_i$
$\mathbf{G}_i\mathbf{x} = h_i$	$\mathbf{G}_i\mathbf{x} \leq h_i$ $-\mathbf{G}_i\mathbf{x} \leq -h_i$
$\lvert\mathbf{G}_i\mathbf{x}\rvert \leq h_i$	$\mathbf{G}_i\mathbf{x} \leq h_i$ $-\mathbf{G}_i\mathbf{x} \leq h_i$

Table 2.1: Different types of constraints can be converted to constraints in standard form.

[2] $\mathbf{x}$ is a common variable name in the formulation of a generic optimization problem and refers not to the reservoir output in this particular case, as in the rest of this work.

3 Practical Realization

While the theoretical background of the CESN approach is elaborated in the previous chapter, the actual training algorithm, including two illustrative examples, is discussed in the following.

3.1 Algorithmic Implementation

Reservoir Initialization and Hyperparameters

The number of different hyperparameters influencing the performance of ESNs is, compared to other machine learning models, relatively large. Moreover, reservoir computing models are more sensitive to changes in the hyperparameters than other RNNs [2]. Thus, an extensive hyperparameter optimization is often essential when dealing with ESNs.

In the beginning of the training process, the ESN reservoir is initialized with input weights $\mathbf{W}^{\text{in}}$, recurrent reservoir weights $\mathbf{W}$ and an initial reservoir state $\mathbf{x}(0)$. The reservoir state is initialized with a zero vector $\mathbf{x}(0) = \mathbf{0}_{N_x}$. The weights $\mathbf{W}^{\text{in}}$ are sampled randomly from a uniform distribution $[-a, a]$ for an input scaling parameter $a \in \mathbb{R}$, which specifies the impact of the input on the reservoir update. The scale of the distribution from which the internal reservoir weights $\mathbf{W}$ are sampled is defined by the spectral radius parameter ρ. It specifies the spectral radius of $\mathbf{W}$, which is the largest absolute value of the eigenvalues of $\mathbf{W}$. The spectral radius of $\mathbf{W}$ influences how fast an input is vanished out of the reservoir state over time. For tasks that require a longer memory of the input, a greater spectral radius ρ should be selected. To obtain a weight matrix with a predefined spectral radius, $\mathbf{W}$ is first sampled randomly from the uniform distribution $[0, 1]$, then the current spectral radius is computed as

$$\rho_{\text{current}}(\mathbf{W}) = \max\{|\lambda_1|, \ldots, |\lambda_{N_x}|\}, \tag{3.1}$$

where $\lambda_1, \ldots, \lambda_{N_x}$ are the eigenvalues of $\mathbf{W}$ and finally $\mathbf{W}$ is scaled to match the target spectral radius:

$$\mathbf{W}_{\text{new}} = \rho \cdot \frac{\mathbf{W}}{\rho_{\text{current}}(\mathbf{W})}. \tag{3.2}$$

While several works address the question of what requirements the spectral radius must meet to deliver stable ESNs [3, 9, 17, 20, 48], it is often sufficient to find a well working spectral radius by a suitable hyperparameter optimization technique in practice. The sparsity of $\mathbf{W}$ defines the number of weights of size 0 in $\mathbf{W}$, which specifies the number of connections inside the reservoir. While the original ESN was proposed with a sparsely connected reservoir, it turned out that the sparsity is only a minor factor of hyperparameter optimization [25]. The leaking rate α is already known from the reservoir update definition

(Equation 2.1) and influences the temporal characteristics of the reservoir. Finally, the reservoir size is an important parameter, which specifies the capacity of the reservoir. It should be selected larger for more challenging tasks and tasks where a lot of input information needs to be remembered over time [25]. While all hyperparameters mentioned up to this point refer to the reservoir initialization, only one hyperparameter influences the actual network training, namely the regularization parameter β of the ridge regression, introduced in Equation 2.28. A detailed overview of all ESN hyperparameters is given in [25].

Training Data and Constraint Data

In general, a CESN is trained on a dataset of N_{tr} training samples

$$\left\{ \left(\mathbf{u}_1(t), \mathbf{y}_1^{\text{target}}(t) \right) \ \ldots, \left(\mathbf{u}_{N_{\text{tr}}}(t), \mathbf{y}_{N_{\text{tr}}}^{\text{target}}(t) \right) \right\},$$

where each sample consists of an input time series $\mathbf{u}_i(t)$ and a corresponding target time series $\mathbf{y}_i^{\text{target}}(t)$. Moreover, another set of N_{co} input time series $\{\mathbf{u}_1^{\text{co}}(t), \ldots, \mathbf{u}_{N_{\text{co}}}^{\text{co}}(t)\}$ is needed, on which the constraints are learned. Note that no further target output time series are needed here since only the input and the reservoir output, which is computed from the input, are required to embed the constraints to the CESN. This dataset for the constraint embedding does not necessarily have to be disjoint from the training dataset, but may also be a subset or a superset of the input time series of the training samples.

For simplicity, we discard the concept of multiple time series samples in the following section and assume that we have to deal with single, large time series only. Thus, the CESN is trained on one input time series $\mathbf{u}(1), \ldots, \mathbf{u}(T_{\text{tr}})$ and associated target outputs $\mathbf{y}^{\text{target}}(1), \ldots, \mathbf{y}^{\text{target}}(T_{\text{tr}})$ and the constraints are learned on another input time series $\mathbf{u}^{\text{co}}(1), \ldots, \mathbf{u}^{\text{co}}(T_{\text{co}})$. In practice, training data consisting of multiple samples of time series can easily be converted to one large time series by joining them together. Only care must be taken to reset the internal reservoir state whenever a new sample begins.

Network Training

The training procedure of the CESN is summarized in Algorithm 1. The procedure takes the training and the constraint input signal, the target output signal and the constraints as input and computes the optimized read-out weights $\mathbf{W}^{\text{out}}$. Here, the matrices $\mathbf{U} \in \mathbb{R}^{T_{\text{tr}} \times N_u}$ and $\mathbf{U}^{\text{co}} \in \mathbb{R}^{T_{\text{co}} \times N_u}$ collect the training input signal and the constraint input signal, respectively, over all time steps. The same holds for $\mathbf{Y}^{\text{target}} \in \mathbb{R}^{T_{\text{tr}} \times N_y}$. The constraints are defined as sets $\mathcal{C}_1, \ldots, \mathcal{C}_{T_{\text{co}}}$, where $\mathcal{C}_t$ contains all constraints corresponding to time step t in the constraint input signal. Here, a constraint is defined by the bound c and all coefficients γ_{ih} as introduced in Equation 2.22. A constraint can, of course, also be defined for multiple time steps. In that case, it is present in multiple constraint sets.

Assume that the CESN is already initialized as described above. In the beginning, all training inputs are forwarded through the reservoir successively, according to Equation 2.1.

The resulting reservoir outputs are collected in the matrix $\mathbf{X} \in \mathbb{R}^{T_{\text{tr}} \times N_x}$ (line 1–3). Once finished, the linear least squares problem defined in Equation 2.28 is constructed, involving $\mathbf{U}$, $\mathbf{Y}^{\text{target}}$ and the reservoir outputs $\mathbf{X}$ (line 4). Since no constraints are applied yet, the linear least squares is solved using the computational cheap ridge regression and an initial $\mathbf{W}^{\text{out}}$ is obtained (line 7). After resetting the reservoir state (line 8), the constraint input time series is forwarded through the network using the currently computed $\mathbf{W}^{\text{out}}$. This delivers network outputs $\mathbf{Y}^{\text{co}} \in \mathbb{R}^{T_{\text{co}} \times N_y}$ (line 12). In order to provide the reservoir history as required for the constraint embedding, the reservoir states are collected here as well (line 13). Subsequently, all constraints defined for the current time step are evaluated with the predicted outputs. Since the constraints are constructed so that they base on the output history only, all necessary data is already available. If a constraint is violated, the collected reservoir history $\mathbf{X}^{\text{co}}$ and the input history $\mathbf{U}^{\text{co}}$ are used to construct the QP constraint as introduced in Equation 2.36, which is added to the set of constraints that shall be applied in the next iteration (line 14-17). This process is repeated, but this time the LLS problem is transformed to a QP as described in Section 2.3 and solved using a QP solver. When all constraints are satisfied, the procedure terminates and the final $\mathbf{W}^{\text{out}}$ is returned. Instead of adding constraints iteratively and solving the QP several times, adding all constraints to the QP directly and solving the problem in one pass is also possible. This can save time, especially if it is known that a large percentage of the defined constraints are violated during the training process. However, this one-shot training may require much more memory (see Appendix A3).

Initial Reservoir State Washout The training algorithm as introduced above uses the entire training time series, beginning with time step 1. But at the very beginning, the initial reservoir state is initialized artificially, which is in our case with $\mathbf{0}_{N_x}$. This initial reservoir state influences the first reservoir outputs and may cause an undesired behavior of the network in the beginning. Depending on the reservoir parameters and the input data, it can take up to several hundred time steps until this initial reservoir state has mostly vanished and the reservoir state is influenced only by the input history. For this reason, usually, the first $t_0 \in \mathbb{N}$ time steps of the time series are forwarded through the reservoir, but not used for the optimization of $\mathbf{W}^{\text{out}}$ and the embedding of constraints. This t_0 is called *washout time* or *transient time*. Concerning the training algorithm, this is observed in line 4, where the first t_0 time steps are discarded when constructing the LLS, and in lines 14-17, where the constraints are evaluated and applied only for time steps after t_0.

Some remarks to the practical implementation, including the tackling of numerical issues of the QP solver and the efficient utilization of memory capacity, are given in Appendix A3.

Data Input signals $\mathbf{U}$ and $\mathbf{U}^{\text{co}}$, target signal $\mathbf{Y}^{\text{target}}$, constraint sets $\mathcal{C}_1, \ldots, \mathcal{C}_{T_{\text{tr}}}$
Result Read-out weights $\mathbf{W}^{\text{out}}$

$\mathbf{X}^{\text{tr}} \leftarrow$ Empty array of size $T_{\text{tr}} \times N_x$
for $t \leftarrow 1, \ldots, T_{\text{tr}}$ **do**
    $\mathbf{X}_t^{\text{tr}} \leftarrow$ forwardReservoir$(\mathbf{U}_t)$
Construct LLS problem from $(\mathbf{U}, \mathbf{X}, \mathbf{Y}^{\text{target}})$
$\mathcal{C}_{\text{applied}} \leftarrow \{\,\}$
repeat
    $\mathbf{W}^{\text{out}} \leftarrow$ Solve LLS s.t. $\mathcal{C}_{\text{applied}}$
    Reset reservoir state
    $\mathbf{Y}^{\text{co}} \leftarrow$ Empty array of size $T_{\text{co}} \times N_y$
    $\mathbf{X}^{\text{co}} \leftarrow$ Empty array of size $T_{\text{co}} \times N_x$
    for $t \leftarrow 1 \ldots, T_{\text{co}}$ **do**
        $\mathbf{Y}_t^{\text{co}} \leftarrow$ Predict$(\mathbf{W}^{\text{out}}, \mathbf{U}_t^{\text{co}})$
        $\mathbf{X}_t^{\text{co}} \leftarrow$ Current reservoir state
        for all c in $\mathcal{C}_{\text{t}}$ **do**
            if $\mathbf{Y}^{\text{co}}$ violates c **then**
                $c^{\text{QP}} \leftarrow$ construct QP constraint from $(c, \mathbf{X}^{\text{co}}, \mathbf{U}^{\text{co}})$
                $\mathcal{C}_{\text{applied}} = \mathcal{C}_{\text{applied}} \cup \{c^{\text{QP}}\}$
until all constraints satsified
return $\mathbf{W}^{\text{out}}$

Algorithm 1: CESN Training Procedure

3.2 Illustrative Example: Frequency-varying Sine Wave

Consider the following learning task: Given a discrete input signal, generated from a sine wave with a fixed amplitude but a varying frequency, an ESN shall be trained to predict the frequency of the sine wave (see Figure 3.1). The input signal u is defined as

$$u(t) = \sin\left(\sum_{i=1}^{t} \delta(i)\right) \text{ for } t = 1, 2, 3, \ldots, \tag{3.3}$$

where $\delta(t)$ describes the sine wave frequency at time step t. The origin of this definition can easily be explained with an alternative interpretation of the same task: Consider a point that circulates counter-clockwise on a unit circle. The point moves with a variable velocity, which means that on each time step t, the point moves by an angle of $\delta(t)$ on the circle (see Figure 3.2a). The total angle δ_{cum}, the point has moved on the circle so far at a specific time step is the sum of all previously moved angles and the current one:

$$\delta_{\text{cum}}(t) = \sum_{i=1}^{t} \delta(i). \tag{3.4}$$

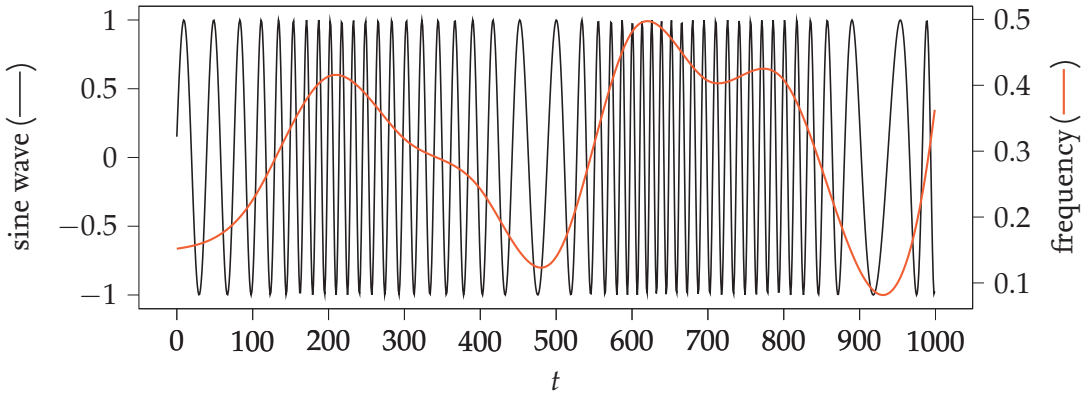

Figure 3.1: A sine wave with a variable frequency.

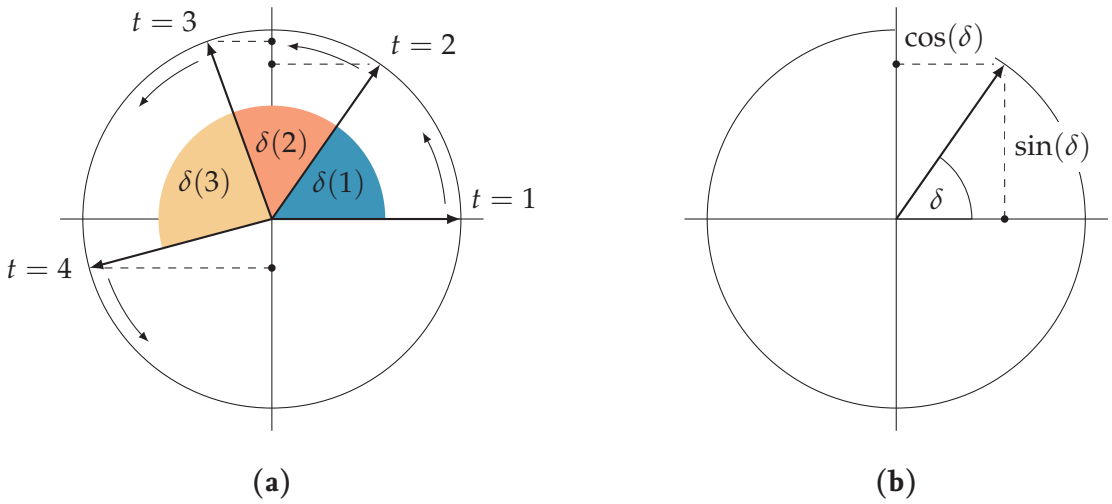

Figure 3.2: **(a)** Considering a point circulating on a unit circle with a variable velocity of $\delta(t)$, its y coordinate at time step t is defined as $\sin\left(\sum_{i=1}^{t} \delta(i)\right)$. **(b)** The point on a unit circle specified by the angle δ is defined as $x = \cos(\delta)$ and $y = \sin(\delta)$.

The coordinates of the point are defined by the sine and the cosine of this total angle $\delta_{\text{cum}}(t)$, or more precisely, by the sine and the cosine of $\delta_{\text{cum}}(t) \bmod 2\pi$ (See Figure 3.2b). Since the sine and the cosine function have a periodicity of 2π, the coordinates of the point at time step t are obtained as

$$x = \cos\left(\delta_{\text{cum}}(t)\right) \text{ and } y = \sin\left(\delta_{\text{cum}}(t)\right). \tag{3.5}$$

Using this formulation of the data, the learning task can also be reformulated: Predict the current velocity of the point on the circle, which is the current $\delta(t)$, given the y coordinate of the point over time. This task can be solved easily by considering the last few time steps. To make the task harder, a delay of T_{delay} time steps between the input and the target signal is created. Hence, the ESN shall not predict the current velocity of the point, but the velocity T_{delay} time steps ago, resulting in a target signal of

$$y^{\text{target}}(t) = \delta(t - T_{\text{delay}}). \tag{3.6}$$

In the following, two examples based on this learning task are presented. The target output signal y^{target} is generated from a cubic spline using randomly sampled points from a uniform distribution in the range $(5/1000, 5/10)$. An additional validation procedure ensures that all interpolated data points are within this range. The input signal u is computed from the given target output signal as defined in Equation 3.3. Two different types of constraints are presented and in both examples, additional prior knowledge in the form of constraints is used to improve the performance of the estimator.

Example 1: Difference Quotient Constraints

In this example, the input-output delay is set to $T_{\text{delay}} = 50$. Thus, the ESN needs to remember information about the previous 50 time steps, which can be a quite challenging task. With an available training dataset size of only 900 time steps, the task gets even more difficult. We use detailed information about the smoothness and curvature behavior of the output signal, which is known from the way, the output signal is constructed from the spline. This prior knowledge can be expressed by the first, second and third order difference quotient and is embedded in form of constraints to the CESN, which are defined as

$$\begin{aligned} |\mathrm{DQ}_1(y(t))| &\leq 1.1 \cdot \mathrm{DQ}_1^{\max} \\ |\mathrm{DQ}_2(y(t))| &\leq 1.1 \cdot \mathrm{DQ}_2^{\max} \\ |\mathrm{DQ}_3(y(t))| &\leq 1.5 \cdot \mathrm{DQ}_3^{\max}, \end{aligned} \tag{3.7}$$

where DQ_i is the i-th order difference quotient and $\mathrm{DQ}_i^{\max}$ is an absolute upper bound to the i-th order difference quotient of the output signal. The total signal has a length of 6000 time steps. The first 1000 time steps are used for training, from which the first 100 time steps serve as initial transients. The constraints are applied to the first 4000 time steps, except the transient ones. Note here, that even if we use an input of 4000 time steps during the training process, the size of the available target outputs is limited to the first

Approach	Abs. Test Error $(\cdot 10^{-3})$		Rel. Test Error	
	Best	Mean	Best	Mean
Baseline	144.6	144.6 ± 0.0	100.0%	100.0%
ESN	31.7	42.1 ± 6.8	21.9%	29.1%
ESN + postproc.	27.0	37.4 ± 4.5	18.7%	25.9%
CESN	**11.2**	$\mathbf{16.5 \pm 7.2}$	**7.7%**	**11.4%**

Table 3.1: Absolute and relative errors of example 1. The relative errors are comparisons to the baseline error.

1000. The idea here bases on the assumption that it is easy to generate new input data, but much more expensive to acquire training target output data. The time steps 4001 to 5000 are used as validation data for the hyperparameter optimization and the final test error is evaluated on the last 1000 time steps. The internal reservoir state is not reset between the different steps, so an initial washout is necessary only at the very beginning. The performance of the CESN is compared to an ESN trained on the same data but without constraints. Moreover, we apply a postprocessing step to the ESN output. The output is modified so that all constraints are satisfied and deviates as little as possible from the original ESN output (in sense of the RMSE). This postprocessing task is also solved with the use of a QP solver. The hyperparameters of both the CESN and the ESN are selected by performing a grid search on the fixed parameter sets listed in Appendix A4.1. In order to deal with the randomness of the reservoir initialization, the experiment, including the hyperparameter optimization, was repeated for 30 different network initializations.

The best and mean results of each approach are listed in Table 3.1. Here, the relative errors are computed by comparing the absolute errors to a baseline error. The baseline error is obtained by using the average of all training outputs as the predicted output at every time step (and thus a constant output signal). From the results in the table can be seen that the postprocessing step improves the ESN results clearly, but in both the best and the mean result, the CESN performs significantly better than the both ESN approaches. The estimated outputs of all three approaches are plotted in Figure 3.3 for the entire 6000 time steps. Here, it can be seen that the ESN is not able to predict good results at lower frequencies. One reason for this could be that hardly any small frequencies are given in the training data and the training signal is just not large enough to learn and generalize the target signal well. Just for comparison, increasing the number of training time steps to 4000, the ESN delivers very good results, too. This shows that the task is generally well solvable, but the small amount of available training data is the problem. In this case, the constraints help to tackle this issue and provide the missing information, usually derived from the higher amount of training data. That the outputs of the CESN are close to the target outputs even for the time steps 4000 to 6000, where no more constraints are embedded, indicates that the constraints generalize well to new data.

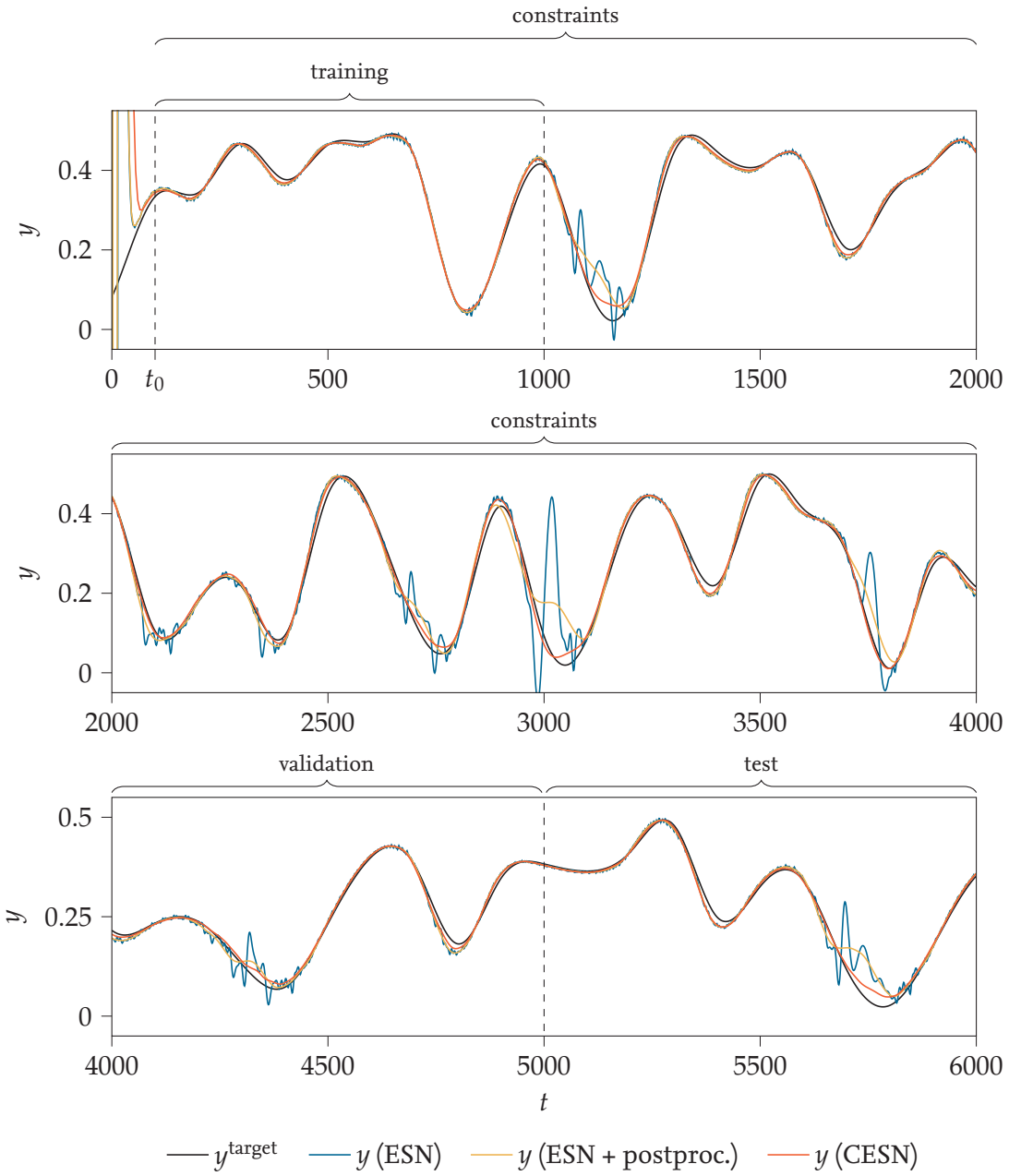

Figure 3.3: While the ESN without constraints fails to predict the sine wave frequency especially for small frequencies, the CESN delivers much better results. This holds even for the time steps, where no constraints are applied (time steps $4000 - 6000$), which indicates that the constraints are generalized well. Applying a postprocessing optimization to the standard ESN results can force the constraints and improve the results. Nevertheless they are clearly worse than the CESN results.

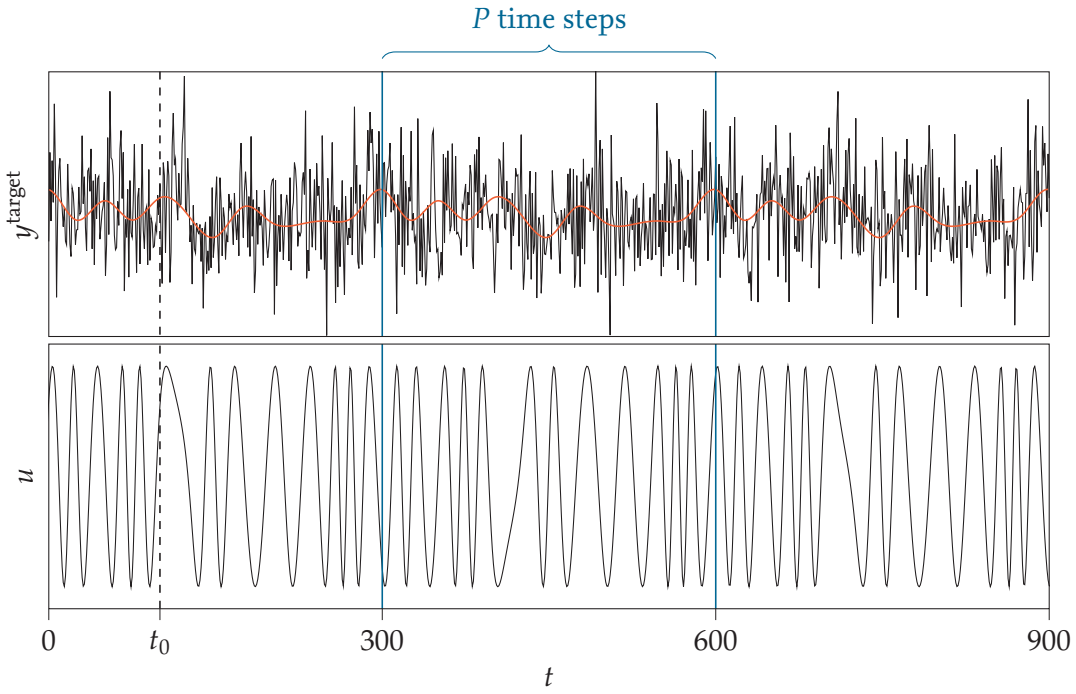

Figure 3.4: Even if the ground-truth training targets (red) repeat with a periodicity of $P = 300$, this does not hold for the input signal u. The noise, added to the ground-truth target signal removes the periodicity in the training targets y^{target}, too.

Example 2: Periodicity Constraints

While we use constraints in the first example in order to bypass the problem of too few training data, this example uses constraints to tackle the issue of very noisy training data. We select $T_{\text{delay}} = 30$. The frequency signal is generated to repeat every $P = 300$ time steps. The sine wave signal is then constructed from the frequency signal. Note that even if the frequency signal repeats with a periodicity of P, this does not hold in general for the sine wave signal (see Figure 3.4). This is caused by how the sine wave is constructed from the frequency signal as defined in Equation 3.3. Afterward, intense Gaussian noise with a mean $\mu = 0$ and a standard deviation $\sigma = 0.35$ is added to the frequency signal. The periodicity in the output signal is used as prior knowledge and constraints are defined as

$$|y(t) - y(t - P)| \leq 0.01. \tag{3.8}$$

As in the first example, one long signal of 4000 time steps is divided into disjoint parts: The first $3P = 900$ time steps are used for training, from which the first 100 time steps are discarded as initial transients. The following 1100 time steps are used to apply constraints and the last 2000 time steps are equally divided for the validation of the hyperparameters and final evaluation (testing). Again, the output signal between the times steps 901 and 2000 is not required since only the input signal is used for the constraint embedding.

This time, we compare two different CESN approaches: One CESN is trained, where the constraints are applied only to the training data. In a second one, they are applied to the training data and additionally to the following 1100 time steps. In the first approach, the time steps 901 to 1100 are not used at all. Thus, we can analyze the impact of the size of the constraint dataset on the performance. Both CESN approaches are compared to a standard ESN without constraints. Furthermore, we train a second standard ESN on preprocessed training data, which take the prior knowledge into account: The 900 available training outputs are averaged every P time steps, resulting in one mean period of P time steps, where each one is a mean of three values. This mean period is repeated three times to cover all training input time steps. Formally, we obtain preprocessed training targets

$$y_{\text{prepoc}}^{\text{target}}(t) = \tfrac{1}{3}\left(y^{\text{target}}(t) + y^{\text{target}}(t+P) + y^{\text{target}}(t+2P)\right) \tag{3.9}$$

for $t = 1, \ldots, P$ and $y_{\text{prepoc}}^{\text{target}}(t) = y_{\text{prepoc}}^{\text{target}}((t-1) \bmod P + 1)$ for $t = P+1, \ldots, 3P$. In a third standard ESN, we repeat this mean period not only for all training input time steps, but for the first 2000 input time steps and the ESN is trained on these 2000 artificially created training time steps. As in the second CESN approach, no further training outputs are required in this case since we use the mean period computed from the 900 available training outputs. Thus, this third ESN approach can be considered as the counterpart to the second CESN approach. As in the first example, the hyperparameters of all five ESNs are optimized independently on the grid search parameters listed in Appendix A4.1. The experiment is repeated for 30 different network initializations.

The best and mean results are listed in Table 3.2, where the baseline error and the relative errors are defined as above. The error is evaluated not on the noisy output data, but on the original ground-truth signal without noise. The standard ESN without preprocessing and constraints performs just slightly better than the baseline model, which most likely is caused by the intense noise on the training data. The approaches with data preprocessing yield smaller test errors, especially if the mean is artificially extended to the first 2000 time steps. Nevertheless, both CESN approaches outperform all three standard ESN approaches significantly and deliver comparably best test errors. Concerning the mean test error, the CESN with an extended constraint range performs considerably better as if the constraints are applied to the training data only. In Figure 3.5, the predicted outputs of all five approaches (using the best performing ESN each) are plotted for the training (time steps 1 to 900) and the test data (time steps 3001 to 4000). It can be seen that both CESN approaches deliver much smoother results that are close to the ground-truth signal.

Approach	Abs. Test Error $(\cdot 10^{-3})$		Rel. Test Error	
	Best	Mean	Best	Mean
Baseline	97.5	97.5 ± 0.0	100.0%	100.0%
ESN	78.8	95.3 ± 8.3	80.6%	97.7%
ESN (mean)	66.7	84.8 ± 8.5	68.4%	87.0%
ESN (mean extended)	48.7	58.9 ± 4.4	49.9%	60.4%
CESN (training data only)	**28.9**	$\mathbf{36.4 \pm 5.1}$	**29.7%**	**37.3%**
CESN	**26.1**	$\mathbf{28.7 \pm 1.4}$	**26.8%**	**29.4%**

Table 3.2: Absolute and relative errors of example 2. The relative errors are computed as a comparison to the baseline error.

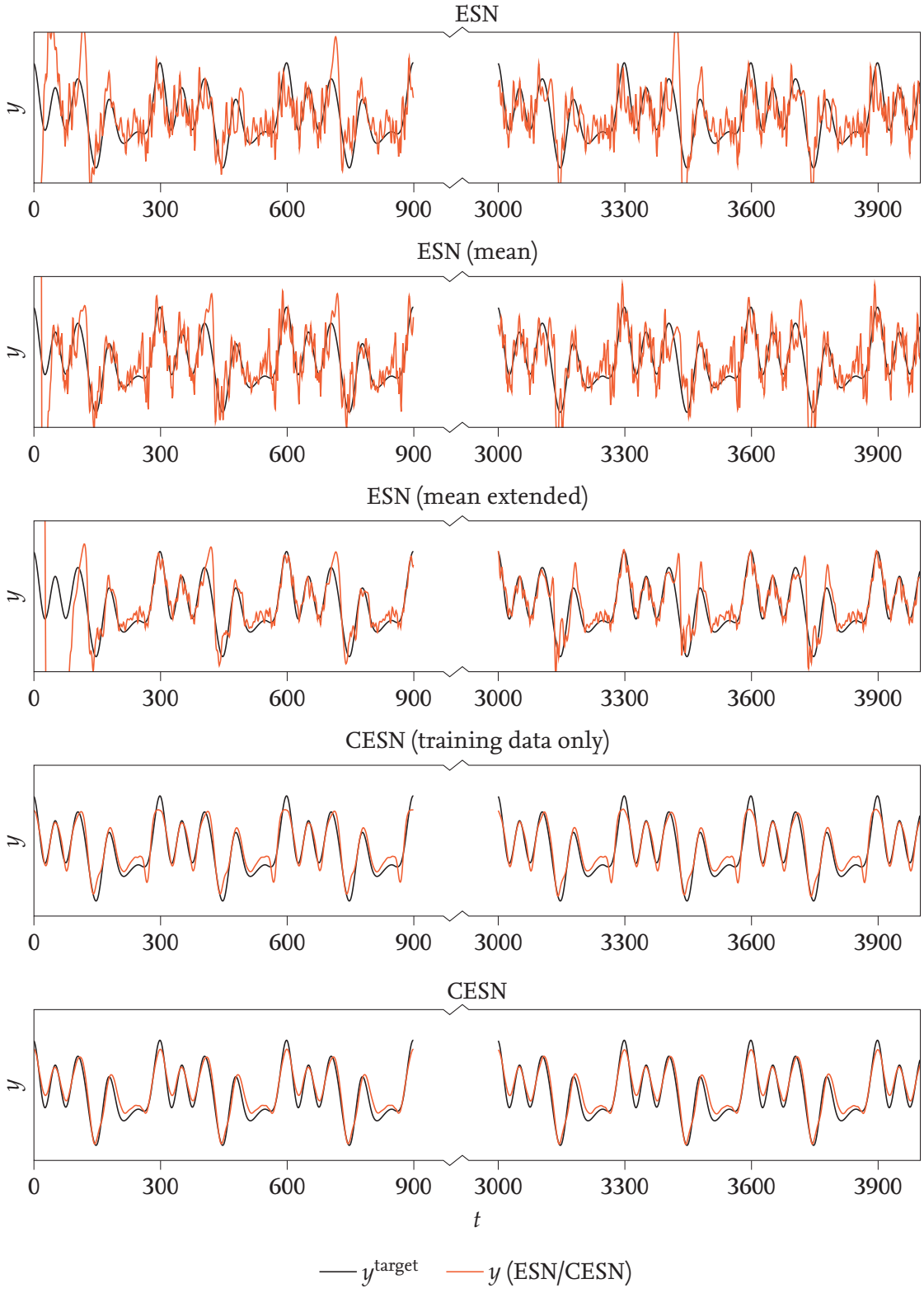

Figure 3.5: The network outputs of the different approaches for the training ($0 - 900$) and test inputs ($3000 - 4000$). While a periodicity constraint is embedded on the training data in the CESN approaches, it is not on the test data.

4 Application: Satellite Image Forecasting

This chapter considers a practical application of the introduced CESN: The forecasting of satellite image data. Given a series of satellite images, each of which covering the same area on Earth but captured at a different time, the task is to forecast the image at the next time step. The Area of Interest (AOI) considered in this chapter is the Harz highland area in northern Germany and in particular, we have a closer look at woodland areas.

4.1 Planet Labs Inc.

Founded in 2009 by ex-NASA scientists, Planet Labs Inc. (in the following called *Planet*) is a private Earth-imaging company.[1] With their goal of "imag[ing] the entire Earth every day and mak[ing] global change visible, accessible, and actionable"[1], they take a completely different approach than other companies in this sector. Instead of using a few satellites to perform specific and scheduled orders, as of today, Planet's PlanetScope satellite constellation consists of approximately 120 small satellites ($10\,\text{cm} \times 10\,\text{cm} \times 30\,\text{cm}$) in space, covering the entire land surface area of 149 million km^2 every day. The constellation provides images in four spectral bands (red, green, blue and near-infrared) with a resolution of $3.7 - 4.1\,\text{m/pixel}$ (depending on the altitude). Furthermore, Planet operates an additional satellite constellation, called SkySat, with only 21 satellites, but a higher resolution of approximately $1\,\text{m/pixel}$ (multispectral) and $0.7\,\text{m/pixel}$ (panchromatic), respectively, used for scheduled orders.[2]

4.2 Data and Data Preparation

Data Planet offers a product called Basemap. A Basemap is an aggregated and processed image of a previously specified AOI, computed from a set of satellite images of this AOI captured in a specific time range. This usually provides a high image quality, since noisy images, for example caused by clouds, snow or fog, can be filtered. Basemaps are computed on a daily, weekly, monthly or quarterly base. We consider the monthly aggregated Basemaps in the following. Basemaps are available for all months starting from January 2016. Ending with December 2020, this results in 60-months image time series with one image per month. The image resolution of the Basemaps is $4.77\,\text{m/pixel}$ at the Equator.

[1] Planet Company Introduction (accessed: March 2021) `https://www.planet.com/company/`

[2] Planet Imagery Product Specification (accessed: March 2021) `https://assets.planet.com/docs/combined-imagery-product-spec-april-2019.pdf`

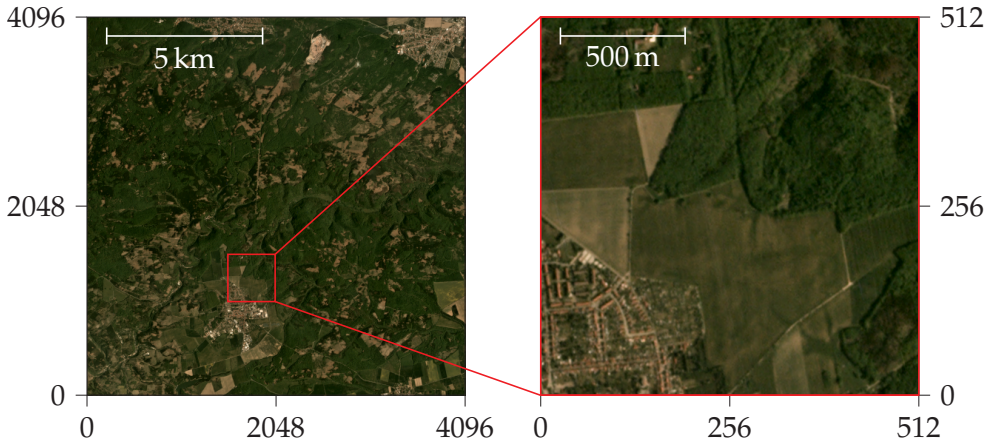

Figure 4.1: The Planet Basemap of the Harz highland area provides a high resolution of 2.93 m/pixel.

Since the altitude of a satellite is not fixed and depends on its current latitude, the image resolution deviates as well. According to Planet, the image resolution at a specific latitude is approximately obtained as

$$\mathit{resolution} \approx \mathit{resolution}_{\text{Equator}} \cdot \cos(\mathit{latitude}) = 4.77\,\text{m/pixel} \cdot \cos(\mathit{latitude}) \tag{4.1}$$

resulting in an image resolution of 2.93 m/pixel for the Harz highland area Basemap (see Figure 4.1).[3] The Basemap is divided into 4096×4096 pixels sized tiles, each of which corresponds to a surface area of approximately 12×12 km (see Figure 4.2).

Data Preparation Considering a Basemap tile for 60 time steps with three color channels with an 8-bit color depth each would result in 2.88 GiB of data just for one tile. In order to restrict the learning problem to a manageable size, each considered Basemap tile is downscaled to 128×128 pixels by averaging each 32×32 pixel square to one pixel (see Figure 4.1). In total, five Basemap tiles from different areas of the Harz with a high percentage of woodlands are selected for training and testing (see Figure 4.2). Since we are interested in making predictions about forest areas only, we create a mask indicating woodlands for each considered tile manually. This mask is used to identify the pixels on which the model is trained and tested (see Figure 4.3).

Noisy Data What exactly is defined as noise is completely task-dependent and may vary strongly between different tasks. Image artifacts caused by the satellite cameras or the image postprocessing algorithms are probably considered as noise in most applications. But, for example, if the task is to forecast the weather, the cloud coverage is most likely an important feature of the input data. On the other hand, if the task is to make predictions

[3] Planet Basemaps Product Specification (accessed: March 2021) `https://assets.planet.com/products/basemap/planet-basemaps-product-specifications.pdf`

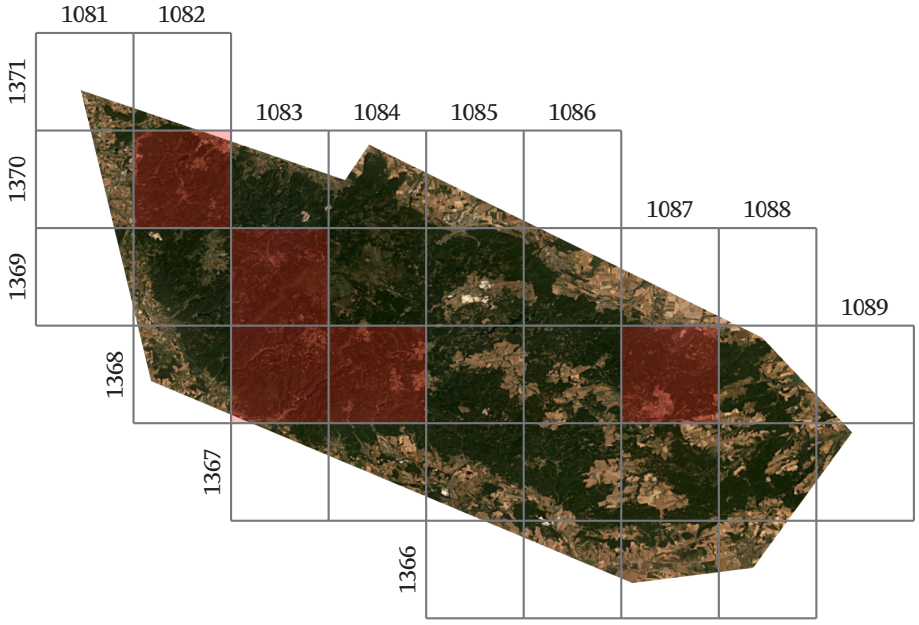

Figure 4.2: Planet Basemap of the Harz highland area (July 2019). The Basemap is divided in 4096×4096 pixel sized tiles, identified by consecutive column and row ids. We consider five different tiles with a high percentage of woodlands (highlighted in red).

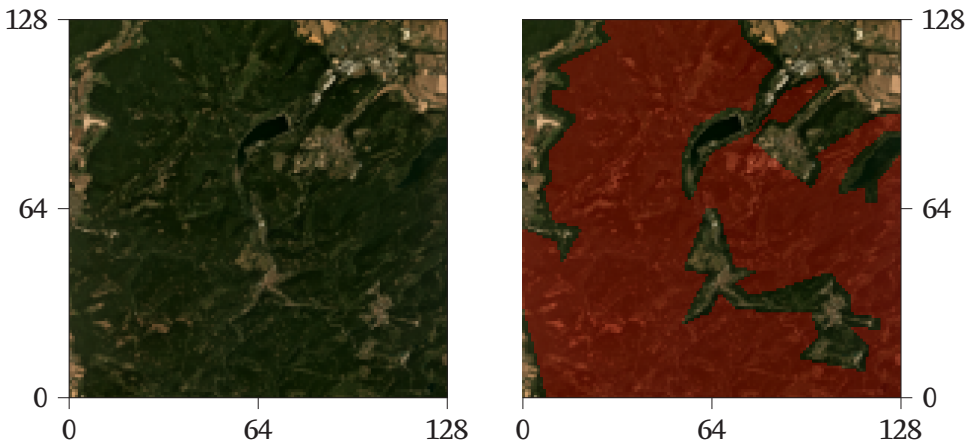

Figure 4.3: A manually created mask covers woodland areas on the scaled Basemap tiles.

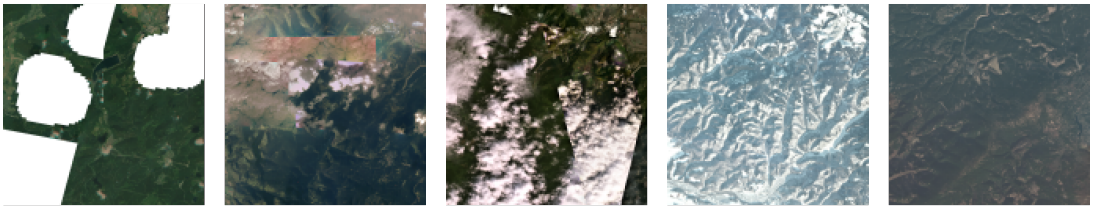

Figure 4.4: The artifacts in the leftmost image are most likely caused by an image post processing procedure. The three rightmost images contain noise in form of clouds, snow and fog. The red shade in the second picture from left could be caused by the sunlight, the camera lens or a post processing step.

about the appearance of a landscape, the clouds can be considered as noise, since they cover the underlying landscapes. In this scenario, knowledge about the presence of snow might be a helpful feature, because it influences the landscape's appearance directly. Since we are interested in the appearance of forests, we consider both clouds and snow as noise. Both cover the real dyes of the foliage and plants. Another noise factor is foggy and dull weather, which can turn a bluish tint to the image. Some examples of noisy images are shown in Figure 4.4.

4.3 Experimental Setup

Learning Task Our goal is, given a month-based time series of satellite images of woodland areas, to forecast the image of the same AOI one time step ahead, which is one month after the last time of the time series. We train and apply a model on a pixel base, which means that instead of considering one time series of images, we consider many time series of single pixels. Each pixel time series is evaluated independently from the other ones. One pixel is defined by the three color channels red, green and blue. Thus, at least in the case that we use a recurrent model, the input and output layers consist of three units each. To keep the notation concise, we denote the output units y_1, y_2 and y_3 as y_R, y_G and y_B.

Constraints Appropriate constraints shall be embedded in the model in order to get good predictions even for small training datasets. Since we do not have access to ground-truth data or specific domain knowledge, we derive some constraints from the given data. It must be noted that this constraint extraction strategy is only used to show the CESN functionality. In a real-world application, one would derive constraints from ground-truth data or with domain experts' help. Deriving constraints from the training data may amplify unwanted (and maybe unrecognized) biases. We define two different kinds of constraints, on the one hand boundaries for the color values and on the other hand constraint regarding the change of the color value between consecutive time steps. To get well-suited bounds for the single color values, we picked several images from the training data and labeled them manually as *noisy* or *not noisy*. Then we analyzed the color value ranges of the pixels of the

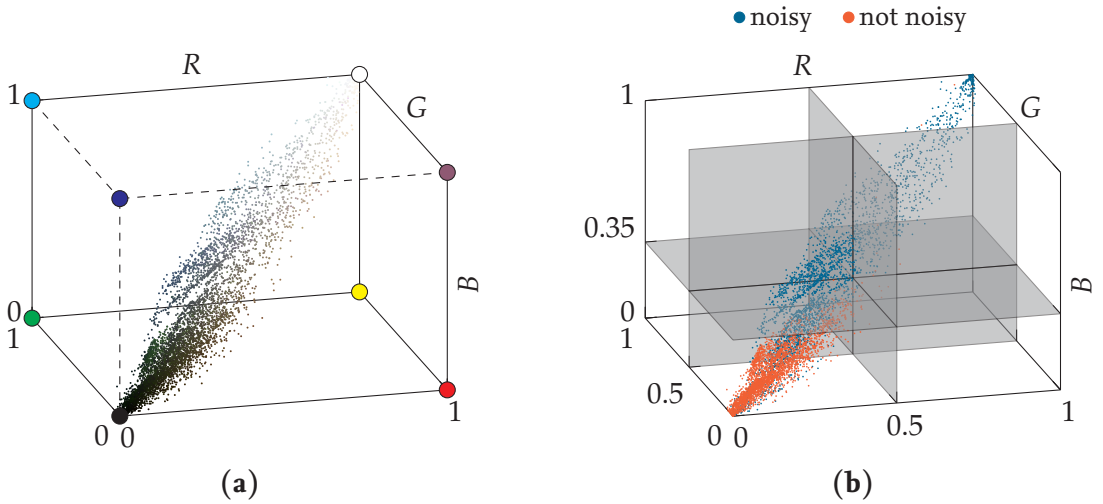

Figure 4.5: (**a**) Sample pixels of the woodland satellite images in the RGB color space. (**b**) By selecting appropriate boundary constraints, 99% of the sample pixels without noise are within these bounds, while 57% of the noisy image pixels are outside these bounds.

not noisy labeled images. Figure 4.5a visualizes the sampled pixels in the RGB colorspace. In Figure 4.5b, the two classes *noisy* and *not noisy* are colored differently. By selecting the boundaries

$$\begin{aligned} y_R(t) &\leq 0.50 \\ y_G(t) &\leq 0.50 \\ y_B(t) &\leq 0.35, \end{aligned} \tag{4.2}$$

as indicated in the plot, 99% of the *not noisy* image pixels are within these bounds. On the other hand, 57% of the pixels of the *noisy* images are outside that bounding box. Note here that all images in the class *not noisy* do not contain any visible noise, but images from the class *noisy* can be partially noise-free. The second kind of constraint bases on the assumption that the color values of the forest pixels, mainly influenced by the color of the foliage, change throughout the year, but only by a small amount between two consecutive months. Here, the boundaries are derived from the training data directly, too, resulting in the following difference constraints:

$$\begin{aligned} y_R(t) - y_R(t-1) &\leq 0.05 \\ y_G(t) - y_G(t-1) &\leq 0.05 \\ y_B(t) - y_B(t-1) &\leq 0.05. \end{aligned} \tag{4.3}$$

Experiment Description We implement an experiment with two independent variables: In one dimension, we compare the impact of the constraints in a CESN to an ESN without constraints. The second dimension is the size of the training dataset. Here, we compare training sample dataset sizes of 25, 50, 75, 100, 150, 200, 500 and 1000. As already mentioned above, we use five different Basemap tiles (and thus five different woodland areas in the

Harz) for training and testing in total. Since we want to train especially on small training datasets, but the tiles have a large variance among themselves, we use a special kind of cross-validation to obtain statistically more representative results. Instead of leaving out a small subset of the training data for validation, as it is usually done in cross-validation, we do it the other way around: We pick our training samples from one of the five tiles and evaluate the trained model on the other four tiles. For comparison, the model is also tested on the training tile (using all pixels except the ones used for training). This experiment is repeated so that each of the five tiles serves once as the training tile. In this way, we conduct a total of 80 experiment runs (five different training tiles with 16 parameter combinations each).

The training pixels are sampled uniform-randomly from the training tile. If a pixel is selected, the whole 60-months time series of that pixel is used for training. The first 24 months of each pixel are discarded as transients to washout the initial reservoir state. The months 25 to 59 are used as training inputs and the months 26 to 60 are the corresponding target outputs. Here, the ESN is always fed by the inputs in that order (month 1 to month 59), starting with an initial reservoir state of $\mathbf{0}_{N_x}$. While the training dataset size is variable, the number of pixel time series used to embed the constraints is fixed to 1000 in each run, as was also done in Section 3.2. As constraint data, all training samples are used and additional constraint input pixel time series are also sampled from the training tile.

The hyperparameters are determined in advance with a grid search on the parameters *spectral radius*, *leaking rate* and *input scaling*. The other parameters seem to have only a minor impact on the performance of the ESN here. The training dataset size is fixed to 1000. The grid search is repeated 15 times with different training sample datasets and reservoir initializations. It was executed for the ESN only and the resulting hyperparameters are used for both, the ESN and the CESN. We assume that good hyperparameters for the ESN perform similarly well for the CESN. The entire grid search parameters are listed in Appendix A4.2. The final ESN is initialized with the fixed hyperparameters *regularization*: 10^{-8}, *reservoir size*: 750 and *sparsity*: 0.0, together with the grid search selected hyperparameters *spectral radius*: 0.99, *leaking rate*: 1.0 and *input scaling*: 0.1.

4.4 Results and Evaluation

For each of the five tiles, we obtain one training tile error and four test tile errors (in the sense of RMSE), corresponding to the four test tiles. Note that here, unlike usual, by training tile error we do not refer to the prediction error on the training dataset, but on the pixels of the training tile that were not used for training. In this way, we can analyze how well the model generalizes to the tile it was trained on compared to how well it generalizes to other tiles. For each training dataset size we obtain a mean training tile error by averaging over the five single training tile errors and a mean test tile error by averaging over all 20 resulting test tile errors. The mean errors are listed in Table 4.1 and visualized in Figure 4.6. In Figure 4.6a can be seen that, expectably, the generalization

Training Data Size	RMSE Training Tile			RMSE Test Tiles		
	ESN ($\cdot 10^{-2}$)	CSN ($\cdot 10^{-2}$)	CESN/ESN	ESN ($\cdot 10^{-2}$)	CESN ($\cdot 10^{-2}$)	CESN/ESN
25	14.7	7.9	54.7%	21.3	9.3	46.7%
50	7.9	7.5	95.4%	12.8	7.9	70.0%
75	6.7	7.5	110.1%	10.9	8.7	80.2%
100	6.5	7.4	112.7%	10.6	8.7	81.8%
150	6.1	7.4	120.2%	10.1	8.7	86.8%
200	5.9	7.4	123.5%	9.7	8.6	89.1%
500	5.7	7.3	128.0%	9.2	8.6	93.4%
1000	5.6	7.3	130.0%	9.0	8.5	94.8%

Table 4.1: The absolute mean errors of the ESN and the CESN approach on the training and the test tiles and the ration between both in percent.

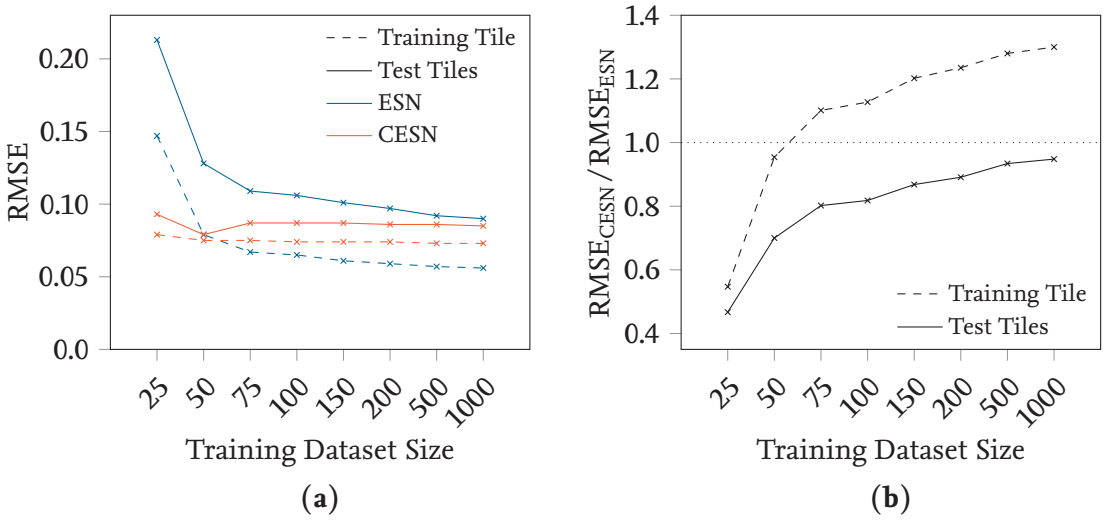

Figure 4.6: (**a**) The error on the training tile is in general smaller than the one on the test tiles. While the performance of the ESN increases with an increasing training dataset size, the CESN delivers similarly good results across all training dataset sizes. (**b**) The CESN performs worse than the ESN on the training tiles, but outperforms it on the test tiles.

basically works better on the training tile than on the test tiles. The ESN performance increases steadily with an increasing training dataset size. In contrast, the CESN delivers similarly good results across all training dataset sizes. But while the CESN performs, in general, worse than the ESN on the training tile, it outperforms it on the test tiles, especially for the smaller training dataset sizes. The larger the training dataset becomes, the more both results approach each other, but the CESN still performs slightly better even for a maximum training dataset size of 1000 (see Figure 4.6b).

In the following, a detailed analysis of the results is given for a training dataset size of 1000 and exemplary for a specific training tile (1082-1370) and a specific test tile (1084-1368). In Figure 4.7, the predicted outputs of all three color channels of 200 randomly selected pixels of the test tile are plotted over the entire 60 months. As a reference, the corresponding target image pixels are also visualized. Since the first 24 months are not used for training, but only for an initial washout, the prediction results are very bad in this range. Beginning with the 25-th month, the results are much better for both approaches. One can see that the CESN output signals are much smoother than the ones generated by the ESN and mostly within the bounds defined by the constraints. This indicates that the constraints, in general, are satisfied even for new input data. The labeled time steps in the plot correspond to the images in Figure 4.8. In Figure 4.8 the full predicted output images at these time steps are shown side by side with the associated target images and a pixel-wise RMSE computed as

$$\sqrt{\frac{1}{3}\left(\left(y_R^{\text{pred}} - y_R^{\text{target}}\right)^2 + \left(y_G^{\text{pred}} - y_G^{\text{target}}\right)^2 + \left(y_B^{\text{pred}} - y_B^{\text{target}}\right)^2\right)}, \tag{4.4}$$

where y^{target} is the target pixel at the specific time step and y^{pred} the predicted output of the ESN or CESN, respectively, at the specific time step. In Figure 4.8c and Figure 4.8d, corresponding to the months April and November 2019, it can be seen that both approaches can yield good forecasting results. The noise caused by clouds is obviously not forecasted, which yields larger errors at these pixels. According to the satellite image, the weather in February 2018 (Figure 4.8a) was very snowy. This noise in the form of snow exists also in the ESN predicted image. The CESN forecasts an image without snow. In contrast, there is almost no snow on the image from March 2018 (Figure 4.8b), but as in January, the ESN forecasts a snowy landscape and the CESN a not snowy one. Thus, the CESN error is much smaller in this case. A similar behavior can be observed in October 2020 (Figure 4.8e). The ESN predicts a much brighter image than it actually is, where the CESN predicted one is just slightly brighter than the target. The reason for such predictions is that the kind of noise we are dealing with in the training data usually extends across the entire training tile (see Figure 4.9). Thus, it is learned by the ESN and then generalized to the test tiles, on which the same noise does not necessarily occur. Here, the CESN ensures by its constraints that the noise in the training data is not learned and thus not generalized to new data.

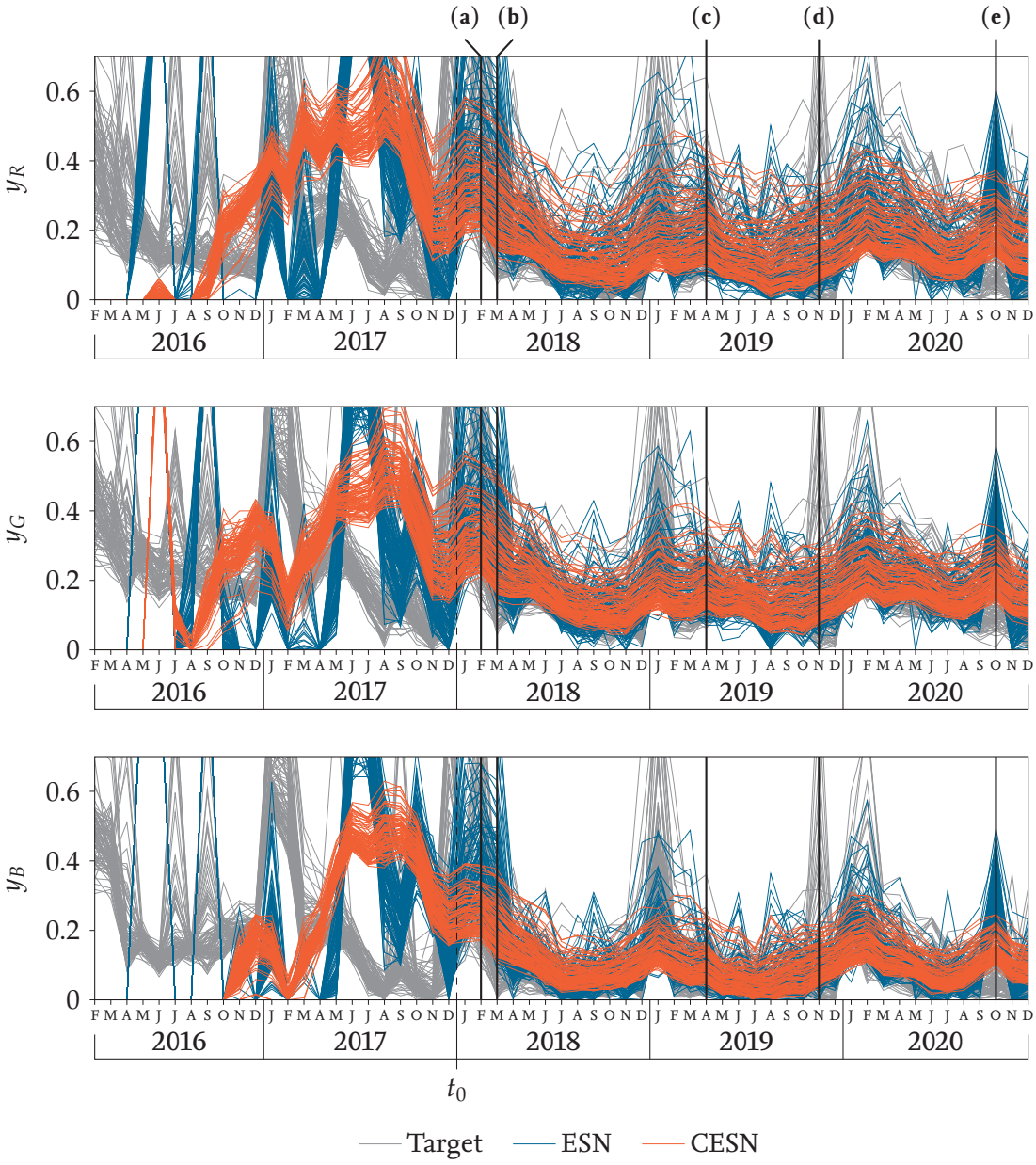

Figure 4.7: The red, green and blue color channels of 200 pixels sampled from the test tile 1084-1368 and plotted over the entire 60 months. The predictions of the ESN and the CESN are compared to the target pixels. The labeled time steps correspond to Figure 4.8.

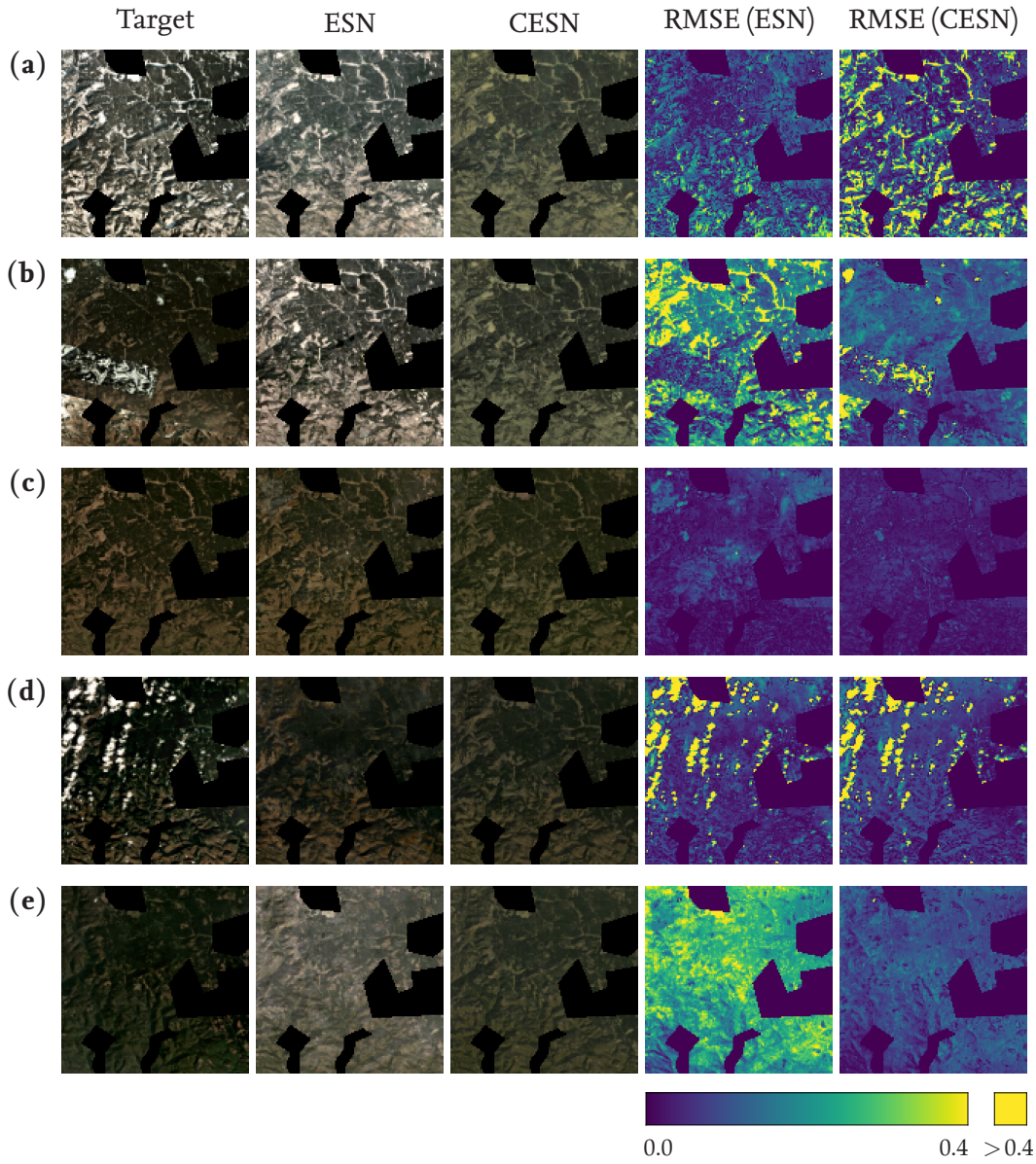

Figure 4.8: Forecasts of the ESN and the CESN of the test tile 1084-1368 for the months **(a)** February and **(b)** March 2018, **(c)** April and **(d)** November 2019 and **(e)** October 2020. The networks are trained on tile 1082-1370. The errors are computed between the target image and the predicted ones.

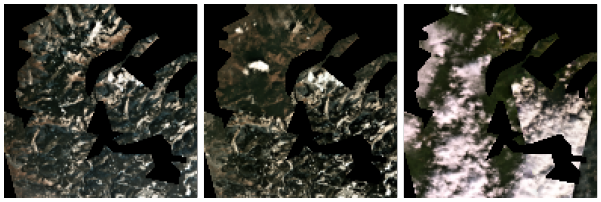

Figure 4.9: The target images of the training tile 1082-1370 for the months February and March 2018 and October 2020 are all noisy in terms of snow or clouds.

5 Extension: Output Feedback

The reservoir update equation we considered so far in Equation 2.1 differs slightly from the one Jeager introduced when initially proposing the ESN approach in 2001 [17]. In the original ESN definition, a recurrence exists within the reservoir, but also between the output units and the reservoir neurons, resulting in a feeding of the network output back to the reservoir (see Figure 5.1a). This kind of output feedback may be required, if the output depends not only on the input history, but also on the output history. One common task tackled with this definition is pattern generation: The network is trained with an initial target output signal and the task is to predict the output for several upcoming time steps recursively, where each prediction depends on the previously predicted one. There may even be no input at all and the reservoir update is influenced only by the fed back output signal. An often used benchmark in the context of pattern generation with ESNs is the prediction of the Mackey-Glass time series [10, 17, 23, 28, 35].

Adding output feedback to the ESN yields an extended reservoir equation of

$$\mathbf{x}(t) = (1-\alpha)\,\mathbf{x}(t-1) + \alpha\,\sigma\left(\mathbf{W}^{\text{in}}\begin{bmatrix}1\\ \mathbf{u}(t)\end{bmatrix} + \mathbf{W}\mathbf{x}(t-1) + \underbrace{\mathbf{W}^{\text{fb}}\mathbf{y}(t-1)}_{\text{output feedback}}\right), \tag{5.1}$$

where the reservoir neurons are not only fed by the inputs $\mathbf{u}(t)$ and the previous reservoir state $\mathbf{x}(t-1)$, but also by the previous network output $\mathbf{y}(t-1)$, weighted with feedback weights $\mathbf{W}^{\text{fb}} \in R^{N_x \times N_y}$. As with $\mathbf{W}$ and $\mathbf{W}^{\text{in}}$, the feedback weights $\mathbf{W}^{\text{fb}}$ are assigned randomly from a selected distribution. Due to the modified network architecture, the ESN loses its fundamental characteristic: The read-out layer cannot be decoupled from the rest of the network anymore, even for the unconstrained ESN, as visualized in Figure 5.2.

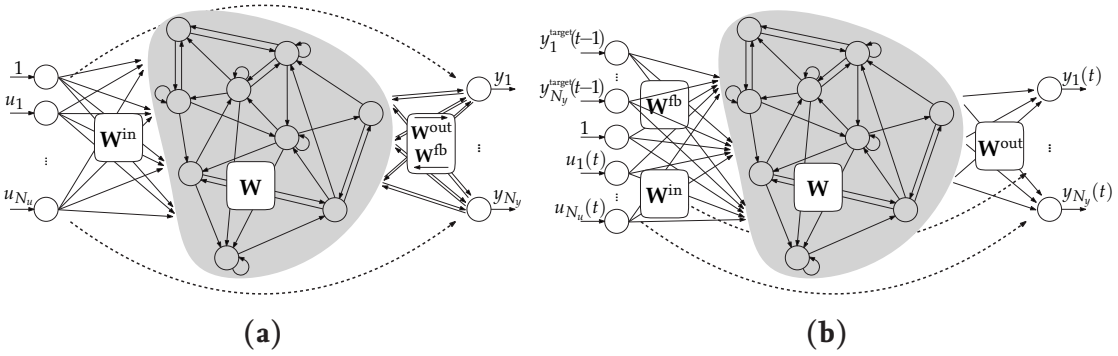

Figure 5.1: (**a**) The ESN with output feedback is recurrent not only inside the reservoir but also between the reservoir and the read-out layer. (**b**) During the training process, the feedback connections are replaced by the training target signal as a further input.

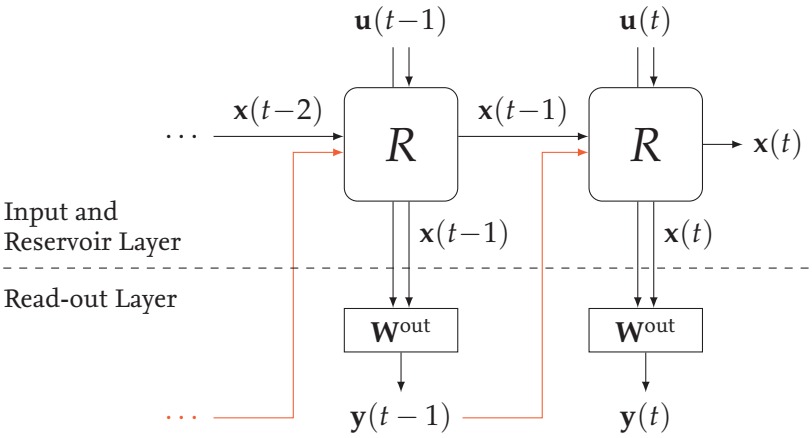

Figure 5.2: The output feedback connections create a recurrence between the reservoir layer and the output layer. Thus, the read-out layer of the ESN can no longer be fully decoupled from the rest of the network. The reservoir state $\mathbf{x}(t)$ depends on the network output $\mathbf{y}(t-1)$ and thus cannot be computed without knowing the output weights $\mathbf{W}^{\text{out}}$.

Hence, this property, used to avoid computational expensive, iterative learning methods, can no longer be exploited in the training process. A solution to this issue, called teacher forcing, is presented in [17]. The concept of teacher forcing is based on the assumption that the trained network approximates the training target output signal $\mathbf{y}^{\text{target}}$ quite well. With this in mind, we can remove the feedback connections during the training process and "bootstrap" the training process by feeding the reservoir with the target output signal as a further input instead. Thus, we keep the feedforward property of the read-out layer during the training and train the network as without output feedback (see Figure 5.1b). In case $\mathbf{y} \approx \mathbf{y}^{\text{target}}$ holds, the trained network will work as expected after reinserting the feedback connections.

But while this approach can work well for the ESN training, there are some challenges when it comes to the embedding of constraints, if output feedback is involved. First of all, one key concept of the introduced constraint framework is that the reservoir history can be computed in advance and accessed explicitly to construct the constraints out of it. This precomputation is no longer possible since the reservoir history depends highly on the output weights $\mathbf{W}^{\text{out}}$, as indicated Figure 5.2. Using teacher forcing to precompute the reservoir history does not work here for two reasons. On the one hand, the constraints are not necessarily defined on the training data. Suppose constraints are defined on a separate input signal, as it is done in the previous examples. In that case, no target output signal $\mathbf{y}^{\text{target}}$, which could be used as a teacher signal, is available for this data. On the other hand, even if the constraints are defined on the training data signal only, there is a big pitfall preventing the use of teacher forcing for the constraint embedding: If the target output signal $\mathbf{y}^{\text{target}}$ does not satisfy the constraints by default, the embedded constraints would influence the network output $\mathbf{y}$ in the attempt to satisfy them. As a result, $\mathbf{y}^{\text{target}} \approx \mathbf{y}$ does no longer hold, which is a prerequisite for teacher forcing to work. Hence, the constraints

would be constructed from a reservoir history which is based on wrong network outputs $\mathbf{y}$, which finally leads to an embedding of constraints that do not refer to the actual network output.

In the following, we present an approach tackling these issues and embed constraints to an ESN with output feedback. The idea is to use teacher forcing for the actual training of the ESN, but using an iterative approach to construct the constraints correctly in several iterations. On the first iteration, the network is trained without any constraints using the teacher forcing approach. Afterward, the trained ESN is run up to the last time step at which a constraint is defined. The resulting reservoir states $\mathbf{x}$ and outputs $\mathbf{y}$ are used to construct the constraints. All violated constraints are added and the training of the ESN is repeated. Thus, in each iteration, new constraints are added to the network, which all refer to the $\mathbf{W}^{\text{out}}$ computed in the previous iteration. Assuming that $\mathbf{W}^{\text{out}}$ converges to a valid solution in which all constraints are satisfied, this process is repeated until this fixed point is reached (or a maximum number of iterations is reached). Note here that we use teacher forcing with the initial target signal to decouple the read-out layer from the reservoir in the objective function, but use the previously computed output weights to refine the constraints. Furthermore, we only add new constraints and keep the previous ones. Thus, a constraint, formally defined at one time step, may results in multiple applied constraints in the end, since for one specified constraint, a new one can be added in each iteration. The more iterations are required to reach the fixed point, the more constraints are applied, referring to previously computed output weights $\mathbf{W}^{\text{out}}$. As a result, this can lead to numerical issues in the QP solver or even to unsatisfiable solutions, if the reservoir capacity is not large enough to represent the large number of (often redundant or obsolete) constraints. As a solution, we discard a specific amount of old constraints in each iteration. In general, the old constraints are still required, since otherwise, the solver would just fallback to its initial (and wrong) solution. But we assume that many constraints are redundant and by removing part of the old constraints randomly, the above-mentioned issue is solved.

The key concept of the presented approach can be summarized as a kind of "double bootstrapping": On the one hand, the initial teacher signal "bootstraps" the objective function minimization, while, on the other hand, the satisfaction of the constraints is "bootstrapped" by itself iteratively. However, there is no guarantee that this approach works in general, but a working example is presented in the following.

Example: Pattern Generation Given an initial training target signal y^{target} for T_{train} time steps, an ESN should be trained, which continues the signal on its own. After running for some time steps, the signal should be shifted five time steps ahead. This condition is forced using constraints. The target signal is generated as

$$y^{\text{target}}(t) = \frac{t}{100} \sin\left(\frac{t}{10}\right). \tag{5.2}$$

We initialize an ESN with 200 reservoir neurons with the hyperparameters *input scaling*: 0.5, *leaking rate*: 0.3, *spectral radius*: 0.7, *sparsity*: 0.0 and *regularization*: 10^{-8}. The ESN is trained

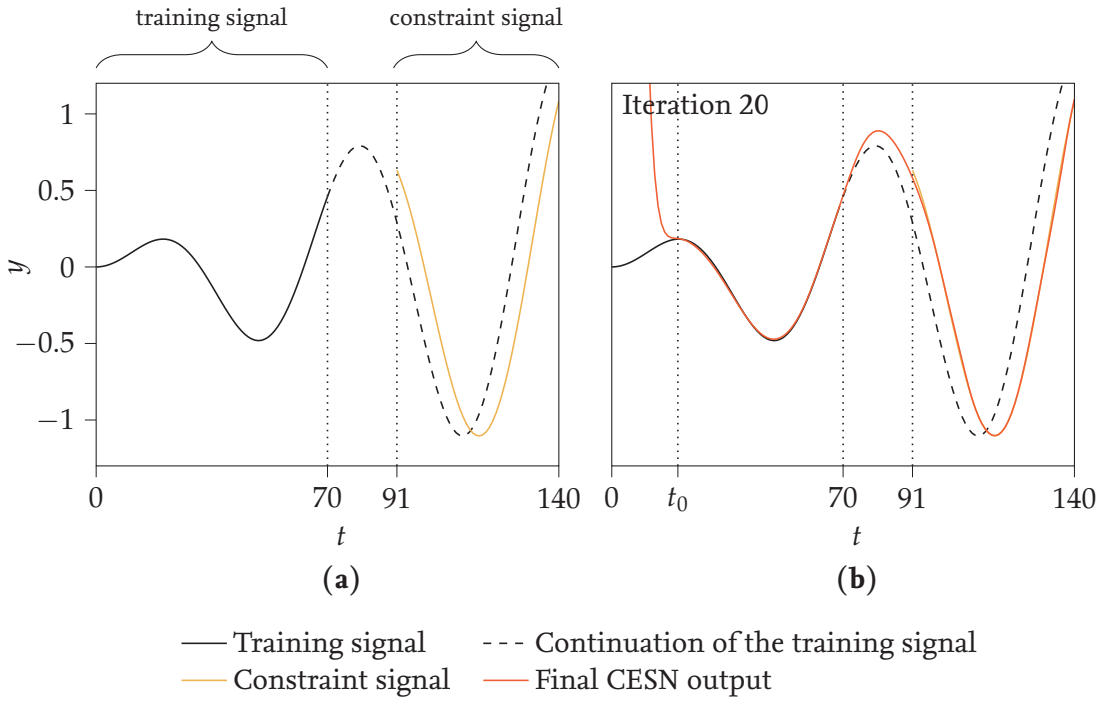

Figure 5.3: **(a)** The ESN is trained on a training signal $y^{\text{target}}(t)$ and constraints shall shift the learned signal to $y^{\text{target}}(t+5)$. **(b)** The trained ESN approximates the training signal well, while satisfying all constraints.

on the first $T_{\text{train}} = 70$ time steps with an initial washout time of 10 time steps. Afterward, the signal runs free for 20 time steps and then shall be shifted by five time steps ahead for the next 50 time steps (see Figure 5.3a). We allow an absolute constraint tolerance of $\varepsilon = 0.05$. Expressed formally, we obtain 100 constraints, of which at most the half can be violated at once:

$$\begin{aligned} y(t) &\leq y^{\text{target}}(t+5) + \varepsilon \quad \forall 91 \leq t \leq 140 \\ y(t) &\geq y^{\text{target}}(t+5) - \varepsilon \quad \forall 91 \leq t \leq 140. \end{aligned} \tag{5.3}$$

The plots in Figure 5.5 show the change of the learned signal over several iterations. Initially, the ESN is trained without any constraints. Thus, almost all constraints are violated and the learned signal is a good approximation of the continuation of the training signal. After the first few iterations, the learned signal looks quite unstable with many oscillations, but the more iterations are executed, the closer the learned signal approaches the target. After 14 iterations, most of the constraints are satisfied, but it takes another 6 iterations until the training process terminates after 20 iterations with the satisfaction of all constraints. Figure 5.4 shows the change of the number of violated constraints and the number of applied constraints during the training process. While the number of violated constraints is limited to 50, the number of applied constraints begins with 0 and increases steadily up to 235 constraints in the last iteration.

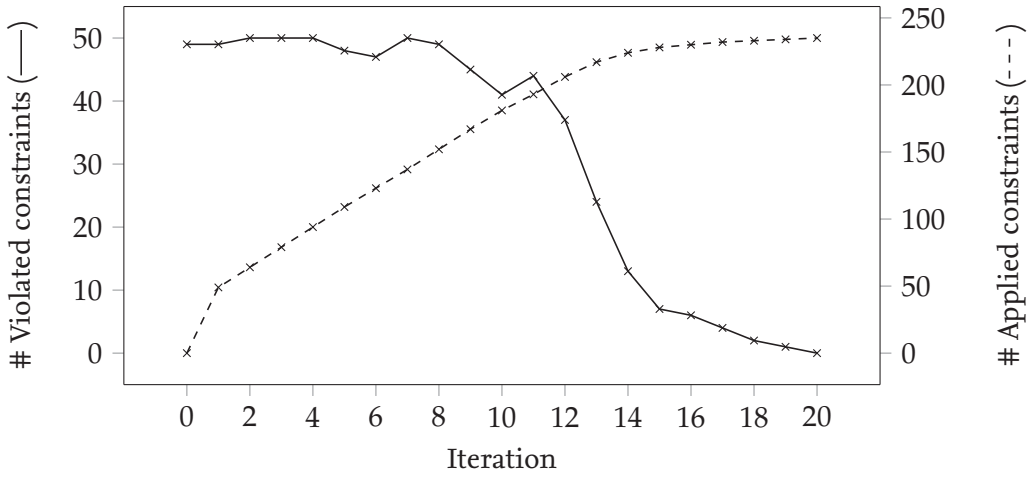

Figure 5.4: In each iteration, new constraints are added to the training problem. The number of violated constraints decreases during the training process until all constraints are satisfied.

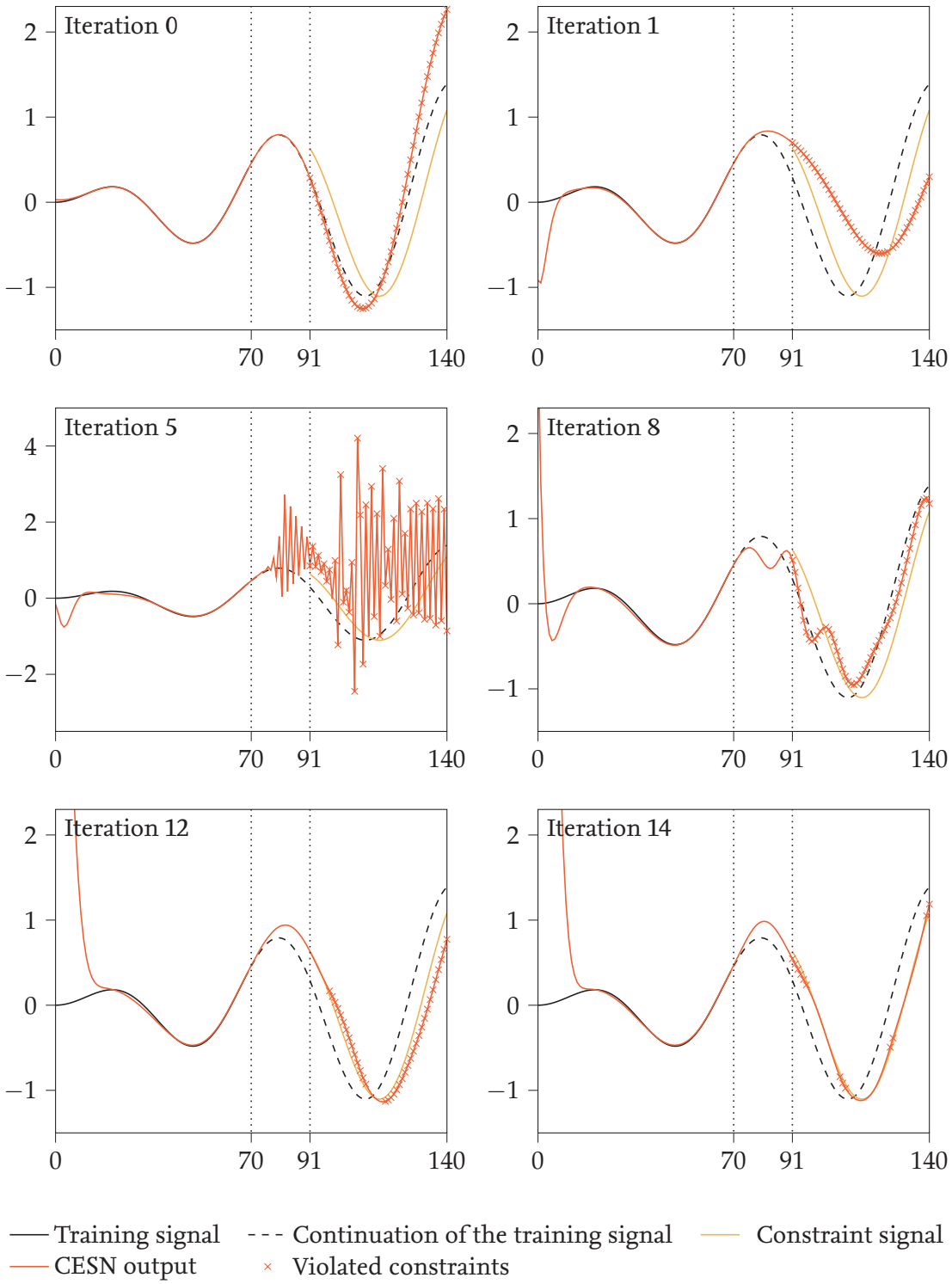

Figure 5.5: Initially, the CESN output signal matches the natural continuation of the training signal. By iteratively adding constraints, the signal approaches the target signal step by step. The satisfaction of all constraints is reached after 20 iterations (see Figure 5.3b for the final result).

6 Conclusion and Discussion

Within this thesis, we introduced a framework based on [34] that allows embedding constraints into ESNs, which are RNNs from the field of reservoir computing. The specific structure of the ESN is very similar to the feedforward neural network used in [34]. Thus, the ESN is predestined to adapt the constraint-embedding framework. While an adaption to other reservoir computing approaches may work due to their similar network characteristics, other RNN structures in general, like LSTMs, do not seem to meet the requirements to adapt this approach.

We discussed different types of constraints. The constraints introduced in [34] depend entirely on the current network input, which does not make much sense when dealing with dynamic systems. Hence, we introduced the discrete time-dependent constraints, which do not influence the sensitivity to change of the network output with respect to changes in the input, but the sensitivity with respect to a change in time. Here, we elaborated two different approaches: First, we considered the discrete ESN equation as a continuous one and approximated the discrete-time constraints. In the second approach, we defined the constraints explicitly over the discrete network output. While the first approach can lead to an undesired behavior, at least in some cases, we showed in different examples that the second approach works quite well. In an illustrative example, we used constraints for smoothing by regulating the discrete derivatives of the ESN output on the one hand and forced the output signal to be periodic on the other hand. In both examples, the constraints led to a kind of smoothing and regularization of the output. This behavior is also described for the feedforward case in [33]. As a practical application, we applied the introduced constraint framework to the forecasting of satellite images. We considered monthly satellite images from woodland areas and used constraints to deal with noise like clouds and snow. The constraints were defined as limits to the color values of the single pixels and upper bounds to the change of a color between two consecutive months. Here, we derived the constraints only from the available data without having any further domain knowledge.

Even if we presented some well working examples in this thesis, the use cases for this framework are restricted to very particular scenarios, which we discuss in the following. Several challenges needs to be tackled. First of all, task-specific domain knowledge is required in order to be able to define general constraints about the data at all. Suppose we just derive the constraints from the available training data, as done exemplary in the satellite image example. In that case, we may define constraints amplifying a bias that is in the data and does not represent the real, expected target. In a real-world application, the constraints should always be discussed and elaborated with domain experts in order to validate the correctness of the constraints. Once representative constraints are worked out, these constraints need to be formalized to fit into the framework. Even if the framework can

express many different properties like monotonicity, smoothness, periodicity, curvature, or upper and lower bounds, it is often not trivial to transform informal constraints into formal, mathematical expressions. The third challenge consists of identifying which constraints can add value to the learning task in the end. Constraints that are already satisfied by the training data would not necessarily improve the model, especially if the training dataset is large enough. On the other hand, if the constraints contradict the training data strongly, it is questionable whether the model learns from the training data at all or only enforces the constraints without learning anything. The training data and the constraints should complement each other. As shown in the examples, constraints can be very useful if the amount of available training data is very small or the dataset is very noisy. In this case, the knowledge expressed with constraints is already decoded implicitly inside the training data, but it is hard to learn due to missing patterns in the data. Furthermore, it can be beneficial to apply constraints not only to the training data, but also on further input signals, as also shown in this thesis. This approach is useful especially if the generation of training target outputs is expensive. But this approach is only applicable for tasks where it is easy to generate new input data. One example could be the prediction of observations from satellite images. Thanks to companies like Planet or the European Union's Copernicus Programme, satellite images (the input data) of most areas of the Earth are available on a nearly daily basis. But ground-truth measurements about the observations (the training target data) often can only be measured manually. Here, constraints can help to fill the gap between ground-truth measurements if the corresponding input data is available. Finally, the last challenge is the evaluation of the learned model, as already explained in Chapter 1. Depending on the task and the training data, it can be hard to find suitable evaluation metrics. This holds especially when comparing the CESN approach to unconstrained models, since in this case basically different training goals are tackled. In summary, four major challenges need to be taken into account for the successful implementation of a CESN:

1. Constraint definition using domain knowledge
2. Constraint formalization
3. Identification of constraints that add value to the model
4. Evaluation of the trained model

Future Work

One aspect of constraint learning that is not covered by this thesis is the verification of the learned constraints. The approach we implemented so far is limited to the learning of constraints from samples. While the presented examples showed that the generalization of constraints seems to work well, there is no verification ensuring that the constraints hold for new data in general. The feedforward CELM introduced in [34] takes this one step further: By using the second order Taylor approximation and a clever sampling strategy, it

can be ensured that the embedded constraints will also hold for new data from a predefined input domain. This can be required if a reliable performance has to be guaranteed. One example could be a neural network that controls the movements of a hardware system. Limitations of the hardware usually bound the range of motion of such a system. Then, the constraints, describing these limitations, need to be satisfied to ensure a smooth process.

Another aspect of future work is the deeper analysis of CESNs with output feedback. Recursive prediction like pattern generation is a major field in the RNN research, for example when dealing with speech recognition. Recursive prediction with ESNs is usually achieved with output feedback ESNs. We presented a proof-of-concept that the constraint learning approach can also work for this kind of ESNs. However, many questions still remain unanswered, for example, how well this approach works in general, where the limitations are and whether it can be guaranteed that the procedure converges to a fixed point. Another interesting facet regarding this topic is the stability of a recursive, dynamic system. Here, the question arises, whether constraints can help to improve the stability of such a system.

Bibliography

[1] A. F. Atiya and A. G. Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *IEEE Transactions on Neural Networks*, 11(3):697–709, 2000.

[2] F. M. Bianchi, L. Livi, C. Alippi, and R. Jenssen. Multiplex visibility graphs to investigate recurrent neural network dynamics. *Scientific Reports*, 7(1):1–13, 2017.

[3] M. Buehner and P. Young. A tighter bound for the echo state property. *IEEE Transactions on Neural Networks*, 17(3):820–824, 2006.

[4] B. Chen, Z. Hao, X. Cai, R. Cai, W. Wen, J. Zhu, and G. Xie. Embedding logic rules into recurrent neural networks. *IEEE Access*, 7:14938–14946, 2019.

[5] N. Chouikhi, B. Ammar, N. Rokbani, and A. M. Alimi. PSO-based analysis of echo state network parameters for time series forecasting. *Applied Soft Computing*, 55:211–225, 2017.

[6] H. Cui, X. Liu, and L. Li. The architecture of dynamic reservoir in the echo state network. *Chaos*, 22(3):033127, 2012.

[7] H. Daniels and M. Velikova. Monotone and partially monotone neural networks. *IEEE Transactions on Neural Networks*, 21(6):906–917, 2010.

[8] C. Gallicchio and A. Micheli. Deep echo state network (deepesn): A brief survey. *CoRR*, abs/1712.04323, 2017.

[9] C. Gallicchio and A. Micheli. Echo state property of deep reservoir computing networks. *Cognitive Computing*, 9(3):337–350, 2017.

[10] C. Gallicchio and A. Micheli. Why layering in recurrent neural networks? a deepesn survey. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018.

[11] C. Gallicchio, A. Micheli, and L. Pedrelli. Deep reservoir computing: A critical experimental analysis. *Neurocomputing*, 268:87–99, 2017.

[12] M. Han and D. Mu. Multi-reservoir echo state network with sparse bayesian learning. In *Advances in Neural Networks - ISNN 2010*, pages 450–456, 2010.

[13] E. Hartman. Training Feedforward Neural Networks with Gain Constraints. *Neural Computation*, 12(4):811–829, 04 2000.

[14] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: a new learning scheme of feedforward neural networks. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, volume 2, pages 985–990 vol.2, 2004.

[15] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489–501, 2006.

[16] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. *German National Research Center for Information Technology, Technical Report*, GMD 148, 2001. Contains errors.

[17] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks - with an erratum note. *German National Research Center for Information Technology, Technical Report*, GMD 148, 2001. Corrected version from 2010.

[18] H. Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach. *German National Research Center for Information Technology, Technical Report*, GMD 159, 2002.

[19] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

[20] J. Jiang and Y.-C. Lai. Model-free prediction of spatiotemporal dynamical systems with recurrent neural networks: Role of network spectral radius. *Physical Review Research*, 1:033056, 2019.

[21] B. Lang. Monotonic multi-layer perceptron networks as universal approximators. In *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005*, pages 31–37, 2005.

[22] F. Lauer and G. Bloch. Incorporating prior knowledge in support vector regression. *Machine Learning*, 70(1):89–118, 2008.

[23] D. Li, M. Han, and J. Wang. Chaotic time series prediction based on a novel robust echo state network. *IEEE Transactions on Neural Networks and Learning Systems*, 23(5):787–799, 2012.

[24] Y. Li and F. Li. PSO-based growing echo state network. *Applied Soft Computing*, 85:105774, 2019.

[25] M. Lukoševičius. A practical guide to applying echo state networks. In *Neural Networks: Tricks of the Trade – Second Edition*, pages 659–686. Springer, Berlin, Heidelberg, 2012.

[26] M. Lukoševičius, H. Jaeger, and B. Schrauwen. Reservoir computing trends. *Künstliche Intell.*, 26(4):365–371, 2012.

[27] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[28] S.-X. Lun, X.-S. Yao, and H.-F. Hu. A new echo state network with variable memory length. *Information Sciences*, 370-371:103–119, 2016.

[29] S.-X. Lun, X.-S. Yao, H.-Y. Qi, and H.-F. Hu. A novel model of leaky integrator echo state network for time-series prediction. *Neurocomputing*, 159:58–66, 2015.

[30] Q. Ma, E. Chen, Z. Lin, J. Yan, Z. Yu, and W. W. Y. Ng. Convolutional multitimescale echo state network. *IEEE Transactions on Cybernetics*, 51(3):1613–1625, 2021.

[31] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[32] T. Natschläger, W. Maass, and H. Markram. The "liquid computer": A novel strategy for real-time computing on time series. *TELEMATIK*, 8(1):39–43, 2002.

[33] K. Neumann. *Reliability of Extreme Learning Machines*. PhD thesis, Faculty of Technology, Bielefeld University, 2013.

[34] K. Neumann, M. Rolf, and J. J. Steil. Reliable integration of continuous constraints into extreme learning machines. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 21(supp02):35–50, 2013.

[35] J. Qiao, F. Li, H. Han, and W. Li. Growing echo-state network with multiple subreservoirs. *IEEE Transactions on Neural Networks and Learning Systems*, 28(2):391–404, 2017.

[36] A. Rodan and P. Tiňo. Minimum complexity echo state network. *IEEE Trans. Neural Networks*, 22(1):131–144, 2011.

[37] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. J. Gomez. Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779, 2007.

[38] B. Schrauwen, D. Verstraeten, and J. M. V. Campenhout. An overview of reservoir computing: theory, applications and implementations. In *ESANN 2007, 15th European Symposium on Artificial Neural Networks*, pages 471–482, 2007.

[39] J. J. Steil. Backpropagation-decorrelation: online recurrent learning with O(N) complexity. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, volume 2, pages 843–848, 2004.

[40] J. J. Steil. Memory in backpropagation-decorrelation O(N) efficient online recurrent learning. In *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, 15th International Conference*, pages 649–654, 2005.

[41] J. J. Steil. Online stability of backpropagation–decorrelation recurrent learning. *Neurocomputing*, 69(7):642–650, 2006.

[42] J. J. Steil. Online reservoir adaptation by intrinsic plasticity for backpropagation–decorrelation and echo state learning. *Neural Networks*, 20(3):353–364, 2007.

[43] Z. Sun, Z. Zhang, H. Wang, and M. Jiang. Cutting plane method for continuously constrained kernel-based regression. *IEEE Transactions on Neural Networks*, 21(2):238–247, 2010.

[44] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391–403, 2007.

[45] H. Wang and X. Yan. Optimizing the echo state network with a binary particle swarm optimization algorithm. *Knowledge-Based Systems*, 86:182–193, 2015.

[46] Z. Wang, Y.-R. Zeng, S. Wang, and L. Wang. Optimizing echo state network with backtracking search optimization algorithm for time series forecasting. *Engineering Applications of Artificial Intelligence*, 81:117–132, 2019.

[47] Y. Ye, M. J. Todd, and S. Mizuno. An $O(\sqrt{n}L)$-iteration homogeneous and self-dual linear programming algorithm. *Mathematics of Operations Research*, 19(1):53–67, 1994.

[48] I. B. Yildiz, H. Jaeger, and S. J. Kiebel. Re-visiting the echo state property. *Neural Networks*, 35:1–9, 2012.

[49] P. Yu, L. Miao, and G. Jia. Clustered complex echo state networks for traffic forecasting with prior knowledge. In *2011 IEEE International Instrumentation and Measurement Technology Conference*, pages 1–5, 2011.

Appendices

A1 ESN Derivatives

A1.1 Derivative with respect to the input

The M^i-th derivative of $\mathbf{y}^{\text{cont}}$ with respect to $u_{m_1^i}, \ldots, u_{m_{M^i}^i}$ results as

$$\begin{aligned} D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{y}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) &= D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{W}^{\text{out}} \begin{bmatrix} 1 \\ \mathbf{u} \\ \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix} \\ &= \mathbf{W}^{\text{out}} \begin{bmatrix} 0 \\ D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{u} \\ D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix}, \end{aligned}$$

where the derivatives of $\mathbf{u}$ and $\mathbf{x}^{\text{cont}}$ are obtained as

$$D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{u} = \begin{cases} \underbrace{\begin{bmatrix} 0 & \ldots & 0 & 1 & 0 & \ldots & 0 \end{bmatrix}^T}_{\text{1}\text{-}m_1^i\text{-th position}} & \text{if } M^i = 1, \\ \begin{bmatrix} 0 & \ldots & 0 & 0 & 0 & \ldots & 0 \end{bmatrix}^T & \text{otherwise,} \end{cases}$$

and

$$\begin{aligned} D_{\mathbf{u}}^{\mathbf{m}^i} \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) &= \underbrace{D_{\mathbf{u}}^{\mathbf{m}^i} (1 - \alpha) \mathbf{x}^{\text{prev}}}_{=0} + D_{\mathbf{u}}^{\mathbf{m}^i} \alpha \sigma \Bigg(\underbrace{\mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u} \end{bmatrix} + \mathbf{W} \mathbf{x}^{\text{prev}}}_{=: f(\mathbf{u})} \Bigg) \\ &= \alpha \cdot \sigma^{(M^i)}(f(\mathbf{u})) \cdot \frac{\partial}{\partial u_{m_1^i}} f(\mathbf{u}) \cdots \frac{\partial}{\partial u_{m_{M^i}^i}} f(\mathbf{u}) \\ &= \alpha \cdot \sigma^{(M^i)}(f(\mathbf{u})) \cdot \frac{\partial}{\partial u_{m_1^i}} \mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u} \end{bmatrix} \cdots \frac{\partial}{\partial u_{m_{M^i}^i}} \mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u} \end{bmatrix} \\ &= \alpha \cdot \sigma^{(M^i)} \left(\mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u}(t) \end{bmatrix} + \mathbf{W} \mathbf{x}^{\text{prev}} \right) \cdot \mathbf{W}^{\text{in}}_{\cdot m_1^i} \cdots \mathbf{W}^{\text{in}}_{\cdot m_{M^i}^i}. \end{aligned}$$

A1.2 Derivative with respect to the reservoir state

The M^i-th derivative of $\mathbf{y}^{\text{cont}}$ with respect to $x^{\text{prev}}_{\hat{m}^i_1}, \ldots, x^{\text{prev}}_{\hat{m}^i_{M^i}}$ results as

$$\begin{aligned} D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{y}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) &= D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{W}^{\text{out}} \begin{bmatrix} 1 \\ \mathbf{u} \\ \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix} \\ &= \mathbf{W}^{\text{out}} \begin{bmatrix} 0 \\ 0 \\ D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) \end{bmatrix}, \end{aligned}$$

where the derivatives of $\mathbf{x}^{\text{cont}}$ is obtained as

$$\begin{aligned} D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{cont}}(\mathbf{u}, \mathbf{x}^{\text{prev}}) &= D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} (1-\alpha) \cdot \mathbf{x}^{\text{prev}} + D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \alpha \cdot \sigma\Bigg(\underbrace{\mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u} \end{bmatrix} + \mathbf{W}\mathbf{x}^{\text{prev}}}_{=: f(\mathbf{x}^{\text{prev}})} \Bigg) \\ &= (1-\alpha) \cdot D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{prev}} + \alpha \cdot \sigma^{(\hat{M}^i)}(f(\mathbf{x}^{\text{prev}})) \cdot \frac{\partial}{\partial x^{\text{prev}}_{\hat{m}^i_1}} f(\mathbf{x}^{\text{prev}}) \cdots \frac{\partial}{\partial x^{\text{prev}}_{\hat{m}^i_{\hat{M}^i}}} f(\mathbf{x}^{\text{prev}}) \\ &= (1-\alpha) \cdot D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{prev}} + \alpha \cdot \sigma^{(\hat{M}^i)}(f(\mathbf{x}^{\text{prev}})) \frac{\partial}{\partial x^{\text{prev}}_{\hat{m}^i_1}} \mathbf{W}\mathbf{x}^{\text{prev}} \cdots \frac{\partial}{\partial x^{\text{prev}}_{\hat{m}^i_{\hat{M}^i}}} \mathbf{W}\mathbf{x}^{\text{prev}} \\ &= (1-\alpha) \cdot D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{prev}} + \alpha \cdot \sigma^{(\hat{M}^i)} \left(\mathbf{W}^{\text{in}} \begin{bmatrix} 1 \\ \mathbf{u}(t) \end{bmatrix} + \mathbf{W}\mathbf{x}^{\text{prev}} \right) \cdot \mathbf{W}_{\cdot \hat{m}^i_1} \cdots \mathbf{W}_{\cdot \hat{m}^i_{\hat{M}^i}}. \end{aligned}$$

Here, the derivative of $\mathbf{x}^{\text{prev}}$ is defined similar to the one of $\mathbf{u}$ above:

$$D^{\hat{\mathbf{m}}^i}_{\mathbf{x}} \mathbf{x}^{\text{prev}} = \begin{cases} \begin{bmatrix} 0 & \ldots & 0 & 1 & 0 & \ldots & 0 \end{bmatrix}^T & \text{if } \hat{M}^i = 1, \\ \begin{bmatrix} 0 & \ldots & 0 & 0 & 0 & \ldots & 0 \end{bmatrix}^T & \text{otherwise.} \end{cases}$$

$\hat{m}^i_1$-th position

A2 Linear Least Squares as Quadratic Program

The reduction of a CLLS to a QP requires only a reduction of the objective function. The constraints are equally defined in both problem definitions:

Constrained Least Squares		**Quadratic Program**	
minimize	$\|\|\mathbf{Ax}-\mathbf{b}\|\|_2^2$	minimize	$\mathbf{x}^T\mathbf{Qx}+\mathbf{cx}$
subject to	$\mathbf{Gx} \preceq \mathbf{h}$	subject to	$\mathbf{Gx} \preceq \mathbf{h}$

The objective function of the CLLS can be reduced to the QP's objective function as

$$\begin{aligned}||\mathbf{Ax}-\mathbf{b}||_2^2 &= (\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b}) \\ &= (\mathbf{x}^T\mathbf{A}^T-\mathbf{b}^T)(\mathbf{Ax}-\mathbf{b}) \\ &= \mathbf{x}^T\mathbf{A}^T\mathbf{Ax}-\mathbf{x}^T\mathbf{A}^T\mathbf{b}-\mathbf{b}^T\mathbf{Ax}+\mathbf{b}^T\mathbf{b}.\end{aligned}$$

Since the multiplication of $\mathbf{x}^T \in \mathbb{R}^{1\times n}$, $\mathbf{A}^T \in \mathbb{R}^{n\times m}$ and $\mathbf{b} \in \mathbb{R}^{m\times 1}$ is a real number, it equals its transpose:

$$= \mathbf{x}^T\mathbf{A}^T\mathbf{Ax}-2\mathbf{b}^T\mathbf{Ax}+\mathbf{b}^T\mathbf{b}.$$

The offset $\mathbf{b}^T\mathbf{b}$ can be dropped since it doesn't influence the value of $\mathbf{x}$ in an optimal solution. Setting $\mathbf{Q} := \mathbf{A}^T\mathbf{A}$ and $\mathbf{c} := -2\mathbf{b}^T\mathbf{A}$ yields the objective function of a QP in standard form:

$$= \mathbf{x}^T\mathbf{Qx}+\mathbf{cx}$$

A3 Practical Implementation

Overview The CESN framework introduced in this work and used for the following experiments is implemented in Python 3.8[1]. Internally, it uses the scientific computing library NumPy[2]. The optimization tool Gurobi[3] is used as the primary QP solver, but other solvers are supported, too, via the high-level convex optimization interface CVXPY[4]. Nevertheless, it is recommended to use Gurobi, since the detour via the CVXPY interface can reduce the runtime performance and the implementation is optimized for the use with Gurobi. The main interfaces of the CESN implementation are the functions `fit` and `predict`, which, on the whole, behave as described in Section 3.1. The constraint definition is implemented using Python operator overloading, which allows to define constraints easily using the Python standard expression syntax. The basic module of a constraint expression is the class `ConstraintY`, which is used as the abstract `y` in the constraint equation. For example, a constraint that bounds the first output variable upwards to 5 can be defined by

```
Y = ConstraintY()
c1 = Y[0] <= 5
```

The output history can be involved, by adding a second argument to `Y`. Then, the first argument specifies the affected output variable and the second one the number of time steps back in the history. For example, a constraint that forces the second output variable to increase monotonously over time (defined by the first order difference quotient as $(y_2(t)-y_2(t-2))/2 \geq 0$) can be formulated as

```
c2 = (Y[1, 0] - Y[1, -2]) / 2 >= 0
```

Constraints involving difference quotients can further be simplified by specifying them directly:

```
c2 = DQ(1, 1) >= 0
```

Here, `DQ(n, delta)` is internally resolved to a constraint expression representing the n-th order difference quotient with respect to a specific difference Δ.

---

1 Python Programming Language (accessed: March 2021)
`https://www.python.org`

2 NumPy Scientific Computing Library (accessed: March 2021)
`https://www.numpy.org`

3 Gurobi Optimization Tool (accessed: March 2021)
`https://www.gurobi.com`

4 CVXPY Convex Optimization Interface (accessed: March 2021)
`https://www.cvxpy.org`

Numerical Issues The QP formulation introduced in Equation 2.38 and Equation 2.39 and used to optimize the network read-out weights seems to be unstable in the sense of numerical accuracy and solvability, at least in the context of the ESN training reduction as used in this thesis. This issue is tackled by using an alternative, equivalent QP formulation on the one hand and using suitable QP solver settings on the other hand. The alternative QP formulation seems to be more stable and is obtained as

$$\begin{aligned} \underset{\tilde{\mathbf{x}}\in\mathbb{R}^m,\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad & \tilde{\mathbf{x}}^T\tilde{\mathbf{x}} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}-\tilde{\mathbf{x}}=\mathbf{b} \\ & \mathbf{G}\mathbf{x}\preceq\mathbf{h} \end{aligned} \tag{A3.1}$$

for an $\mathbf{A}\in\mathbb{R}^{m\times n}$, $\mathbf{b}\in\mathbb{R}^m$, $\mathbf{G}\in\mathbb{R}^{p\times n}$ and $\mathbf{h}\in\mathbb{R}^p$. Here, the term $\mathbf{A}\mathbf{x}-\mathbf{b}$ in the objective function in Equation 2.30 is substituted by a slack variable $\tilde{\mathbf{x}}$, where the substitution is realized via constraints. This QP can be rephrased to a QP with only one optimization variable $\mathbf{x}$ as

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^{m+n}}{\text{minimize}} \quad & \mathbf{x}^T\mathbf{Q}\mathbf{x} \\ \text{subject to} \quad & \tilde{\mathbf{A}}\mathbf{x}=\mathbf{b} \\ & \tilde{\mathbf{G}}\mathbf{x}\preceq\mathbf{h} \end{aligned} \tag{A3.2}$$

with

$$\mathbf{Q}=\begin{bmatrix}\mathbf{I}_m & \mathbf{0}_{m\times n}\\ \mathbf{0}_{n\times m} & \mathbf{0}_{n\times n}\end{bmatrix}=\begin{bmatrix}1&0&\cdots&0&0&\cdots&0\\0&1&\cdots&0&0&\cdots&0\\\vdots&\vdots&\ddots&\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&1&0&\cdots&0\\0&0&\cdots&0&0&\cdots&0\\\vdots&\vdots&\ddots&\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&0&0&\cdots&0\end{bmatrix}\in\mathbb{R}^{m+n\times m+n}, \tag{A3.3}$$

$$\tilde{\mathbf{A}}=\begin{bmatrix}-\mathbf{I}_m & \mathbf{A}\end{bmatrix}=\begin{bmatrix}-1&0&\cdots&0&a_{11}&\cdots&a_{1n}\\0&-1&\cdots&0&a_{21}&\cdots&a_{2n}\\\vdots&\vdots&\ddots&\vdots&\vdots&\ddots&\vdots\\0&0&\cdots&-1&a_{m1}&\cdots&a_{mn}\end{bmatrix}\in\mathbb{R}^{m\times m+n} \tag{A3.4}$$

and

$$\tilde{\mathbf{G}}=\begin{bmatrix}\mathbf{0}_{p\times m} & \mathbf{G}\end{bmatrix}=\begin{bmatrix}0&\cdots&0&g_{11}&\cdots&g_{1n}\\\vdots&\ddots&\vdots&\vdots&\ddots&\vdots\\0&\cdots&0&g_{p1}&\cdots&g_{pn}\end{bmatrix}\in\mathbb{R}^{p\times m+n}. \tag{A3.5}$$

Beside this reformulation, we set different Gurobi solver settings to tackle numerical issues. In some cases, the homogeneous barrier algorithm [47] can solve problems, for which the default barrier algorithm cannot find a valid solution.[5] This algorithm is a bit

[5] Gurobi Guidelines for Numerical Issues (accessed: March 2021) `files.gurobi.com/Numerics.pdf`

slower than the default one.[6] Another important parameter is the *feasibility tolerance*.[7] As its name already suggests, this parameter specifies to which tolerance all constraints must be satisfied. A constraint $\mathbf{g}\mathbf{x} \leq h$ will be considered to hold, if $\mathbf{g}\mathbf{x} - h \leq \mathit{feasibilityTolerance}$.[8] This tolerance needs to be taken into account in the CESN training algorithm, where the constraints are evaluated independently from the Gurobi solver. If the tolerance in the CESN training algorithm is smaller than the Gurobi's one, it happens in rare cases that Gurobi considers constraints as satisfied which are not considered as satisfied by the CESN training algorithm. This would lead to a non-terminating loop of training iterations and QP solving.

A third aspect of the solving process is the termination with a sub-optimal solution. If Gurobi is not able to find an optimal solution in a reasonable amount of time, it terminates with the best solution found until then. This solution is maybe not optimal, or optimal but the solver could not proof its optimality. But optimal solutions are not necessary at all and in many cases the returned sub-optimal solution is already sufficient good. Other neural network training algorithms, e.g., gradient descent based algorithms, do not yield (global) optimal solutions either.

Memory limits The size of the matrices in the QP become very large even for middle sized problem formulations. This is mainly caused by the large QP matrices, resulting from the reduction presented in Chapter 2 and the reformulated QP formulation presented above. The sizes of these matrices depend on the size of the training dataset, the number of network outputs, the reservoir size and the number of applied constraints. Since most of these matrices are very sparse, we use the scientific computing library SciPy[9], which has an exhaustive sparse matrix framework, fully compatible with NumPy. But even if Gurobi supports Scipy's sparse matrices for the QP definition, the actual solving process requires much more memory, even for sparse matrices. One solver setting that reduces the required amount of memory is to disable Gurobi's pre-solve algorithm, which does not seem to extend the solving time much. Furthermore, Neumann [33] proposed a simpler (and much smaller) QP formulation, which can be used, if the CESN outputs can be assumed as independent, also in the sense of prior knowledge. By default, the CESN is trained for all output units at once. This allows to define constraints involving several outputs. But in the case that every constraint involves only one output unit, the CESN training can be considered separately for each output unit. This does not only simplify the QP in the sense of computation costs, but also in the sense of required memory. All three learning tasks presented in this thesis deal with independent outputs only.

---

6 Gurobi Reference to Parameter BarHomogeneous (accessed: March 2021)
`https://www.gurobi.com/documentation/9.1/refman/barhomogeneous.html`

7 Gurobi Reference to Parameter FeasibilityTol (accessed: March 2021)
`https://www.gurobi.com/documentation/9.1/refman/feasibilitytol.html`

8 Gurobi Reference to Tolerances and user-scaling (accessed: March 2021)
`https://www.gurobi.com/documentation/9.1/refman/tolerances_and_user_scalin.html`

9 SciPy Scientific Computing Library (accessed: March 2021)
`https://www.scipy.org`

A4 Gridsearch parameters

A4.1 Illustrative Examples

Example 1: Difference Quotient Constraints

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0
Leaking rate	0.1, **0.2**(**17**), 0.3(13), 0.7, 0.9
Spectral radius	0.1, **0.2**(**26**), 0.4(3), 0.6, 0.7(1), 0.99
Input scaling	0.01, 0.05, 0.1, 0.5(9), **1**(**18**), 2(2), 4(1)
Regularization	$\mathbf{10^{-8}}$(**27**), 10^{-5}(2), 10^{-4}(1), 10^{-3}, 1
Reservoir size	100(6), **500**(**24**)

Table A4.1: Gridsearch: Illustrative example 1 (ESN)

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0, Regularization: 10^{-8}
Leaking rate	0.1(1), **0.3**(**29**), 0.7
Spectral radius	0.1, 0.2, 0.4, **0.6**(**27**), 0.9(3)
Input scaling	0.1(2), **0.5**(**26**), 1(2), 2, 4
Reservoir size	100, **500**(**30**)

Table A4.2: Gridsearch: Illustrative example 1 (CESN)

Example 2: Periodicity Constraints

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0
Leaking rate	0.1, **0.3**(**25**), 0.7(5)
Spectral radius	0.4(3), **0.6**(**23**), 0.9(4)
Input scaling	0.1(4), **0.5**(**17**), 1.0(8), 2.0(1), 4.0
Regularization	$\mathbf{10^{-8}}$(**23**), 10^{-5}(7), 10^{-3}
Reservoir size	100(7), **500**(**23**)

Table A4.3: Gridsearch: Illustrative example 2 (ESN)

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0
Leaking rate	0.1(2), 0.3(13), **0.7**(**15**)
Spectral radius	0.4(7), **0.6**(**15**), 0.9(8)
Input scaling	0.1, 0.5(12), **1.0**(**14**), 2.0(4), 4.0
Regularization	10^{-8}(5), $\mathbf{10^{-5}}$(**16**), 10^{-3}(9)
Reservoir size	**100**(**17**), 500(13)

Table A4.4: Gridsearch: Illustrative example 2 (ESN (mean))

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0
Leaking rate	**0.1**(**30**), 0.3, 0.7
Spectral radius	**0.4**(**21**), 0.6(8), 0.9(1)
Input scaling	0.1, 0.5(2), **1.0**(**28**), 2.0, 4.0
Regularization	10^{-8}, $\mathbf{10^{-5}}$(**30**), 10^{-3}
Reservoir size	100, **500**(**30**)

Table A4.5: Gridsearch: Illustrative example 2 (ESN (mean extended))

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0, Regularization: 10^{-8}
Leaking rate	**0.1**(**26**), 0.3(4), 0.7
Spectral radius	0.4(7), **0.6**(**23**), 0.9
Input scaling	**0.1**(**18**), 0.5(10), 1.0(2), 2.0, 4.0
Reservoir size	100(12), **500**(**18**)

Table A4.6: Gridsearch: Illustrative example 2 (CESN (training data only))

Hyperparameter	Values (Occurrences out of 30 repetitions)
Fixed Parameters	Sparsity: 0.0, Regularization: 10^{-8}
Leaking rate	**0.1**(**30**), 0.3, 0.7
Spectral radius	0.4(1), **0.6**(**29**), 0.9
Input scaling	**0.1**(**29**), 0.5(1), 1.0, 2.0, 4.0
Reservoir size	100(1), **500**(**29**)

Table A4.7: Gridsearch: Illustrative example 2 (CESN)

A4.2 Application: Satellite Image Forecasting

Hyperparameter	Values (Occurrences out of 15 repetitions)
Fixed Parameters	Sparsity: 0.0, Regularization: 10^{-8}, Reservoir size: 750
Leaking rate	0.85, **1.0**(**15**)
Spectral radius	0.9(1), **0.99**(**14**), 1.25
Input scaling	0.01, 0.05(5), **0.1**(**10**), 0.5

Table A4.8: Gridsearch: Satellite image forecasting application

A5 Master's Thesis Scope

When dealing with model learning for time series like speech recognition, text analysis, or stock market prediction, using Recurrent neural networks (RNN) is state-of-the-art. Reservoir networks are a specific kind of RNNs and have proven to be an efficient framework for learning dynamic systems. Instead of training the entire network, only the so-called readout stage is optimized and the weights of all internal connections are assigned randomly and are fixed during training. Since the readout stage may be optimized using training methods like linear regression or Ridge regression, the computational training costs are reduced extremely. In order to increase the performance of the trained network, a-priori knowledge shall be injected using constraint optimization as previously introduced by Neumann, Rolf & Steil, 2013. This novel approach shall then be benchmarked on synthetic data and applied to time series of satellite data.

Task 1 Literature Research on reservoir learning, including literature research to the theoretical background, benchmarks and experimental results and on the constraint optimization technique for training neural networks.

Task 2 The Constraint Optimization Approach shall be implemented and applied to reservoir networks. It is implemented using a suitable framework (e.g. Python or MATLAB) including documentation and tested using some synthetic training data shall provide a first account on the suitability of the method.

Task 3 Satellite Image Data Time Series shall be considered as a specific application domain. From this particular scenario-specific constraints are derived. The images are either preprocessed using state-of- the-art image processing methods like Fourier transformations or inputted in raw format. In cased, no suitable data can be obtained or no reasonable domain-specific constraints can be derived, a simpler standard time-series forecasting scenario will serve as a fallback. Results are evaluated w.r.t. to the domain-specific setting.

Task 4 The Master Thesis shall document the results, including the implementation according to the usual standards for software-related work.

Braunschweig, April 6, 2021