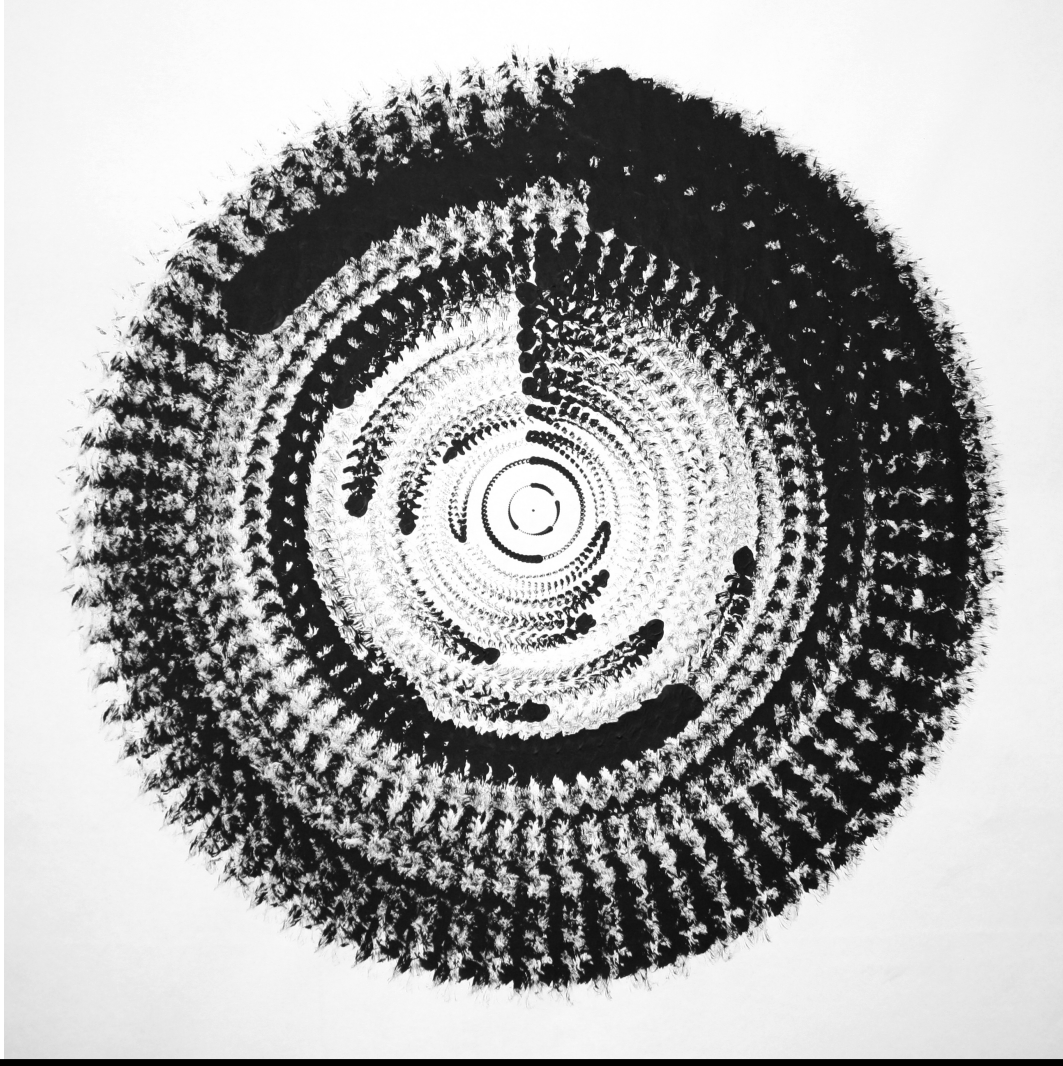


Automated Anomaly Detection In Chaotic Time Series

A Reservoir Computing Approach

Master Thesis

by Niklas Heim

September 4th, 2018

Advisors

Professor Brian Vinter

Asst. Professor James E. Avery

University of Copenhagen

Niels Bohr Institute

Acknowledgements

I would like to express my gratitude to *eScience* and *TeamOcean*, my two research groups at the University of Copenhagen. Their Professors *Brian Vinter* and *Markus Jochum* have provided a stimulating, collaborative environment for interesting research which I have highly appreciated. Further I want to thank all the group members, that always had an open ear for my countless questions and have had a great influence on this thesis with their ideas and suggestions. Especially, I want to thank *James Avery* who has provided a tremendous amount of help, feedback, and inspiration during the whole duration of this project without ever expecting anything in return.

A big thank you also goes to all my friends who have read and re-read this thesis and made suggestions for improvements. Additionally, most of the visual appeal of this work is owed to *Fabian Dornhecker* from the *BamOida Interrogang***?** The cover image was drawn by a neural network at the Wentworth Institute of Technology.

Finally I want to thank my parents for always supporting and encouraging me in my undertakings despite this 'rather aberrant choice of a career'.

Abstract

The purpose of this study is to create a framework that is able to automatically detect unusual behaviour in non-linear dynamical systems. We assume no prior information about the physics that govern these dynamics, so there is no knowledge about the kind of anomaly that we are looking for. This is motivated by the large amounts of output that state of the art, eddy-resolving ocean models produce. These large datasets might contain unknown physical behaviour, such as the recently discovered Kuroshio anomaly (Sec. 1.8). It is impossible for humans to evaluate all the available climate model data within an acceptable time frame. An automated anomaly detection is a first step towards harnessing the full potential of such expensive simulations and could contribute to a deeper understanding of the ocean circulation.

The detection problem is approached by trying to define what is normal, so that everything that looks significantly different from this norm can be treated as anomalous. This norm is found by predicting the future evolution of a system that has been observed for a certain amount of time. This is not a trivial task, because non-linear systems can exhibit chaotic behaviour which makes their prediction notoriously hard. However, recent research indicates that it can be solved by employing a special kind of recurrent neural network. Once the prediction is extracted from the network, it can be compared to the true values of the dataset and where they deviate significantly, a potential anomaly is found.
The type of recurrent network that is used is called *echo state network* and belongs to the class of *reservoir computing* methods. They feature a comparatively low computational cost and have been shown to be able to predict chaotic systems with surprising accuracy [Pathak et al. 2018].

Concepts of machine learning and artificial intelligence are, despite their proven effectiveness in other fields, still very sparsely utilized in climate research. Therefore this work also serves as a showcase of what can be done by expanding the set of standard analysis tools towards these methods. The final result is the successful detection of the Kuroshio anomaly in the turbulent ocean dataset.

The Python package that implements the anomaly detection is published in a GitHub repository (`https://github.com/nmheim/torsk`).

Contents

Chapter 1
Introduction

This thesis will explore the possibility of creating an anomaly detection for non-linear time series. In its broadest sense this means finding unusual, unexpected, or new patterns in a given dataset. In this introductory chapter, the motivation for searching for anomalous behaviour is described alongside the requirements for a general anomaly detection algorithm. Because many physical systems that are of interest for an outlier detection are non-linear and possibly even show *chaotic* behaviour, a brief definition of chaotic time series is given, followed by three concrete exemplary datasets.

The problem of detecting anomalies has been studied in many fields, which has lead to a great number of different approaches to the problem, some of which will be summarized in Chapter 2; Chapter 3 introduces Neural Networks and their potential for predicting arbitrary time series, their implementation is described in Chapter 4 and finally Chapter 5 presents the so-called *echo state network* applied to the three datasets that were introduced earlier.

1.1 Motivation

The identification of unusual or aberrational behaviour is treated in many other scientific fields: The purpose of magnetic resonance imaging (MRI) is to find tissues that are not expected to be found at a certain location in the body, indicating a tumor. Biology studies genetic anomalies, *mutations*, that can cause either illness or an increased chance of survival of an individual. Engineers monitor their equipment to detect early warning signs before a machine breaks. Banks try to identify fraudulent transactions based on peculiarities in credit card data. Anomalies in weather data can give early hints on upcoming droughts, storms or other weather phenomena. DNA sequence mutations, engine monitoring, fraud detection, and weather data are examples of sequential datasets that can be regarded as time series. Detecting outliers in time series can help prevent undesired behaviour or develop an early warning systems. With the growing amount of data that is being produced in many different fields it becomes infeasible to manually analyze the given datasets. It is therefore highly desirable to create an anomaly detection that is as general as possible with regard to the data it can successfully process.

A field which could benefit immensely from an automated anomaly detection is ocean modelling. Large scale, high resolution simulations that cover the whole Earth with more than 30 different variables such as temperature, velocity, and density easily take up tens of gigabytes for a single time step. The vast majority of the simulated ocean, much like the real ocean, is almost completely unexplored. Potentially unknown physical behaviour that could be hidden in these datasets could be found by an automated outlier search. An example of such an anomaly is the state changing ocean current called *Kuroshio* on the coast of Japan. In irregular periods of several years it changes from an elongated to a contracted state. This phenomenon has just recently come to the attention of the scientific community and its origin is subject of vivid debate. A detection of similar anomalies would be a very interesting finding in itself, but it could also contribute to a deeper understanding of the Kuroshio anomaly and the ocean circulation as a whole.

A large part of the anomaly detection model consists of a Neural Network. They have been successfully applied to a very large number of different problems, but have not yet found their way into the standard analysis tools of climate scientists. With this thesis we hope show what could be achieved with an AI driven approach to climate research.

1.2 Defining Normality

This thesis aims for the creation of an automatic detection algorithm that can analyze large spatio-temporal datasets. We assume no prior knowledge about the physics that produces the data, so there is no knowledge about the kind of anomaly that we are looking for. This requires to define what is *normal*, so that everything that looks significantly different than this norm can be treated as anomalous. In time series this norm can be defined by trying to predict the future evolution based on a previously observed history. This means that we want to create a model that consumes an input sequence $\mathbf{u}$ and returns a prediction

sequence $\mathbf{y}$ (formal description in Sec. 2.2.3):

$$\mathbf{y} = F(\mathbf{u}) \tag{1.1}$$

The acquired prediction can subsequently be compared with the true values that the sequence takes on for the predicted interval. As soon as a reasonably good prediction is made, the actual detection of an anomaly becomes quite simple. Based on the degree of the deviation of prediction and truth, an *normality score* (Sec. 2.2.4) can be calculated to define how large the deviation must be in order to count as an anomaly.

1.3 Applications

The generality of the algorithm is showcased with three exemplary datasets. Each of the datasets will be described in more detail in the sections 1.6-1.8 of this chapter, but in short they can be summarized as follows.
The first test-problem is a scalar chaotic system governed by the Mackey-Glass equation. Scalar systems will further be referred to as zero-dimensional (0D). Next, the algorithm is applied to a real-world example: climate records that were obtained from Greenland ice-cores and include well known anomalies that are called Dansgaard-Oeschger (DO) events. The time series consists of two markers that are fed to the detection algorithm, which is slightly increasing the complexity of the task. The DO dataset can be regarded as a very small one-dimensional (1D) system, as the input is a vector with two components. Finally, the performance of the algorithm on a 2D system is analyzed. It consists of a sequence of simulated sea surface height (SSH) images of the previously mentioned Kuroshio region next to Japan. A successful anomaly detection on this dataset would equate to an automated novelty detection in a vast amount of climate data.

1.4 Outline

The second chapter gives a brief introduction to what anomalies are, describes some frequently used detection techniques, and defines the goal of the predictive anomaly detection that will be used. In the third chapter, whose subject is Neural Networks, the theoretical background that is needed to implement an automated time series prediction is laid out. It includes an introduction to feedforward and recurrent networks before describing a new machine learning concept called *reservoir computing* (RC). The RC approach, in comparison to traditional ML, is less computationally expensive and was shown to be capable of forecasting chaotic systems surprisingly well in a paper by [Jaeger and Haas 2004] and for the first time for high dimensional datasets by [Pathak et al. 2018]. The implementation of the algorithm is sketched in chapter 4 and the results are presented in chapter 5.

1.5 Chaotic Time Series

Time series are characterized by a chronological sequence of events. In this thesis only discrete time series, where each data point is associated with a timestamp, are treated. The basic components of a *linear* time series are level, trend, seasonal, and random effects. *Level* is simply the current value of the series, the *trend* describes the increase and decrease from one step to another. The *seasonal* component refers to recurring patterns that can be explained by some kind of seasonal influence like the tides. *Random* effects describe the statistical fluctuations of the series. This is a typical linear treatment of the problem of analysing sequences. The complex problem is broken down into parts which can be solved individually and finally linearly recombined. For *non-linear* systems this cannot be done so easily. Various physical systems exhibit this non-linear behaviour which can, under certain conditions, lead to *chaotic* time series.

> *Chaos is aperiodic long-term behaviour in a deterministic system that exhibits sensitive dependence on initial conditions.*
>
> (S. H. Strogatz)

By aperiodic we mean behaviour that cannot be explained by strictly periodic effects. There are no random inputs needed for this aperiodicity to occur, which is suggested by the deterministic nature of the system. Its capricious behaviour arises from the non-linearity. The sensitivity to initial conditions is the most commonly noted feature of chaotic systems. It means that two points $x(t)$ and $x(t)+\delta(t)$ that start out infinitesimally close to each other will quickly diverge and evolve in completely different ways. Their separation may initially grow exponentially fast

$$||\delta(t)|| \sim ||\delta_0|| e^{\lambda t}. \tag{1.2}$$

In this case the number λ is called Lyapunov exponent. In systems where $\lambda > 0$, one finds chaotic behaviour and the larger λ grows the more sensitive is the system to initial perturbations. The Lyapunov exponent for the famous Lorenz attractor [Lorenz 1963] can be computationally found to be $\lambda \approx 0.9$. Chaotic systems are not the same as *unstable* systems, which also exhibit this sensitivity to initial conditions. Unstable systems however, diverge towards infinity, while in chaotic systems $\delta(t)$ is bounded by the radius of the *attractor*. A formal definition of attractors is out of the scope of this work, but the interested reader may be referred to the book *Non-linear Dynamics and Chaos* by [Strogatz 2018].

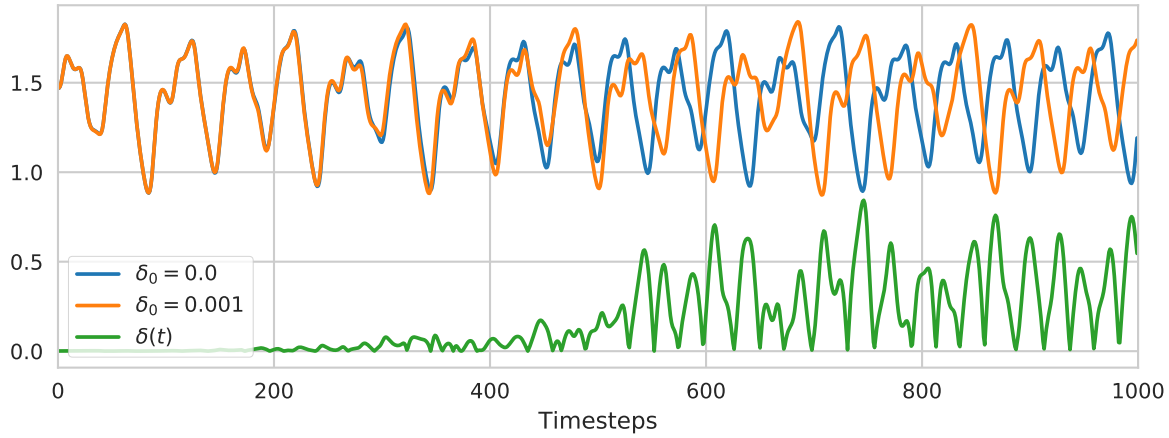


Figure 1.1: *Mackey-Glass time series created from Eq. 1.3. It nicely shows the aperiodic behaviour of chaotic time series, which makes it difficult to predict the next cycle. The two different evolutions are caused by a separation of $\delta_0 = 0.001$. There are no anomalies in the sequence. They will be introduced in Sec. 5.2 by slightly varying γ over time.*

1.6 Mackey-Glass System

The Mackey-Glass (MG) system is a simple *delay* differential equation, that exhibits chaotic behaviour under certain conditions. It is studied extensively in non-linear dynamics and serves as a benchmark for chaotic prediction algorithms. It is defined by

$$\frac{\partial x}{\partial t} = \beta \frac{x_\tau}{1 + x_\tau^n} - \gamma x, \tag{1.3}$$

where β and γ are constants and x_τ denotes the value $x(t-\tau)$, representing the delay. Two sequences that are created by solving Eq. 1.3 with the values $\beta = 0.1$, $\gamma = 0.2$ and a delay of $\tau = 17$ are shown in Fig. 1.1. To solve the MG system initial values of the length of the delay τ need to be specified. The two different evolutions in the plot are created by slightly perturbing the initial values by δ_0. Additionally the plot shows how the separation $\delta(t)$ of the two runs evolves. As expected, the peaks of the separation look very much like an exponential for some time. The local lyapunov exponent (LLE, [Eckhardt and Yao 1993]) of the Mackey-Glass system was calculated to LLE ≈ 0.01 by [Sprott 2007].

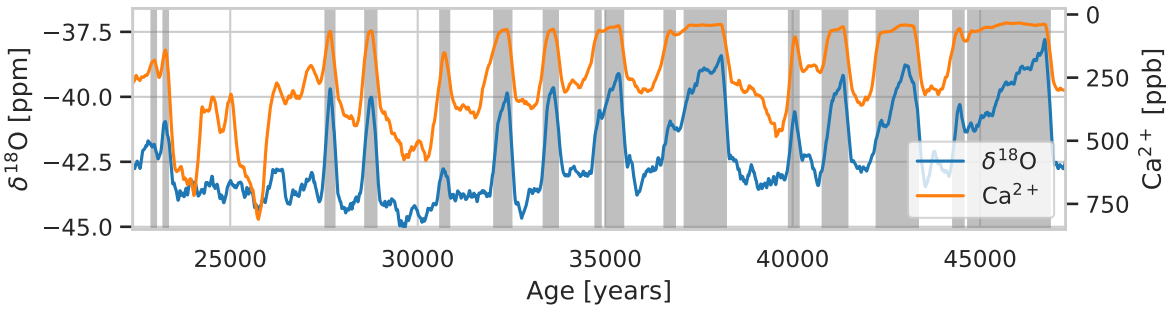


Figure 1.2: *A sample of the $\delta^{18}O$ (related to temperature) and Ca^{2+} (dust content) sequences. Dansgaard-Oeschger are marked by the grey shaded regions.*

1.7 Dansgaard-Oeschger Events

During the last glacial period the North Atlantic region was subject to a number of large climatic fluctuations known as Dansgaard-Oeschger (DO) events. They describe abrupt increases in surface temperature of up to 15°C over a few decades, followed by a gradual cooling. Automatically detecting DO events in time series that are obtained from Greenland ice-cores could serve as an interesting benchmark for this anomaly detection method. Additionally the dataset serves as a first step towards higher dimensional time series.
The INTIMATE project (INTegrating Ice-core, MArine, and TErrestrial records) has recently managed to extract records of the DO events from Greenland ice-cores that were drilled down to depths of about 3000m and contain climatic information of the past 100 000 years. Two different markers that clearly show DO events are considered here: the so-called $\delta^{18}O$ value is directly linked to past temperatures, while the Ca^{2+} concentration measures dust content, which can be connected to atmospheric conditions such as the atmosphere's circulation pattern.
The $\delta^{18}O$ value measures the ratio of the lighter oxygen isotope ^{16}O and the heavier ^{18}O in precipitation water [S. O. Rasmussen et al. 2014]. Higher temperatures aid the evaporation of water that contains the heavier oxygen isotope, which means that roughly, one per mill change in $\delta^{18}O$ corresponds to a temperature difference of 1.5-4°C. With Greenland ice-cores it is possible to resolve annual temperature oscillations as far back in time as 8000 years. The sequence that is considered here contains annual means because we are not interested in seasonal changes.
The concentration of Ca^{2+} is essentially a measure of the dust content of the ice. More dust indicates stronger winds that carried it to Greenland. By measuring other isotopes it is even possible to determine where the dust comes from, which gives hints on the atmospheric circulation patterns at the time, but these isotopes are not considered here. Fig. 1.2 shows how well the Ca^{2+} and $\delta^{18}O$ values coincide. Not that the time axis shows age, so the time moves forward from right to left. The DO events are indicated by the grey shaded regions.

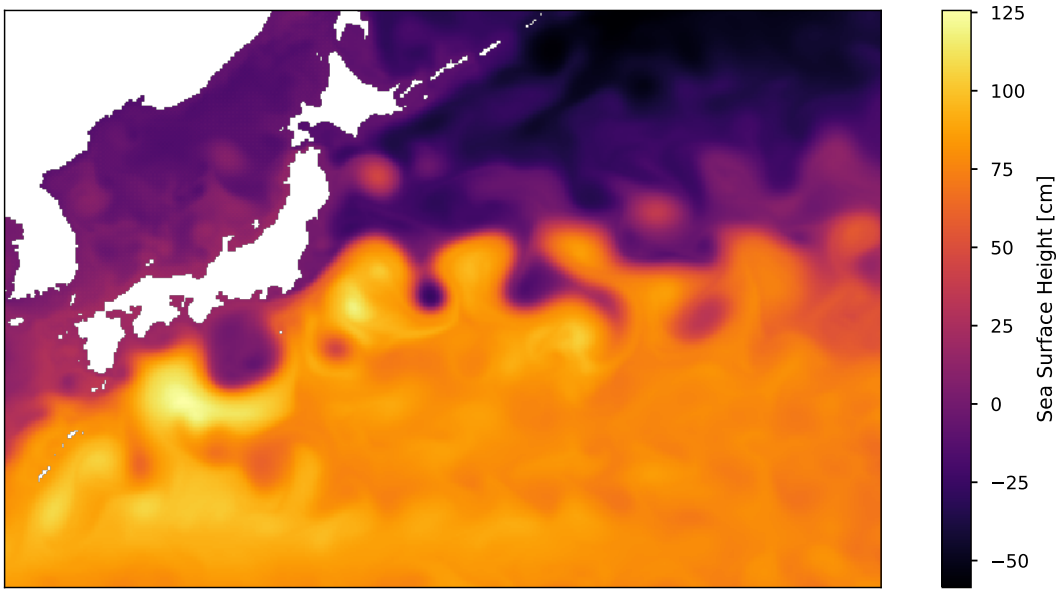


Figure 1.3: *3-day mean of simulated sea surface height data. The Kuroshio is visible at the sharp transition between blue and yellow color. It flows along the coast of Japan before turning into the North Pacific basin. The figure is created by a 440x290 pixel window of the global simulation domain that is 3600x2400 pixels large.*

1.8 Kuroshio

The Kuroshio [JAPANESE: BLACK TIDE] is one of the strongest ocean boundary currents in the world and is the result of the western intensification [Pedlosky 2013] of the ocean circulation in the North Pacific. The 3-day mean of simulated sea surface height (SSH) data (Fig. 1.3) shows the Kuroshio and its extension that reaches into the North Pacific basin. It carries with it large amounts of energy, nutrients and biological organisms, which have a large impact on the local and global climate. To make itself even more relevant, the Kuroshio exhibits a interesting and not yet understood phenomenon. In front of the coast of Japan it oscillates between two distinct states (Fig. 1.4): an elongated (right) and a contracted state (left). The transition between the two states typically takes one to two years and occurs, as it seems, randomly every few years. In 2017 it transitioned to its elongated state for the first time in over a decade, as reported by a Japanese newspaper [Mainichi 2017]. To the interested reader it also describes its impact on the local fishing industry as well as on tide levels and the weather.

The simulations that created the pictures were carried out by *Team Ocean* at the University of Copenhagen. The *Community Earth System Model* (CESM) was used to simulate the global domain with a horizontal resolution of 0.1° and 62 depth layers. It writes out 3-day means for all variables, but in this work only the SSH fields are considered, which results in images of a total size of 3600 x 2400 pixels. A more detailed description of the experimental setup can be found in [Poulsen, Jochum, and Nuterman 2018]. As indicated by Fig. 1.4, the Kuroshio anomaly was reproduced in the CESM simulations. The two plots of the elongated and the contracted state were produced by averaging 200 3-day mean snapshots of the simulation at different time periods. Taking the difference of them should

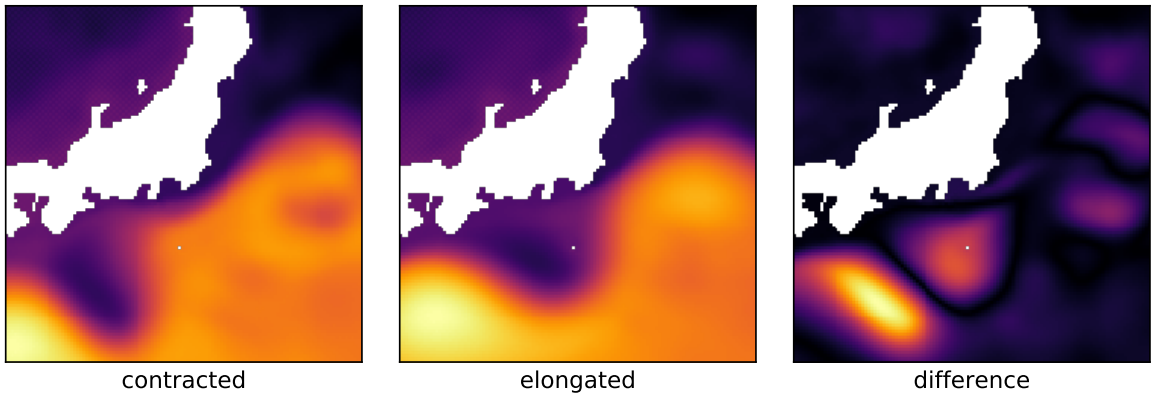


Figure 1.4: *The two distinct states of the Kuroshio created by averaging SSH over two years. The images have a size of 100 x 100 which are sliced out of the global simulation domain of 3600 x 2400 pixels.*

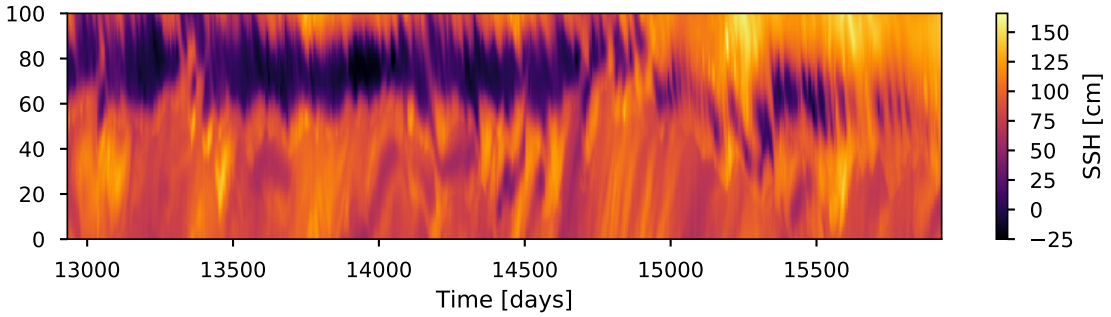


Figure 1.5: *A row of the southern SSH field from Fig. 1.4 over time. The Kuroshio anomaly where the stream transitions from contracted to elongated state is visible between day 14500 and 15000.*

give us an intuition for how a successful anomaly detection should look like.

Detecting the state changes of the Kuroshio with an automated anomaly search could be the first step in creating a way of finding novel behaviour in the vast amount of climate model output, which is essentially as unexplored as the oceans of our real world. Such novelties could, apart from their potential of displaying new physical processes, contribute both to a further understanding of the behaviour of the Kuroshio itself and the ocean circulation patterns as a whole. The automatic detection of the Kuroshio anomaly would mark a major step towards a creation of an algorithm that could find anomalies in noisy and turbulent datasets such as ocean simulations.

Chapter 2

Anomaly Detection

The field of anomaly detection is very broad and many different fields have developed a great variation of different methods for finding outliers. This work is focusing on detecting anomalies in sequences by predicting the expected evolution of a time series and comparing this prediction to real observations (or simply: the truth). Before describing this process in more detail, this chapter defines the different kinds of anomalies that exist in sequences and briefly summarizes two other, common detection methods for sequence outlier detection.

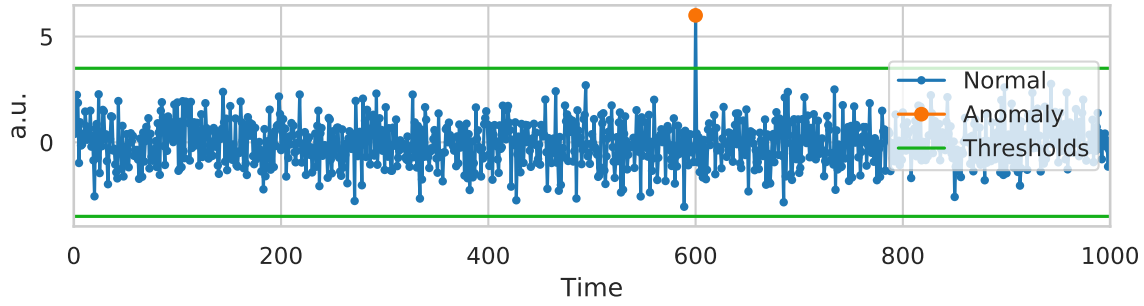


Figure 2.1: *The simple point anomaly can easily be detected by appropriate thresholds (green lines).*

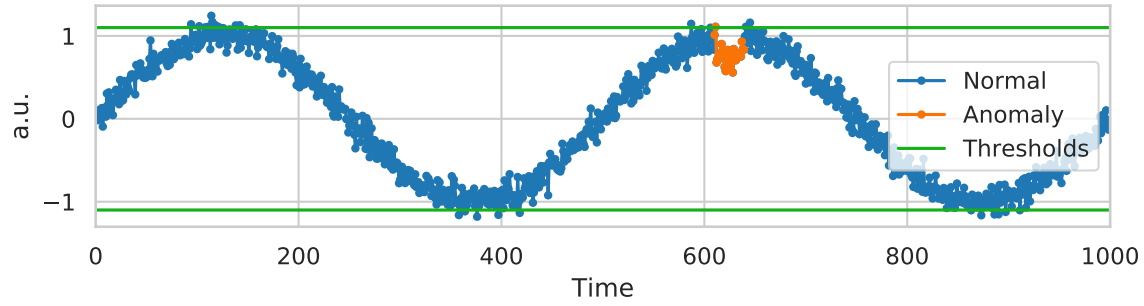


Figure 2.2: *Contextual anomalies are not detectable with simple statistical methods without causing a large number of false positives.*

2.1 Anomalies

An anomaly (or outlier) refers to a pattern in a dataset that does not match an expected behaviour. In time series, there are two basic types of anomalies:

1. *Simple anomalies*, describe instances that can be considered as an outlier only with respect to their value. This is the most basic kind of anomaly which can easily be caught by ordinary, statistical, range-based detection algorithms. The anomaly in Fig. 2.1 consists of a single instance and is hence called a *point* anomaly.

2. *Contextual anomalies* are patterns that are only anomalous within a certain context, but not otherwise. In time series, the context is provided by two attributes: The *contextual attribute* is typically time itself, while the *behavioural attributes* describe the actual values of the examined dataset. Fig. 2.2 shows a contextual anomaly consisting of several abnormal points, which is referred to as a *discord* or *subsequence* anomaly.

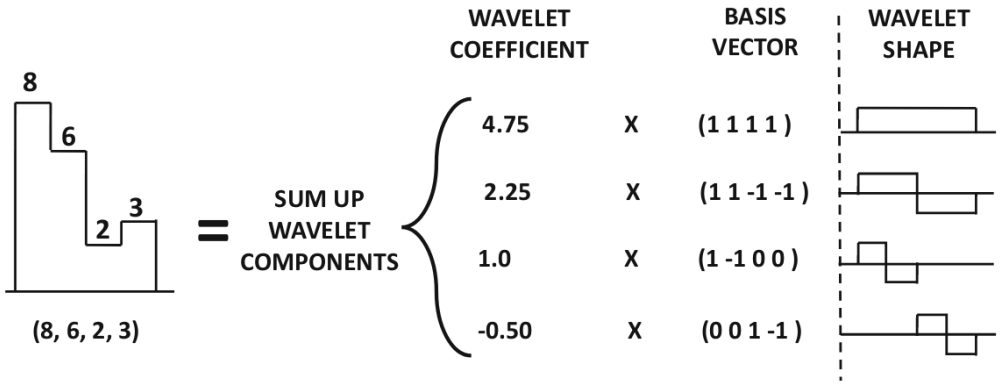


Figure 2.3: *Haar wavelet decomposition of series of length 4 [Aggarwal 2013.*]

2.2 Techniques

The difficulty of anomaly detection stems from the fact that the patterns that are searched for are typically not known. Different fields have come up with a large number of different approaches for the detection of the different kinds of anomalies. Two such approaches are sketched below: *proximity-based* techniques, and *information theoretical* approaches [Aggarwal 2013].

2.2.1 Proximity-based Anomaly Detection

By applying certain transformations to segments of a time series, a segment can be mapped into a multidimensional vector space. The proximity can then be calculated for example with the Euclidean distance, which opens up the whole range of proximity-based outlier detection methods, such as cluster, distance, and density based techniques.

The trivial example of such a transformation is to just consider segments of length n as a vector of length n. Other approaches include *Discrete Fourier Transforms* (DFT) or *Discrete Wavelet Transforms* (DWT), the simplest of which is the *Haar Wavelet*. A Haar Wavelet decomposition is sketched in Fig. 2.3, a more detailed description of how wavelet transformations work is given in [Aggarwal 2013]. The advantage of transforming a sequence into a vector of wavelet coefficients is that the coefficients directly represent short-term and long-term dependencies. This enables a reduction of the dimensionality of the space that has to be analyzed depending on the nature of the problem. For example, frequencies above a certain threshold can be ignored. The application of proximity-based methods naturally becomes much more effective on the transformed sequences. Depending on the nature of the dataset different transformations are more effective. For sequences with dominant periodic parts the DFT works better, while series with discontinuities are well represented by the Haar-DWT.

2.2.2 Information Theoretical Approaches

Information theoretical models rely on the creation of so-called *summaries* of a dataset. The length of the summary is shorter the simpler the sequence is. A completely periodic sequence can be described very concisely. Sequences that contain anomalies require a longer description, hence the summary becomes longer. An anomaly is defined as a point or subsequence that leads to a large increase of the length of the summary. In practice, measures such as entropy can take the part of the summary length. The entropy of a sequence X is defined by:

$$H(X) = -\sum_{x \in X} P(x) \log P(x), \tag{2.1}$$

where $P(x)$ is the probability of a value x.

2.2.3 Prediction

The approach that is used in this thesis relies on modelling the normal behaviour of the given dataset and creating predictions. The predictions can then be compared to the actual values of the series, essentially reducing the problem to a *simple anomaly*, which can be detected by a simple threshold on the error.
Considering a time series of length T, the single input *frames* of a series will further be denoted by $\vec{u}_t$ with $t \in [0, T]$. An input frame contains all the features at time t and is represented by a vector with m components, which could be the pixel values of a flattened image. The same holds for the target frames $\vec{d}_t$, which hold the desired output at every time step. The output of the forecasting algorithm is called the prediction, denoted by $\vec{y}_t$, and ought to be as close as possible to the target $\vec{d}_t$. Prediction and target frames can have a different size from the input frames:

$$\begin{aligned} \vec{u}_t &= (u_1^t, u_2^t, ..., u_m^t)^T, \\ \vec{d}_t &= (d_1^t, d_2^t, ..., d_k^t)^T. \\ \vec{y}_t &= (y_1^t, y_2^t, ..., y_k^t)^T. \end{aligned} \tag{2.2}$$

By defining an input sequence $\mathbf{u}$ as a sequence of vectors

$$\mathbf{u} := (\vec{u}_0, \vec{u}_1, \vec{u}_2, ..., \vec{u}_M) \tag{2.3}$$

the prediction problem can be formulated by finding a function F, that returns a good estimate $\mathbf{y}$ of the next N true vectors $\mathbf{d} := (\vec{u}_{M+1}, ..., \vec{u}_{M+N})$.

$$\mathbf{y} = (\vec{y}_{M+1}, ..., \vec{y}_{M+N}) = F(\vec{u}_0, \vec{u}_1, ..., \vec{u}_M) = F(\mathbf{u}) \tag{2.4}$$

The approximation of the function F is a regression problem which has been studied intensively throughout history [Diaconescu 2008]. Recently, Neural Networks (NN) have been applied to anomaly detection, as they can model certain non-linear sequences without a priori knowledge about the data, just by applying a learning algorithm. An in depth explanation of how NNs can be applied to find F for spatio-temporal datasets is given in Chapter 3.

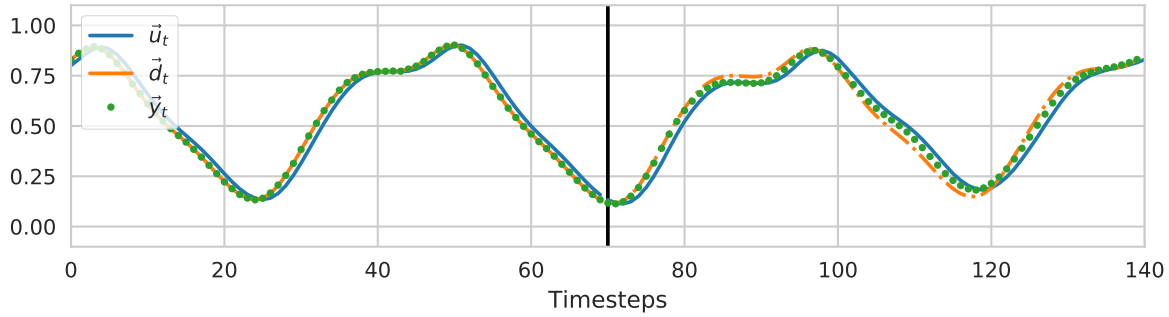


Figure 2.4: *Input $\vec{u}_t$, target $\vec{d}_t$, and prediction $\vec{y}_t$ during the regression phase (left of the black line) and during the prediction phase. In the prediction phase the algorithm has no access to the true target any more, which is why they are depicted by a dashed line.*

Before 1980, time series were typically predicted by using autoregressive, moving-average models (ARMA), which where introduced by Box and Jenkins [Box and Jenkins 1970]. Another method published in 1960 by Winters on forecasting sales improved Holt's double exponential smoothing. His method later became known as the Holt-Winters method [Winters 1960]. Both approaches are described in in the next paragraphs. It should be noted though that they are both linear regression models, which makes them incapable of predicting non-linear time series. For the sake of simplicity they are described only for the case of scalar time series.

During the regression (or training) phase of the forecasting algorithm the target for the prediction is typically the true observation of the next time step $\vec{d}_t = \vec{u}_{t+1}$. The parameters of the algorithm are tuned until the predictions $\vec{y}_t$ are good enough. As soon as the 'one step ahead' prediction problem is solved, predictions further into the future can be made by feeding $\vec{y}_t$ back to the algorithm as the next input $\vec{u}_{t+1}$. In this case the algorithm is in the forecasting phase and is not optimized any more. This prevents information about the next frame to leak into the prediction process.

Auto-regressive Integrated Moving Average

Auto-regressive integrated moving average (ARIMA) models are widely used in forecasting and are typically applied to non-stationary time series. A non-seasonal ARIMA model is defined by three parameters p, q, and d. The first parameter p is the order of the auto-regressive (AR) model, q is the order of the moving-average (MA) model, and d is the number of non-seasonal differences that are needed to obtain a stationary time series. This means that Y_t is the result of applying the sequence difference operator $\Delta y_t = y_{t+1} - y_t$ for d times to y_t:

$$Y_t = \Delta^d y_t = \sum_{k=0}^{d} (-1)^{d-k} \binom{d}{k} y_{t+k} \tag{2.5}$$

With the difference Y_t we can write the general forcasting equation of the ARIMA model:

$$y_t = \mu + \sum_{i=1}^{p} \varphi_i Y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.6}$$

where μ is the mean of the series, φ_i are the p parameters of the AR model, θ_i the parameters of the MA model, and ϵ_i white noise error terms. The parameters of the MA and AR models have to be fit to the data and the order of the difference must chosen a priori.

Holt-Winters Method

The Holt-Winters method belongs to the class of exponential smoothing algorithms. In contrast to MA models that apply the same weights over a window of a time series, exponential smoothing uses exponentially decaying weights back in time.
The most basic form is called simple exponential smoothing (SES), where the level of previous points provides an estimate for the next time step. The method maintains an estimated point $\vec{y}_t$, which is calculated based on previous points and estimates. They are assigned weights, which decrease exponentially going back in time.

$$y_t = \alpha d_t + (1-\alpha) y_{t-1} \tag{2.7}$$

The smoothing parameter α determines exponentially decreasing weights that are applied to previous data points. The smoothing parameter is chosen such that the mean squared error of prediction and data point is minimized.
The Holt-Winters method extends the SES approach from only forecasting the level to smoothing equations for trend b_t and seasonal s_t. It is described by the following four equations:

$$l_t = \alpha(d_t - s_{t-L}) + (1-\alpha)(y_{t-1} + b_{t-1}) \qquad \text{level} \tag{2.8}$$

$$b_t = \beta(l_t - l_{t-1}) + (1-\beta) b_{t-1} \qquad \text{trend} \tag{2.9}$$

$$s_t = \gamma(y_t - l_t) + (1-\gamma) s_{t-L} \qquad \text{seasonal} \tag{2.10}$$

$$y_{t+m} = l_t + m b_t + s_{t-L+1+(m-1) \bmod L} \qquad \text{forecast} \tag{2.11}$$

where L is the length of the seasonal component, α, β, γ are constants that need to be fit to the data and m denotes how far into the future the prediction goes.

2.2.4 Anomaly Score

With the predicted sequence $\mathbf{y}$ it becomes quite simple to detect an outlier, because the problem can be reduced to a *simple anomaly* by calculating the absolute error sequence E.

$$E_i = ||\vec{y}_t - \vec{d}_t||_2 \tag{2.12}$$

The error can now be thresholded to detect an anomaly, but the value of this threshold depends on the specific dataset that is being analyzed. To obtain a probability for how anomalous a point or subsequence is, an *anomaly score* Σ is defined as suggested by [Ahmad et al. 2017]:

$$\Sigma = 1 - \text{erf}\left(\frac{\mu_e - \mu_E}{\sqrt{2}\sigma_E}\right), \tag{2.13}$$

where μ_E and σ_E denote the mean and standard deviation of a window of length N of the error sequence E. The mean μ_e is calculated from a smaller subsequence of E of length

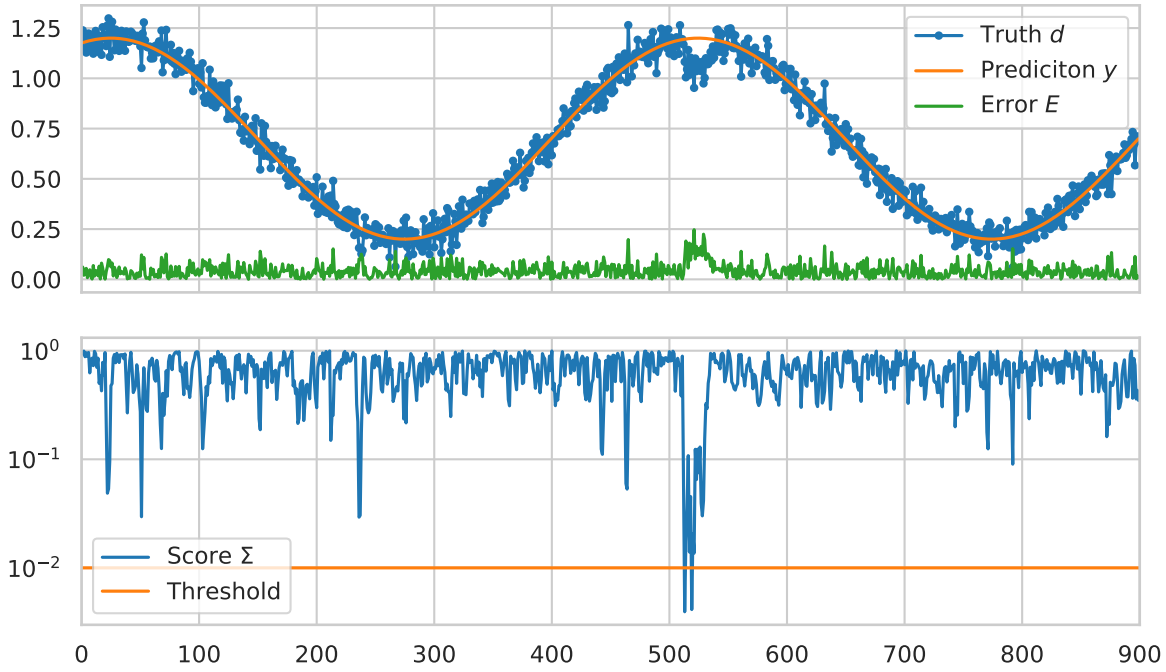


Figure 2.5: *With a good prediction y, the problem can be reduced to a simple anomaly detection over the error sequence E (upper plot). The error sequence still takes on arbitrary values and can be converted into a probability of normality with an anomaly score (lower plot). The normality is calculated with sliding windows of length 100 for μ_E and a length of 5 for m_e.*

$n \ll N$. If μ_e is close to μ_E then Σ will be close to one, and can thus be considered *normal* with a high probability. If μ_e and μ_E are far apart, the probability of normality is low and the point becomes more likely to be an outlier. This results in a score for the whole sequence, which can be used as an intuitive threshold according to the problem at hand. Fig. 2.5 shows how the anomaly detection problem of a contextual outlier is first reduced to a simple anomaly by calculating E and then detected with a threshold on Σ.
If the quantities $\vec{y}_t$ and $\vec{d}_t$ are vectors or matrices it might be desirable to know which components in the vector (matrix) caused the anomaly. For such a spatially resolved anomaly score the norm from Eq. 2.12 can be removed and Σ is calculated component-wise for every coefficient in $|\vec{y}_t - \vec{d}_t|$.

Chapter 3

Neural Networks

In the recent years Neural Networks (NN) have solved more and more tasks that have previously been too difficult or simply too tedious to solve with traditionally coded algorithms. They have been successfully applied to a variety of problems such as pattern recognition, image classification and prediction. As the NN algorithms learn from data, they seem to be good candidates for finding anomalies without any a prior knowledge about the given dataset, as long as it is big enough. By applying online learning algorithms that learn and adapt continuously it should in theory be possible to create an automated, adaptive outlier detection.

This chapter describes the mechanics of NNs from the ground up. Starting with the widely used *feedforward networks* (FNN), we go on to *recurrent neural nets* (RNN). Similar to biological neural networks, they have cyclic connections, and are capable of processing sequences. After that, a special kind of RNN is introduced, that dramatically cuts computational costs and solves some notorious problems that arise during RNN training. In addition to the general difficulty of training NNs, they often rely on hyper-parameters, that are typically set by manually tuning the network. This chapter is closed with a brief description of hyper-parameter optimization techniques which were used to automate this task.

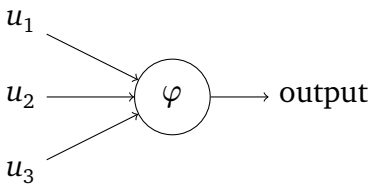


Figure 3.1: *Schematic of a neuron [Nielsen 2015]. The activation function is represented by the circle.*

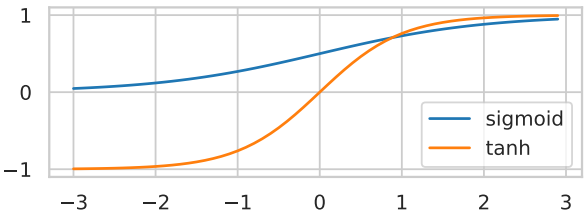


Figure 3.2: *Sigmoid function φ. Typically used as an activation function in feedforward neural nets.*

3.1 Feedforward Neural Networks

The idea for artificial neural networks (ANN) is based on the human brain, which is a highly complex, non-linear, and parallel computer. Just like the brain, ANNs are constructed from small units that are connected to each other with weights. In analogy to its biological counterpart these units are also called *neurons*. Older literature also often refers to them as *perceptrons*. They are capable of processing and passing on incoming information. Traditional feedforward networks implement static input-output mappings, which mathematically makes them pure functions of the input signals. Figure 3.1 shows a simple schematic of a single neuron. It generally has n inputs u_j and one output. Each of the inputs has an assigned weight w_j, which determines the contribution of a given input to the output. The output of a neuron will further be referred to as *activation* y:

$$y = \varphi\left(\sum_j (w_j u_j) + b\right) = \varphi(\vec{w} \cdot \vec{u}_t + b). \tag{3.1}$$

The sum over all the inputs multiplied by their weights can conveniently be represented by a dot product. Often a bias term b is included, which can be taken as a measure of how easy it is to activate the neuron. The function φ is called activation function and can have various different forms, from a simple binary function to an arbitrary (non-linear), monotonically increasing function. A frequently used activation function in FNNs is the sigmoid function

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{3.2}$$

which is shown in Fig. 3.2. To create a *layer* of neurons they are simply stacked on top of each other:

$$\vec{y}_t = \varphi(\mathbf{W}\vec{u}_t + \vec{b}). \tag{3.3}$$

The final FNN with multiple layers (such as in Fig. 3.3) is created by feeding the output of one layer to another one until the last layer of the network is reached. The *universal approximation theorem* [Hornik, Stinchcombe, and White 1989] states the ability of FNNs to approximate an arbitrary non-linear function f

$$\vec{d}_t = f(\vec{u}_t) \tag{3.4}$$

with arbitrary precision. This means that given any $\epsilon > 0$ we can find an FNN F, such that

$$||F(\Theta, \vec{u}_t) - f(\vec{u}_t)||_2 < \epsilon \tag{3.5}$$

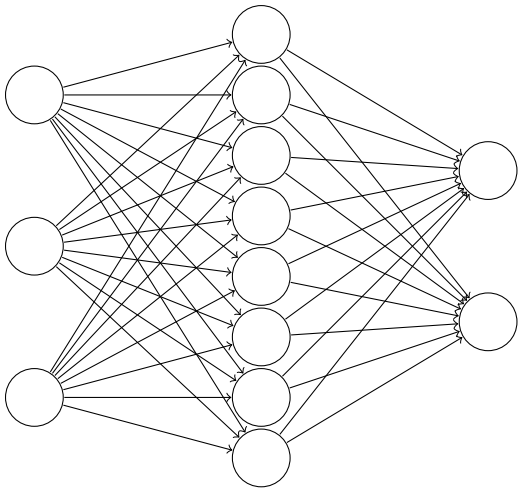


Figure 3.3: *Fully connected feedforward neural network with a single hidden layer and two output neurons.*

for all possible inputs $\vec{u}_t$. The exact form of the approximation F depends on the network architecture, but generally it depends on a set of parameters Θ called weights (and biases), which represent several layers of the FNN. Of course, the universal approximation theorem does not say anything about the practical learnability of a task, which will be discussed in Sec. 3.1.2.

Each layer of an FNN can be represented by a matrix, and therefore only has the computational expressibility of linear functions. This linearity is only broken by the activation function. Without the non-linear activation an arbitrary number of layers would not be more effective than a single one. Through the organization in layers FNNs are able to model not only arbitrary functions but also to separate datasets that are not linearly separable. Fig. 3.3 shows an FNN where the first (input) layer represents the data that is fed into the network, followed by one or more hidden layers, and an output layer. Hidden layers act as a non-linear transform that distorts the input in such a way that its classes become linearly separable by the output layer. An interactive explanation of how this works can be found on [Olah 2014].

Every network that has more than two or three hidden layers is typically called a *deep neural network*. It is fundamentally not different from the basic architecture of the described feedforward networks but holds the potential of more powerful transformations of the input. The type of neural network that is shown in Fig. 3.3 is called *feedforward network*, because the input data is entering the network at the input layer and then passed through the network towards the output layer. Feedforward nets are often applied to classification tasks, such as the recognition of a certain shape in an image.

The goal of the machine learning approach is to find a weight configuration that captures the *essence* of the presented dataset, meaning that it generalizes well, including over inputs that it has not been trained with. This is typically approached by some variation of Gradient Descent (GD). GD algorithms try to minimize a certain loss or cost function with respect to a given weight configuration (more detailed description in Sec. 3.1.2). However, the pursued generalization of the network is highly dependent on the training dataset. It should ideally

include the total variability of possible inputs. This is very hard to achieve in practice and the available datasets are typically split randomly into three categories: *training*, *validation*, and *test* set. The training set is used to optimize the network. Its loss is minimized by a GD algorithm. The validation set is used to determine the performance of the network on inputs that are not included in the training set. The optimization should be stopped as soon as the error on the validation set is not decreasing any more in order to reduce the risk of overfitting. This is one of many regularization methods that are applied in ML in order to achieve a generalization over previously unseen datapoints. Because the information of the validation set is leaking into the network via the early stopping criterion, a third dataset is needed to evaluate the actual performance and generalization of the network. This third set is called the test set.

Feedforward networks outperform conventional algorithms at tasks such as image recognition and other classification and pattern recognition tasks. They are, however, not suitable to model time series, as they cannot model correlations of previous inputs. This means that they are not suitable for the prediction of sequences. To be able to process sequences and make predictions, *recurrent* weight connections are introduced in Sec. 3.2. Despite the compelling results that can be achieved by FNNs, they have a few drawbacks. Backpropagation algorithms typically need very large amounts of data to train the network. The gradient descent steps have to be small enough to not jump over the desired minima, which leads to very long training times. Additionally the nature of the training data can lead to biases of the resulting network if it does not properly represent the possible parameter space. It is hard to infer afterwards how the network came to a specific result, as the weight matrices do not represent a traceable way of reasoning or logic. A trained network is like a black box, which does not come with an obvious way of determining how or why a certain classification was made.

3.1.1 Convolutional Neural Nets

A breakthrough in image classification performance was achieved by the introduction of a subtype of FNNs, the *convolutional neural networks* (CNN), which can leverage the spatial structure of the input data. They combine the concept of filters from signal processing with the ML approach by learning their own filters (often referred to as kernels as well) for feature detection in images. The convolutional operation of a given filter F and an image H is defined by:

$$G = H * F, \text{ where} \tag{3.6}$$

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[i-u, j-v]F[u,v] \tag{3.7}$$

A well known example of a filter for edge detection looks like this:

$$F = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{3.8}$$

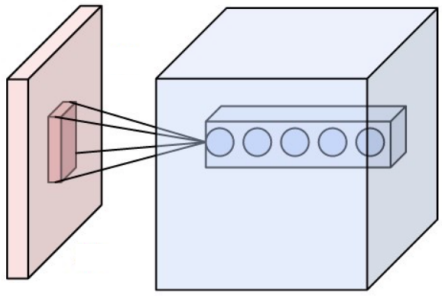


Figure 3.4: *The 2D input image on the right is transformed into a 3D output volume by applying multiple convolutions to the input [Wikipedia 2018.*
]

Applying it to a (black and white) image H

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.9}$$

and padding the convoluted image G to obtain the original dimensions results in a filtered image R, which is zero everywhere except at the location of the edge:

$$R = \begin{bmatrix} - & - & - & - & - & - & - & - & - \\ - & 0 & 0 & 0 & 3 & -3 & 0 & 0 & - \\ - & 0 & 0 & 0 & 3 & -3 & 0 & 0 & - \\ - & 0 & 0 & 0 & 3 & -3 & 0 & 0 & - \\ - & 0 & 0 & 0 & 3 & -3 & 0 & 0 & - \\ - & - & - & - & - & - & - & - & - \end{bmatrix} \tag{3.10}$$

A convolutional layer consists of multiple kernels, which are learned via GD. Each kernel can be thought of as a neuron that sees only a small part of the input image at a time and slides over the whole image. Its output therefore represents the local structure of said part of the image, which makes a CNN capable of exploiting spatial structures in images (see schematic in Fig. 3.4). The kernels which are sliding over the input images naturally lead to a translation invariance of the learned patterns [LeCun, Bengio, et al. 1995]. A pattern that is recognized by a filter in one region of the image can just as well be detected somewhere else, which stands in contrast to fully connected layers for which the same pattern in different parts of an image looks completely different. Scale invariance can be achieved by using multiple kernels of different size. Feedforward nets, which are fully connected to the input cannot leverage the spatial information as efficiently. Additionally, CNNs decrease the number of parameters in comparison to a normal FNN, which reduces the risk of overfitting and thus leads to a better generalization of NNs.
An interesting example of a CNN outside the realm of image classification is AtomNet [Dzamba 2015], which is used to predict bioactivity of molecules for drug discovery by exploiting the local structure of biochemical interactions.

3.1.2 Gradient Descent

The most common technique to train neural networks is Gradient Descent. It is based on minimizing a *loss* (cost or penalty) function $\mathscr{L}$

$$\mathscr{L}(\Theta) = \sum_{\vec{u}_t \in \mathscr{U}} ||F(\Theta, \vec{u}_t) - \vec{d}_t||_2, \tag{3.11}$$

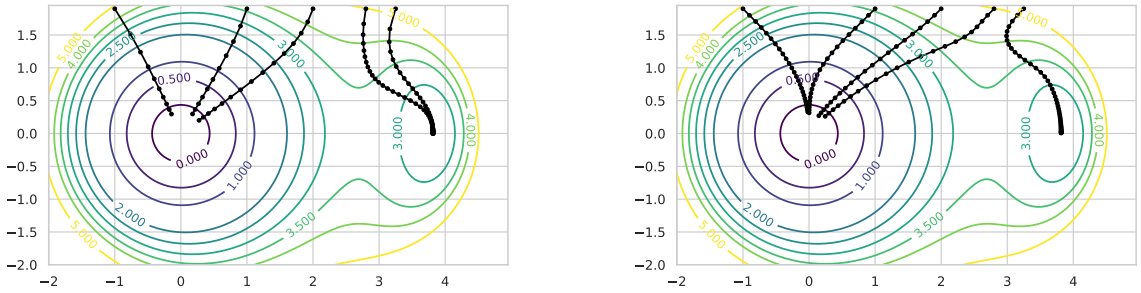
which defines how close the network is to producing the desired results. The loss function that is used throughout this thesis is given by Eq. 3.11, where the target outputs, given a certain input $\vec{u}$ out of all training examples $\mathscr{U}$, are denoted by $\vec{d}_t$.
Initially, the weights of the network are set randomly and are to be adjusted in the optimization phase, also called learning or training. The loss is, of course, dependent on all the weights and biases, which is where the gradient descent comes into play. The partial derivatives of the loss function are taken with respect to all the weight and biases of the network to find the gradient which points in the direction of steepest descent. With this calculated direction we can step the weights towards the nearest local minimum and like that gradually increase the performance of the network.

$$\Theta' = \Theta + \eta \frac{\partial \mathscr{L}}{\partial \Theta} \tag{3.12}$$

The size of the steps is defined by the *learning rate* η, which has to be chosen carefully. If it is too large, the algorithm will oscillate around the desired minimum, but if chosen too small, the training times will become too long. Several adaptive gradient computation algorithms address this issue. One promising algorithm is called *Adam* (Adaptive Moment Estimation) [Kingma and Ba 2014]. Adam combines adaptive gradient descent methods with momentum based algorithms [Kingma and Ba 2014]. Momentum-based optimizers add a fraction of the previously used weight update to simulate an acceleration of the gradient descent. The method of applying the gradient descent algorithm to multi-layer (deep) neural networks is called *backpropagation* (BP), because the gradient calculation is started at the last layer and iteratively propagated back through the whole network by applying the chain rule. A more detailed description of BP is given in Sec. 3.1.3.
There are three different variations of gradient descent, which only differ in the way the sum in Eq. 3.11 is interpreted. Summing over all available training examples $\vec{u}_t$, namely the whole *batch*, is called *batch GD*. Calculating the loss only for a single randomly chosen example, is called *stochastic gradient descent* (SGD). The compromise of the two, mini-batch GD, uses a subset of the training examples, performs a weight update, and then goes on to the next mini-batch. Both stochastic and mini-batch GD end up calculating an approximate loss L from a subset $U \subset \mathscr{U}$:

$$\mathscr{L}(\Theta) \approx L(\Theta) = \sum_{\vec{u}_t \in U} ||F(\Theta, \vec{u}_t) - \vec{d}_t||_2, \tag{3.13}$$

which also leads to an approximated gradient. The updates that are performed with the approximate gradient are hoped to enable the optimizer to jump out of shallow local minima and saddle points, which is discussed further in the next paragraph.



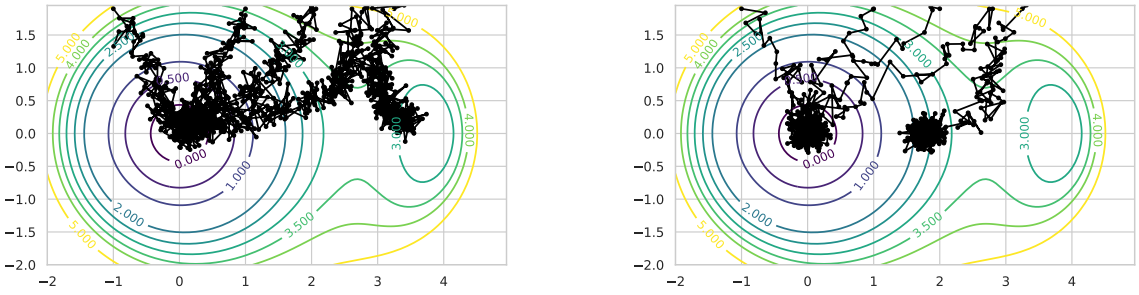
GD Adam

Figure 3.5: *On the left we can see a plain GD optimizer while on the right the Adam optimizer was used. Depending on the starting values of the two variables, there are cases in which the optimization ends in the local minimum on the right. The GD optimizer needs fewer steps, while Adam finds the global minimum in one more case.*

Local Minima of the Error Surface

As BP is a gradient-based method, there is no guarantee at all, that the algorithm will find the global minimum of the error surface. To illustrate this, Fig. 3.5 depicts a two dimensional error surface with a global and a local minimum. Depending on where on the surface the optimization is started, it ends up in the local or global minimum. It was shown that for linear activation functions, the error surface contains only a single minimum, the global one, with all other locations of zero gradient being saddle points [Baldi and Hornik 1989]. Momentum-based optimizers can find the global minimum quite efficiently in such cases. However, this cannot be generalized to the practically used NNs that almost exclusively use non-linear activations. Optimizers that include momentum like the Adam optimizer can still sometimes yield better results, as depicted in Fig. 3.5. The two plots both show a two dimensional error surface, where the coordinates x and y are regarded as the weights that have to be optimized. The black, dotted lines show different runs with varying initial values of x and y. The left plot shows the convergence paths of plain batch GD algorithm as described above, which converges towards the global minimum in three out of five cases. As GD is a purely gradient based method, the path always advances in the direction of steepest descent. With the more advanced Adam optimizer, the global minimum is found in four out of five cases. The momentum towards the global minimum that is gained in the beginning carries the optimization away from the local minimum. Of course there are numerous cases where this approach will not yield a global minimum. Additionally the Adam algorithm is much more computationally expensive than plain GD and needs more steps, as can be seen from the denser dots on the black lines.

To date, the most commonly used technique to encourage GD to converge to the global minimum is mini-batch gradient descent. Instead of evaluating the *true* loss of the whole training set as indicated by Eq. 3.11, SGD adapts the weights after every mini-batch X. This leads to an approximation of the gradient and causes oscillations in the convergence path, which should make it more probable to escape local minima. Fig. 3.6 again compares plain GD and Adam, but now with the perturbing effect of evaluating an approximate loss. The number of training steps that have to be taken increase significantly, but plain SGD is able to



stochastic GD stochastic Adam

Figure 3.6: *The plot on the left shows the convergence paths of stochastic plain GD, which find the global minimum in on more case compared to batch GD.*

find the global minimum in four out of five cases now. Stochastic Adam almost reaches the minimum in all five cases, but seems just get stuck right before the global minimum in the last case. The conclusion we can draw from the above examples is that care has to be taken with respect to the convergence of NNs. Different optimizers, gradient descent variations, and initializations can yield vastly different results. However, experience has shown that in most pattern recognition tasks, SGD algorithms can find a local minimum that yields satisfactory performance, even though it is not guaranteed that this minimum is global. In most applications, an increase in the network connections is supposed to create paths around suboptimal local minima [Rumelhart, Hinton, and R. J. Williams 1986]. Additionally, in most application the training is actually stopped early, meaning at the time the validation error does not decrease further. This is done because a network that is fitted perfectly to the training data would most probably not generalize well over new inputs. In most cases it is therefore acceptable or even desirable to remain in a *good* local minimum that results in a good generalization of the network.

3.1.3 Backpropagation

Backpropagation is the basic algorithm that carries out the GD minimizations that were introduced previously. It was invented in the 1970s but not widely used until the paper by [Rumelhart, Hinton, and R. J. Williams 1986], which marks a breakthrough in the field of machine learning. For an easy to read but in depth description of the subtleties of BP the reader is referred to the book *Neural Networks and Deep Learning* by [Nielsen 2015].
Fundamentally, BP is nothing more than the application of the chain rule to the cost function of a neural network, which can be solved by an *automatic differentiation* (AD) algorithm. In fact, BP is a special case of AD called *reverse mode* AD [Baydin et al. 2018]. To illustrate the BP algorithm we will consider a network with L feedforward layers, where the activations of the first layer are just the inputs to the network ($\vec{u} = \vec{y}_1$) and the last layer contains the network outputs $\vec{y}_L$. The components y_l^j of the vector $\vec{y}_l$ that contains all activations of a layer l are calculated based on the previous layer:

$$\begin{aligned} z_l^j &= \sum_k w_l^{jk} y_{l-1}^k + b_L^k \\ y_l^j &= \varphi(z_l^k), \end{aligned} \tag{3.14}$$

where $\vec{z}_l$ are called the *weighted input*. The system above may be altered such that a given activation depends on any *previous* layer activation, but *not* on activations of any *next* layer. This step is named *forward pass*. During the forward pass, all activations are calculated starting from the first layer. The goal of the *backward pass* is now to adjust all the weights w_l^{jk} (and biases b_L^k) such that the loss L

$$L = \frac{1}{2}\sum_k (d^k - y_L^k)^2 \tag{3.15}$$

of a single input example $\vec{u}$ is minimized. BP solves this by iteratively calculating the gradients needed for Eq. 3.12 starting from the last layer. Before directly calculating the necessary gradients it is easier to compute the *error* δ_L of the weighted inputs for each unit j of the last layer:

$$\delta_L^j = \frac{\partial L}{\partial z_L^j} = \frac{1}{2}\sum_k \frac{\partial}{\partial z_L^j}(d^k - y_L^k)^2 = \frac{\partial L}{\partial y_L^j}\frac{\partial y_L^j}{\partial z_L^j} = (d^j - y_L^j)\varphi'(z_L^j), \tag{3.16}$$

where the sum vanishes because y_L^k only depends on z_L^j if $j = k$. Now the error of any layer l can be expressed in terms of the error of the next layer $l+1$:

$$\delta_l^j = \frac{\partial L}{\partial z_l^j} = \sum_k \frac{\partial L}{\partial z_{l+1}^j}\frac{\partial z_{l+1}^j}{\partial z_l^j} = \sum_k \delta_{l+1}^j \frac{\partial z_{l+1}^j}{\partial z_l^j} = \sum_k \delta_{l+1}^j w_{l+1}^{kj}\varphi'(z_l^j) \tag{3.17}$$

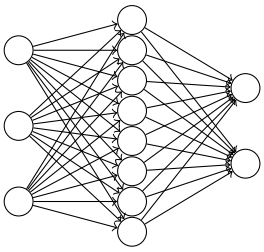
Now the desired error gradients can be found to be:

$$\begin{aligned} \frac{\partial L}{\partial b_l^j} &= \delta_l^j, \\ \frac{\partial L}{\partial w_l^{jk}} &= y_{l-1}^j \delta_l^j \end{aligned} \tag{3.18}$$

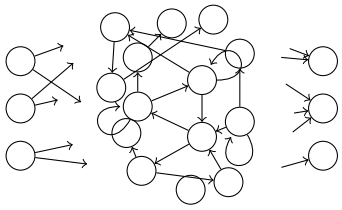
They can now be used to incrementally update the weights and biases according to Eq. 3.12.

The Backpropagation Algorithm

1. *Input*: $\vec{u}$ is the activation vector of the first layer $\vec{y}_1$
2. *Forward pass*: Compute activations $\vec{y}_l$ for each $l = 2, 3, ..., L$ (Eq. 3.14)
3. *Output error*: Compute error δ_L of last layer (Eq. 3.16)
4. *Backpropagate*: Compute δ_{l-1} based on error of layer l (Eq. 3.17)
5. *Output*: Obtain the gradients of the cost function (Eq. 3.18)



Feedforward Network



Recurrent Network

Figure 3.7: *The traditional feedforward network on the left only exhibits forward connections while the recurrent network on the right has cyclic connections within or possibly even between layers if there is more than one hidden weight matrix.*

3.2 Recurrent Neural Networks

The two major types of neural networks are distinguished by the structure of their internal weights. Feedforward networks simply pipe their input through all the layers towards the output. They have proven very useful for tasks that require static, non-linear input-output mappings such as pattern recognition or classification. Processing time series is a different task because, to model a sequence, the network needs information about the past. In a FNN this is not the case. An input vector $\vec{u}_t$ that is fed into the network F does not contain information about previous or future inputs and the prediction $\vec{y}_t$ has to be made solely based on the current input:

$$\vec{y}_t = F(\vec{u}_t) \tag{3.19}$$

In contrast, recurrent neural networks (RNN) possess cyclic weight connections. The outputs of a layer have feedback weights that are connected to the same or a previous layer. This cyclic nature of RNNs mathematically makes them dynamical systems [Funahashi and Nakamura 1993] and it can be shown that they are *Turing complete* [Siegelmann and Sontag 1991]. A brief description on the analogies between RNNs and dynamical systems is given in Sec. 3.2.1. Roughly, RNNs maintain an internal state $\vec{x}_t$ at all times which acts as a memory of previous inputs. At every time step the network receives the previous internal state along with an input vector and returns a prediction $\vec{y}_t$ as well as an updated internal state $\vec{x}_t$:

$$\vec{y}_t,\ \vec{x}_t = F(\vec{u}_t, \vec{x}_{t-1}). \tag{3.20}$$

This enables RNNs to process time series data and they are thus qualified for tasks such as filtering, dynamic pattern recognition, and prediction. RNNs are most widely used in speech recognition or other language processing tasks, but they are also highly interesting from a neurological point of view, as all biological neural networks are recurrent. Generally, they are highly promising tools for non-linear time series modelling. They can be run in a self-monitoring fashion, which makes them very interesting for an automated outlier detection in large scale time series data. Especially in the field of Natural Language Processing (NLP), RNNs have achieved impressive results while requiring little preprocessing of datasets [Sutskever, Martens, and Hinton 2011].

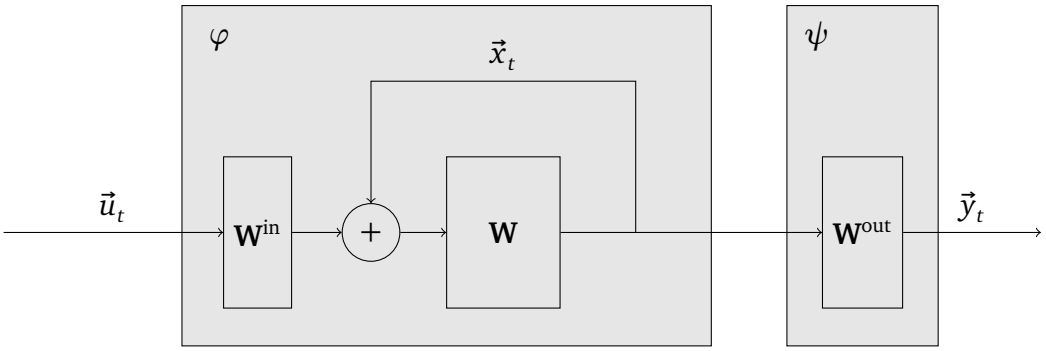


Figure 3.8: *Basic RNN cell flow chart. [... make the interal state arrows thicker ...]*

3.2.1 State Space Model

The simplest dynamical system in discrete time is defined by a state space M, a set of times T, and an evolution function Φ. The function Φ maps a given state $\vec{x}_{t-1} \in M$ at time $t \in T$ to a new state $\vec{x}_t$:

$$\vec{x}_t = \Phi(\vec{x}_{t-1}). \tag{3.21}$$

A dynamical system with inputs $\vec{u}_t$ and outputs $\vec{y}_t$ is defined by the state space representation:

$$\vec{x}_t = \Phi(\vec{u}_t, \vec{x}_{t-1}), \tag{3.22}$$
$$\vec{y}_t = \Psi(\vec{u}_t, \vec{x}_{t-1}). \tag{3.23}$$

In a basic RNN the two functions Φ and Ψ are defined with an input matrix $\mathbf{W}^{\text{in}}$, a recurrent weight matrix $\mathbf{W}$, and an output matrix $\mathbf{W}^{\text{out}}$:

$$\vec{x}_t = \varphi(\mathbf{W}^{\text{in}}\vec{u}_t + \mathbf{W}\vec{x}_{t-1}), \tag{3.24}$$
$$\vec{y}_t = \psi(\mathbf{W}^{\text{out}}\vec{x}_t), \tag{3.25}$$

where φ denotes the component-wise applied, non-linear, state activation function. In RNNs a typical choice is the hyperbolic tangent. The output activation function ψ is commonly chosen to be the identity function, resulting in a linear output layer. A flow chart of the basic RNN is shown in Fig. 3.8. The input weights $\mathbf{W}^{\text{in}}$ have dimensions $n \times m$, hidden weights $\mathbf{W}$: $n \times n$ and output weights $\mathbf{W}^{\text{out}}$: $n \times k$. In the simple RNN the input is not utilized by the output layer, but would be entirely possible to introduce another matrix to do this.
An input series $\mathbf{u}$ of length N that is fed to the network one by one produces N internal states.

$$\mathbf{x} = (\vec{x}_t, \vec{x}_{t+1}, ..., \vec{x}_{t+N}), \tag{3.26}$$

From Eq. 3.24, it becomes evident that the internal state acts as a kind of memory of the network. This memory is dynamic as opposed to the static memory brought about by weight adjustments of GD. The latter is called long-term memory. The dynamic memory of the internal RNN state is termed *short-term memory* (STM). STM will be further discussed in Sec. 3.3.5 and brief computational analysis of the STM capacity is given in Sec. 5.1.

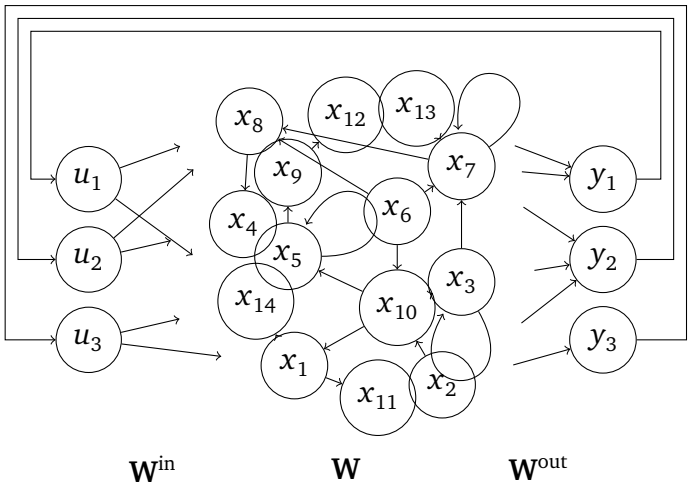


Figure 3.9: *RNN setup that is able to predict n steps into the future by feeding the output back into the input. The input and output layers are fully connected. The internal weight connections can be sparse to speed up the network for large reservoir sizes.*

Every new input overwrites a part of the previous internal state, gradually encoding the input sequence into $\vec{x}_t$. The length of an input sequence that can be encoded into $\vec{x}_t$ depends on the size of the internal state and on the two matrices $\mathbf{W}^{\text{in}}$ and $\mathbf{W}$. Generally, the state size n must be much larger than the input size m, in order to create an effective RNN. How an RNN is trained in the light of a time dependency of the weights is described in Sec. 3.2.2.

As the internal states now contain information both about current and past inputs, it is possible for the output matrix $\mathbf{W}^{\text{out}}$ to create an educated prediction of the next frame. An RNN that receives its output as the next input is called a freely running RNN and enables predictions further into the future as depicted in Fig. 3.9. Of course, in this case the number of input and output units must be the same: $k = m$.

3.2.2 Training Recurrent Neural Networks

There exist various different methods of training recurrent networks, which all have their own benefits and drawbacks or just perform better or worse at different tasks. No clear favourable approach, like the mini-batch GD algorithm for feedfoward networks, was found yet. This is due to several obstacles that arise specifically during RNN training, which will be discussed below. The three most common methods are *Backpropagation Through Time* (BPTT), *Real-time Recurrent Learning* (RTRL, [R. J. Williams and Zipser 1989]), and *Extended Kalman Filtering* (EKF, [R. J. Williams 1992]), the first of which will be treated below.

Backpropagation Through Time

BPTT is an adapted form the of classic backpropagation algorithm of feedforward networks as described in Sec. 3.1 and was developed for the first time by Mozer [Mozer 1995]. The cyclic connections of RNNs prevent the direct application of the backpropagation algorithm. One solution is to *unroll* the network in time by stacking the network on top of itself for a

certain number of time steps. The depth of unrolling is determined by the length N of the sequence $\mathbf{u}$ that is fed into the network.
By unrolling the network in time, one practically ends up with a very deep feedforward network with shared weights between the stacked layers of clones of the network. In the forward pass each clone, which now corresponds to a time step in the sequence, receives the corresponding input $\vec{u}_t$ and updates its own internal state $\vec{x}_t$. The internal state of each clone depends on its input and on the internal state of the previous layer (at time $t-1$). Finally the output $\vec{y}_t$ is computed by each clone. The loss function that is minimized is defined by:

$$\mathscr{L} = \sum_{t=0}^{N} \mathscr{L}_t = \sum_{t=0}^{N} ||\vec{d}_t - \vec{y}_t||_2. \tag{3.27}$$

The weight adjustment is now done by a typical gradient descent algorithm. By collecting all the weights and biases of the state space model in the variable Θ we can write the weight adjustment as:

$$\Theta' = \Theta + \eta \sum_t \frac{\partial \mathscr{L}_t}{\partial \Theta} \tag{3.28}$$

The expression for the gradient of the cost can by derived by applying the chain rule:

$$\frac{\partial \mathscr{L}_t}{\partial \Theta} = \frac{\partial \mathscr{L}_t}{\partial \vec{x}_t} \frac{\partial \vec{x}_t}{\partial \Theta} = \frac{\partial \mathscr{L}_t}{\partial \vec{x}_t} \frac{\partial \vec{x}_t}{\partial \vec{x}_{t-1}} \frac{\partial \vec{x}_{t-1}}{\partial \Theta}, \tag{3.29}$$

which results in a product of derivatives, as every state depends on the previous one.

$$\frac{\partial \mathscr{L}_t}{\partial \Theta} = \frac{\partial \mathscr{L}_t}{\partial \vec{x}_t} \frac{\partial \vec{x}_t}{\partial \vec{x}_0} \frac{\partial \vec{x}_0}{\partial \Theta} \tag{3.30}$$

$$\frac{\partial \vec{x}_t}{\partial \vec{x}_0} = \prod_{i=1}^{t} \frac{\partial \vec{x}_i}{\partial \vec{x}_{i-1}} \tag{3.31}$$

The last derivative of the state $\vec{x}_0$ denotes the derivative of the first state (starting to count from the perspective of the forward pass) of the unrolled network, which is a constant with respect to Θ. From Eq. 3.31 one can see the origin of the vanishing and exploding gradient problems. If the individual derivatives $\frac{\partial \vec{x}_t}{\partial \vec{x}_0}$ are small, the gradient, being product of these derivatives, vanishes very quickly and explodes if they are large. It can be shown that it '[...] is *sufficient* for the largest eigenvalue λ_1 of the recurrent weight matrix to be smaller than one for long term components to vanish (as $t \to \infty$) and *necessary* for it to be larger than one for gradients to explode [Pascanu, Mikolov, and Bengio 2012]. By bounding the spectral radius of the recurrent weights to be smaller than one, it is thus possible to avoid the exploding gradient problem. A solution to the vanishing gradient problem is more complicated and involves advanced network architectures such as the long short-term memory (LSTM) unit [**lstm**]. It introduces additional input, forget and output layers, but the essential part is that the recurrent map of the unit is the identity function. The derivatives in Eq. 3.31 become one and the gradient can flow through many layers. A completely different approach is to avoid training the recurrent weights altogether, which will be described in Sec. 3.3.

3.2.3 The Bifurcating State Space

Another problem that arises with the optimization of recurrent weights is that the state space is not necessarily continuous, which was shown by [Doya 1992]. The points at which the state space has discontinuities are called *bifurcations* and are extensively studied in non-linear dynamics. They can cause discontinuities in the state space and thus impair the learning or prevent convergence to a local minimum completely. To understand what bifurcations are and how they affect RNN training, we will consider the recurrent part of a single unit RNN with the hyperbolic tangent as the activation function. If the RNN has only one unit the state $\vec{x}_t$, as well as the weights and biases become a scalars:

$$x_{t+1} = \tanh(wx_t + b). \tag{3.32}$$

The parameter w denotes the scalar weight of the single unit and $b = w_{in}u_t$ will serve as the bias of a constant input of $u_t = 1$. In Fig. 3.10 we can see the evolution of x_t. Depending on different initial values x_0 and network parameters, the state converges to different values for t towards infinity. These values are called *fixed points* x^* and for them $x_t = x_{t+1}$ holds. In particular, fixed points that the state converges to are called *stable* fixed points (or *attractors*). The second kind of fixed points are *unstable*. The slightest deviation from an unstable fixed point will result in a flow away from the point, which is why they are also called *repellers*. In the first three cases of Fig. 3.10 a fixed point is always reached. The fourth example in the lower right shows representatives of the oscillating fixed point, more specifically *period-2 cycles*, that repeat every second iteration.
By varying the parameters w and b the location and the nature of fixed points can be changed. The blue line in the right plot of Fig. 3.12 splits in two as w is increased. The point at $w = 1$ is called a *bifurcation* point. There are two things that are happening here: the stable fixed point at $x = 0$ becomes unstable (indicated by the dashed line) and two new stable fixed points above and below zero are created.
A mathematical analysis of fixed points can be done by assuming that x^* is a fixed point we can analyze Eq. 3.32:

$$x^* = \tanh(wx^* + b). \tag{3.33}$$

Solving once for w and once for b results in two equations for fixed points:

$$b = \tanh^{-1}(x) - wx \tag{3.34}$$

$$w = \frac{\tanh^{-1}(x) - b}{x}, \tag{3.35}$$

which can be plotted for different values of w and b (Fig. 3.12). The period-2 cycles cannot be found by analysing Eq. 3.33. Instead they can be found analytically by solving

$$x^* = \tanh^2(wx^* + b), \tag{3.36}$$

but also by an intuitive, graphical approach called *cobwebbing* (Fig. 3.11). Starting from an initial point x_0 a vertical line is drawn to the value of the activation function. Now drawing a horizontal line until we intersect with the graph of $y = x$ gives the new input x_1 and so forth.

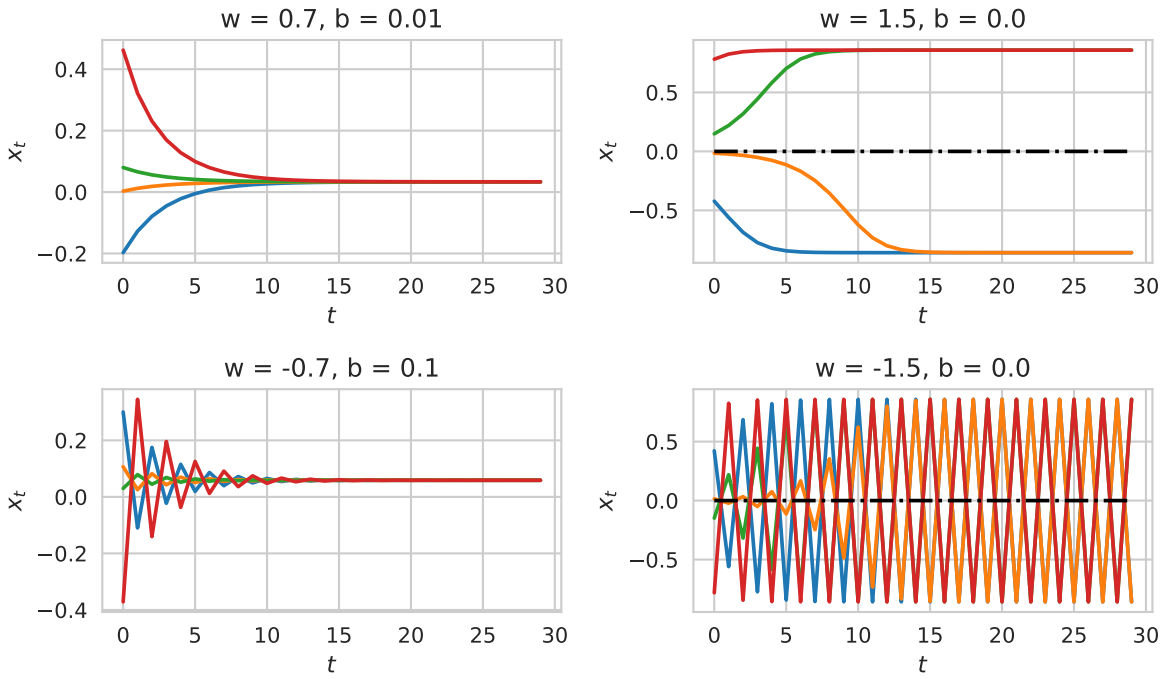


Figure 3.10: *Evolution of x_t over time for different parameters w and b. Dashed lines show unstable fixed points. Apart from the expected fixed points that x_t converges to over time, there are also oscillations visible in the last plot. Such oscillations that repeat every 2 iterations are called period-2 cycles and they appear when $w < -1$.*

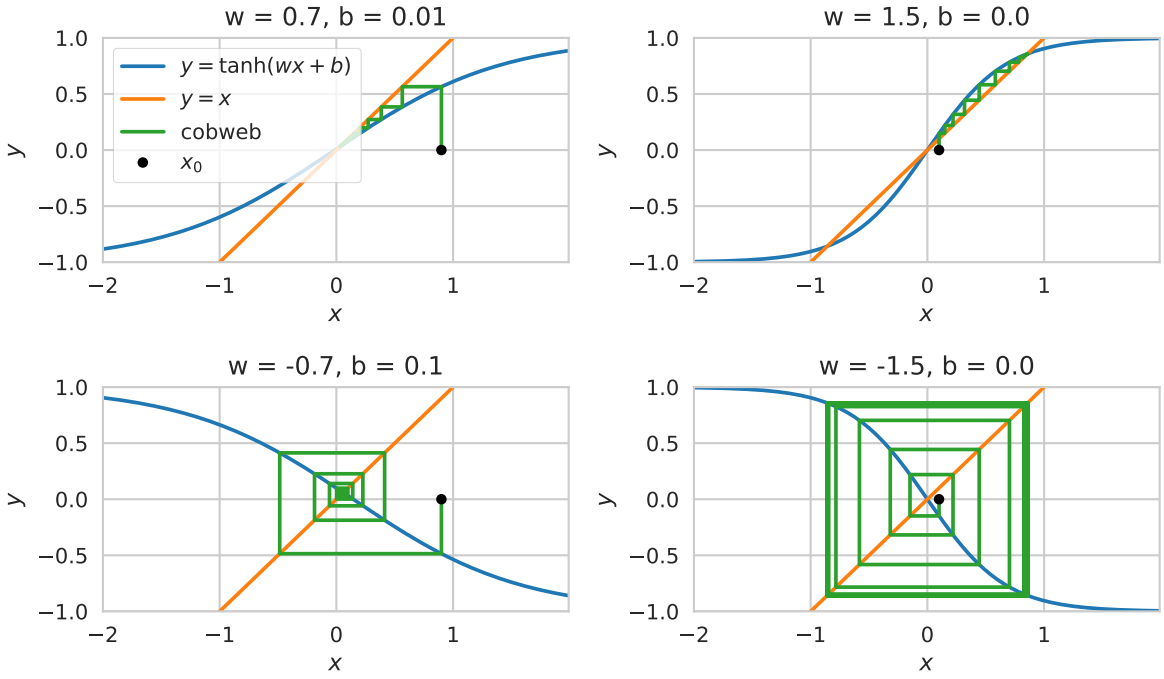


Figure 3.11: *Cobwebs for the same parameters as in Fig. 3.10. The black dot is the initial value x_0. By drawing a vertical line to the intersection with the activation function gives the new input x_1. Drawing a horizontal line to the intersection with $y = x$ projects the point back to the x-axis. The projection is the next input to the activation function. This process is repeated until a stable orbit or a fixed point is reached.*

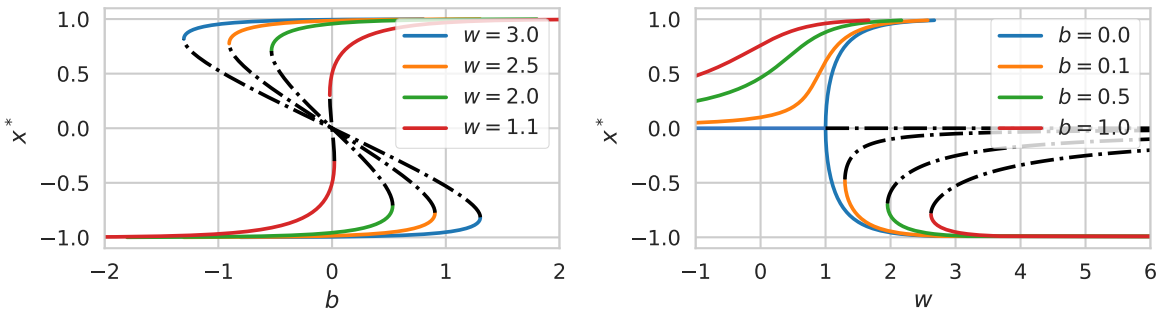


Figure 3.12: *Fixed points for different parameter values of b and w. The values of the fixed points x^* are affected by varying the weights of the RNN. If there is more than one stable fixed point x_t converges to the attractor that is closest to the initial value x_0. Dashed lines denote unstable fixed points, which can only be reached if $x_0 = x^*$.*

Effect on RNN Training

Now that we have an understanding of what fixed points and bifurcations are we can examine their effect on RNN learning. Suppose we initialize the network with a constant $b = 0.1$ and a $w = 3$. If x_0 is negative, the nearest fixed point is on the lower branch of the yellow line in the right plot of Fig. 3.12. Further assume we train the network to output $x_\infty = -0.25$. In this case, w will be lowered to approach $x^* = -0.25$ until the bifurcation point is reached and the stable fixed point vanishes (yellow line becomes dashed line). The fixed point becomes unstable and the network output will change discontinuously as it jumps to the attractor on the upper branch. This will result in a discontinuity in the loss function and an infinite gradient. After jumping to the upper branch w will grow towards infinite values as the GD algorithm tries to approach the target value of $x = -0.25$. Similar examples can be constructed in which parameters oscillate between two bifurcation points. The weights of RNNs are normally initialized to very small values which results in few fixed points. As the network learns some of the weights increase which drives the RNN through bifurcations. The discontinuities that result in very large gradients cause large jumps of the GD algorithm which can nullify the learning of hundreds of steps in a single iteration. Aside from the vanishing and exploding gradient problems, bifurcations are another major reason for the intricacy of RNN training.

One solution to all the headaches that are caused by the bifurcations in the state space, and RNN training in general, is surprisingly simple. The cause of bifurcations are adaptions of the recurrent weights of the RNN, so keeping them constant would eliminate all the complications at once and even come with the additional advantage of not having to change those weights in the first place. This might seem like a rather drastic method as the whole ML approach is based on gradual learning of weights. However, restricting the weight optimization to the non-recurrent weights frees us from the notorious problems of RNNs and can perform just as well. This approach is called *reservoir computing* and will be discussed in the next Section.

3.3 Reservoir Computing

From the description of the available training methods of RNNs in Section 3.2.2, one can see that despite their theoretical capabilities it becomes very difficult to train them in practice:

1. Sequences that require long-range memory are almost impossible to learn, due to the vanishing gradient problem of BPTT based optimizers.

2. The computational complexity of RNN training is very high, with BPTT having a complexity of $O(n^2)$ per weight update per time step for a single output and even $O(n^4)$ for RTRL.

3. Bifurcations in the error surface can prevent the network from converging at all.

A recent approach termed *reservoir computing* (RC) that was introduced by [Lukosevicius and Jaeger 2009] promises to eliminate almost all the previous disadvantages by making a conceptual separation between the actual RNN and the subsequent output layer that maps its internal state to the desired output. The recurrent part of RNN is regarded as a reservoir, which remains untrained and only takes the role of mapping the input into a higher dimensional, temporal representation. A linear output layer is the only part of the network that is optimized to produce the desired output from the internal state of the RNN.

3.3.1 Pattern Recognition in a Bucket

The separation of the recurrent part of the RNN and the output layer is intuitively described by an experiment called *Pattern recognition in a bucket* [Fernando and Sojakka 2003], which basically substitutes the RNN with a bucket of water. A loudspeaker excites waves on the water surface, which is filmed and fed into a simple, one-layer feedforward network, equivalent to the output layer of an RNN. The output layer is trained on the patterns that arise from the audio input and a second control network directly on the audio signal. The control network could not achieve error rates in the classification of simple words below 25%, while the networks trained on the water patters made no more mistakes than 1.5%. The reason for this considerable increase in accuracy is that the water patterns create a high dimensional representation of the input data. This representation does not only hold information of the current time frame, but serves as a memory, as the disturbances in the water propagate over the whole surface. Most importantly, this representation is generated by the underlying physics, which cannot be adjusted to the input data. The water has not undergone any kind of training, however, one can still train the output layer on the deterministically created patterns of its surface, which nicely illustrates the philosophy of reservoir computing. The recurrent weight matrices can, if appropriately initialized, give rise to similar deterministic patterns in the internal state.

3.3.2 The Echo State Network

Mathematically, the part of the static reservoir is taken by the state-space equation (Eq. 3.24) that was introduced in Sec. 3.2.1. The weight matrices are initialized during the setup of the network and are kept static for all times. Only the weights of the output layer (Eq. 3.25) are

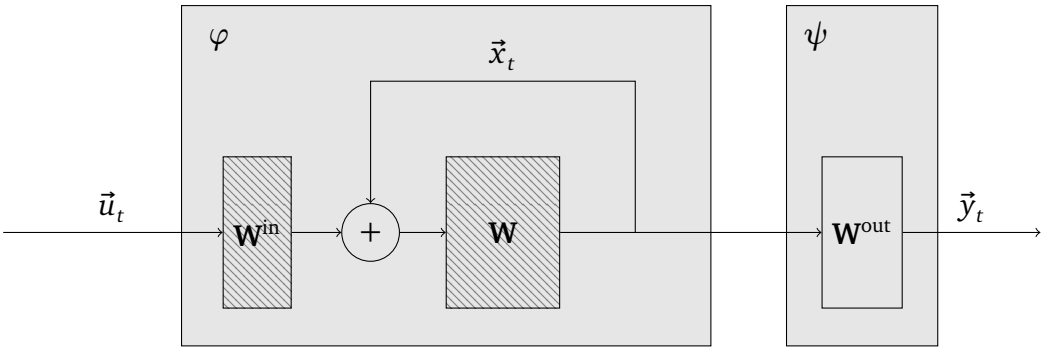


Figure 3.13: *Flow chart of an echo state network. The hatched squares mark the untrained weight matrices.*

optimized during the training phase of the network. This kind of RC network, that is very close to the common RNN is called *Echo State Network* (ESN). Another kind of RC network is the *Liquid State Machine* [Maass, Natschläger, and Markram 2004], which is studied in computational neuroscience, as it tries to mimic the spiking neurons of the brain. It is typically only used for research purposes as its computational complexity is much higher because it has to compute spiking state activations in every neuron.

Even though the reservoir is not optimized at all, it still takes the role of a non-linear expansion into a higher dimensional space and serves as the short-term memory of the input (Sec. 3.3.5). This is possible without adjusting the recurrent weights, given that the initialization of ESN is done carefully. In the following we will explore the demands that need to be made to the reservoir weights in order to ensure its functionality as a temporal memory, which enables predictions of the future based on the past inputs. Assuming that the output layer alone is able to extract the desired prediction from the current internal state, the ESN should be able to perform the same tasks as the general RNN. Additionally the problem of bifurcations in the error surface is eliminated, as the recurrent weights are not altered during the training. A practical comparison between RNNs and ESNs is hard to do and no general, objective favorite with regard to performance has been found yet.

3.3.3 The Echo State Property

As the goal of the ESN is to make predictions based on the history of a given time series, we have to construct an internal state that is a function of the previously seen inputs, such that

$$\vec{x}_t = f(..., \vec{u}_{t-1}, \vec{u}_t) \tag{3.37}$$

without adjusting the internal weights of the network during training. An RNN that exhibits this behaviour is said to satisfy the *echo state property* and called an ESN. The simplest ESN provides some insight into the temporal evolution of the internal state. It consists only of one hidden weight matrix and does not have any external inputs:

$$\vec{x}_t = \varphi(\mathbf{W}\vec{x}_{t-1}) \tag{3.38}$$

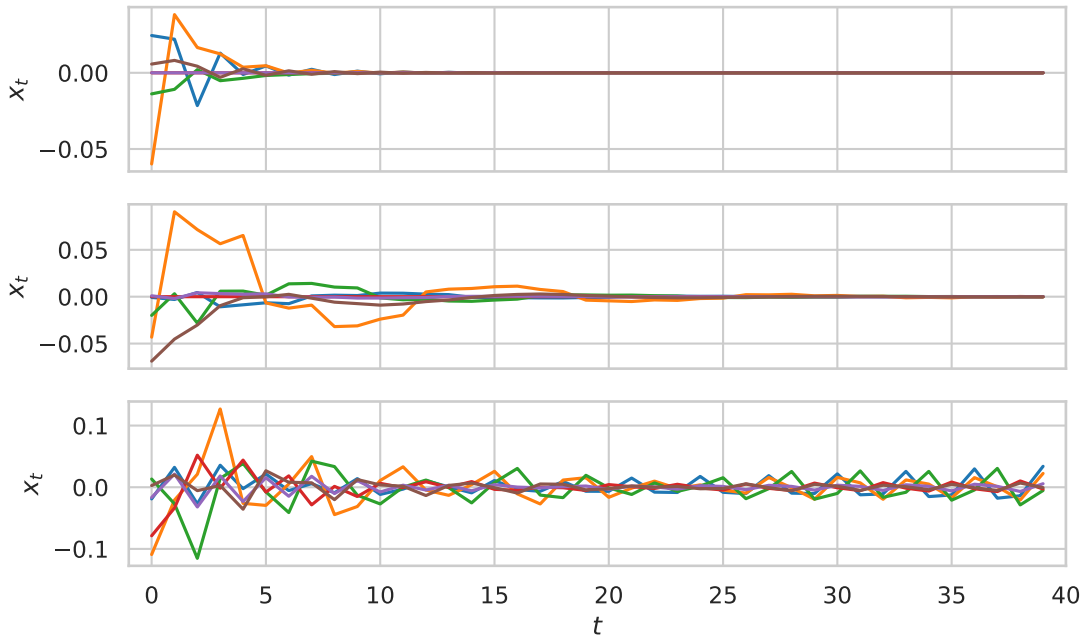


Figure 3.14: *State activations of the simple ESN without input weights as defined in Eq. 3.38. Each line depicts the temporal evolution of a single internal state unit. The initial state decays faster in the top graph, where the spectral radius is $\rho = 0.75$, slower in the middle with $\rho = 0.95$ and echoes indefinitely in the bottom graph with $\rho = 1.05$. The state dynamics are just as expected from the analysis in Sec. 3.2.3.*

By randomly initializing $\mathbf{W}$, the only hyper-parameter of the ESN is the spectral radius $\rho(\mathbf{W})$, which is defined as its the largest eigenvalue:

$$\rho(\mathbf{W}) = \max\{|\lambda_1|, |\lambda_2|, ..., |\lambda_n|\}. \tag{3.39}$$

There exist various libraries that efficiently calculate extremal eigenvalues for example based on simple power methods (described in A.1) or more sophisticated methods such as the Lanczos algorithm [Lanczos 1950]. Note that the spectral radius scales with a factor α:

$$\rho(\alpha\mathbf{W}) = \alpha\rho(\mathbf{W}). \tag{3.40}$$

The desired $\rho(\mathbf{W})$ can therefore be easily adjusted to an arbitrary value by multiplying with an appropriate factor.

As an example we create an initial state x_0 randomly from a normal distribution and the activation function φ is taken to be the hyperbolic tangent. The smaller the spectral radius, the faster the information of the initial state should decay as the network is stepping through time according to Eq. 3.38. The two upper plots of Fig. 3.14 show exactly this effect. The first plot is created by a network with $\rho = 0.75$, while the second one is the result of $\rho = 0.95$. The information of the random, initial state *echoes* much longer with the larger spectral radius and as a result, the memory of the network becomes longer. In both cases the echo state property is satisfied. In [Jaeger 2001] one can find the formal proof of the

sufficient condition for echo states:

$$\rho(\mathbf{W}) < 1, \tag{3.41}$$

which holds for all monotonically increasing φ. In the bottom plot of Fig. 3.14, one can see the effect of $\rho = 1.05 > 1$. Spectral radii $\rho > 1$ would, over time, lead to an exponential growth of the state activations, if it was not for the bounded, non-linear activation function $\varphi = \tanh(x)$. The hyperbolic tangent prevents this divergence and additionally, the internal state activations never settle down and will go on echoing forever. This strongly resembles the periodic fixed points discussed in Sec. 3.2.3 and has interesting, both positive and negative implications for the networks ability to memorize input and model predictions. Non-linear activation functions such as the hyperbolic tangent often prevent the state from exploding, thus making it possible to use spectral radii larger than one. This can still yield valid echo states, but the risk of over-saturating the state becomes much higher. This saturation is caused by the internal dynamic that is introduced to the state by entering the non-linear regime of the activation function and the resulting periodic fixed points. If the internal dynamic starts dominating the input to the ESN it becomes useless, as all the external information is overwritten by its internal dynamics. The implications that a varying spectral radius has for the performance of the ESN on different datasets is discussed in Chapter 5.

3.3.4 Reservoir With Inputs

In the more practical case of an ESN with external inputs, the internal state and thus the memory of the network is also influenced by the degree of which the input overwrites the previous internal state. To incorporate a scalar input in the network we introduce an input weight matrix $\mathbf{W}^{\text{in}}$ with dimensions $(N \times 1)$.

$$\vec{x}_t = \varphi(\mathbf{W}^{\text{in}}\vec{u}_t + \mathbf{W}^{\text{hidden}}\vec{x}_{t-1}). \tag{3.42}$$

The range of values that $\mathbf{W}^{\text{in}}$ is initialized with is a typical hyper-parameter for which there are no solid guidelines, apart from choosing small values. We will choose a uniform distribution of values between $-\kappa < 0 < \kappa$, where κ is typically smaller than one half. The implications of different initializations are shown in Fig. 3.15. It shows how the internal state reacts to a scalar input in form of a simple step function.
For a small $\rho = 0.5$ and large $\kappa = 0.1$, the state x is dominated by the input, which is visualized in Fig. 3.15 B. The state activations quickly approach certain value and form a plateau after just three to four time steps. When such a plateau is reached in all state activations, the network cannot determine the current position in the time series any more and will loose its predictive power. In plot C the network is set up with $\rho = 0.9$ and $\kappa = 0.02$. It takes longer for the initial random state to decay and the individual activations are not reaching a plateau any more. From the recurring state activations that are the same after each period but different for each point within one period, the network can infer where in the time series it currently is and make an accurate prediction for the next step.
Both ρ and κ must be chosen with respect to the task at hand. The last hyper-parameter of the ESN is the sparsity of the reservoir matrix $\mathbf{W}$. It only affects ESN memory for values well

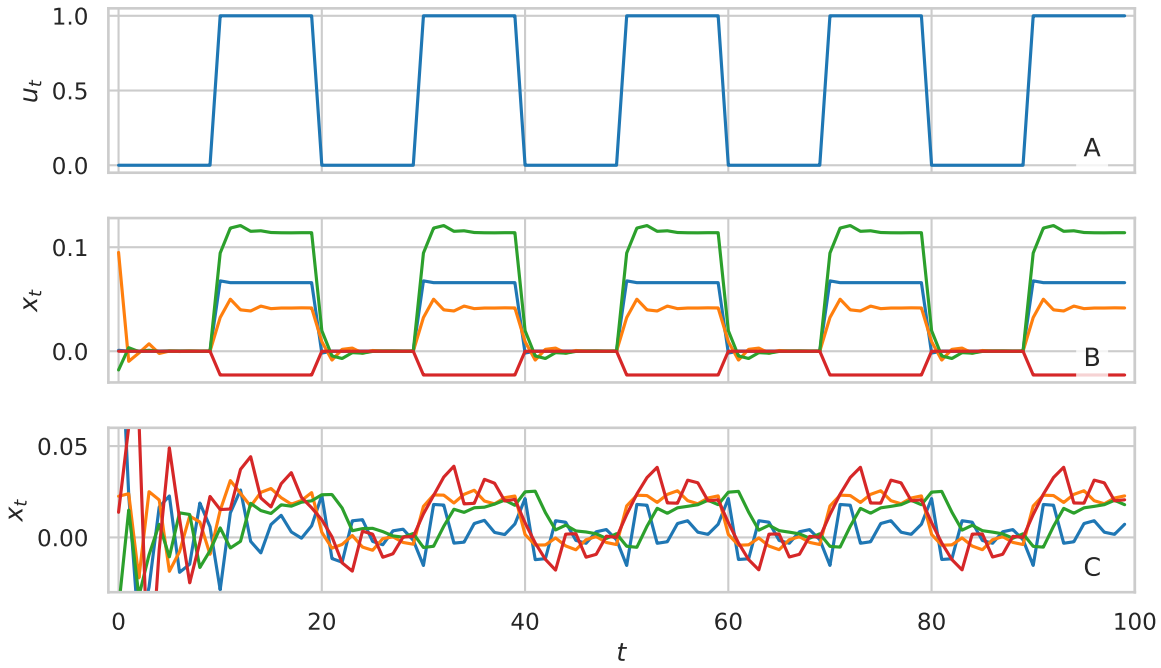


Figure 3.15: *Reaction of the internal state of the ESN to external input $u(t)$ (step function in graph A). A large κ leads to the input dominating the internal state, while a smaller κ enables it to memorize previous inputs.*

above 95% [Farkaš, Bosák, and Gergel' 2016], but it can significantly reduce computational complexity for large reservoir sizes. The specific choice of these parameters can be made with a hyper-parameter optimizer, as described in Sec. 3.4.

3.3.5 Short-term Memory & Reservoir Non-linearity

Short-term memory describes the dynamic memory of an internal ESN (or more generally RNN) state, which should not be confused with the memory of a network that is brought about by weight adaptions (e.g. through gradient descent), which is called long-term memory. We have already convinced ourselves that the internal state acts as a memory of the previous inputs, as depicted in Fig. 3.14. This memory could be almost arbitrarily long if it were not for the problem of vanishing gradients of very deep networks, which bounds the length of a practically learnable sequence to a number of data points in the order of ten. As already discussed, ESNs avoid the vanishing gradient problem altogether by not calculating recurrent gradients at all. Without this problem, how much can an ESN remember? It was shown by [Jaeger 2002] that for linear recurrent weights (where the activation function is the identity) the *memory capacity* (MC) is equal to the size of the ESN state:

$$\mathrm{MC} = N_{units} \ . \tag{3.43}$$

The memory capacity is defined as the number of inputs that a perfectly trained output matrix $\mathbf{W}^{\mathrm{out}}$ can extract from an ESN state which was created by feeding it a random sequence. A random sequence in this context means a series of numbers that were drawn from a uni-

form distribution. How to practically measure MC is shown in Sec. 5.1. Essentially the value of the MC give a measure of how many input steps the ESN can encode in its internal state. For the case of a non-linear activation function in the recurrency of the ESN the MC has an upper bound:

$$\mathrm{MC} < N_{units} \ . \tag{3.44}$$

Intuitively one might think that a larger MC will automatically lead to a better performing prediction. However, there is a second requirement that needs to be fulfilled in order to predict chaotic systems well: The degree of non-linearity (essentially expressed by the spectral radius) of the reservoir must be appropriate for the task. A network with the hyperbolic tangent as an activation function and a spectral radius close to or smaller than one acts mostly in the linear regime of the tanh. This means that the dynamics of the internal state are governed by an approximately linear expansion and in turn makes it very hard for the (also linear) output layer to produce a chaotic prediction. The obvious solution is to increase the spectral radius to values greater than one, but it turns out that there is a tradeoff between MC and the non-linearity of the ESN reservoir. Increasing the spectral radius above one degrades the memory of the ESN (Sec. 5.1), but makes it possible to predict chaotic systems (Sec. 5.2). A potential solution to this problem lies in separating the memory of the ESN from is linear transformation. This was attempted with an extension of the ESN to R^2SP (random static projections) by [Butcher et al. 2013], but its treatment is out of the scope of this work.

3.3.6 ESN Training

To train an ESN any of the previously introduced methods for training RNNs can be used. The fact that only the last layer is trained makes an additional, much faster least squared method possible. In the case of a linear output layer, the predictions of the network can be written as

$$\vec{y}_t = \mathbf{W}^{\text{out}} \vec{x}_t, \tag{3.45}$$

where $\vec{x}_t$ can generally either be the internal state or the internal state concatenated with the corresponding input $\vec{u}_t$. By collecting a number of states (and inputs) that are created with the untrained network, we can rewrite Eq. 3.45 into Eq. 3.48 with concatenated states $\mathbf{X}$ and desired outputs $\mathbf{D}$.

$$\mathbf{X} := (\vec{x}_1, ..., \vec{x}_T) \tag{3.46}$$

$$\mathbf{D} := (\vec{d}_1, ..., \vec{d}_T) \tag{3.47}$$

$$\mathbf{D} = \mathbf{W}^{\text{out}} \mathbf{X} \tag{3.48}$$

To find the optimal weights $\mathbf{W}^{\text{out}}$ we have to solve the overdetermined system in Eq. 3.48, which can be done for example via the pseudo-inverse method. It even gets by without introducing an additional hyper-parameter by utilizing the Moore-Penrose pseudo-inverse $\mathbf{X}^+$:

$$\mathbf{W}^{\text{out}} = \mathbf{D}\mathbf{X}^+ \tag{3.49}$$

$$\mathbf{X}^+ = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \tag{3.50}$$

Another method is called *Tikhonov Regularization*, which, outside geophysical contexts, is more frequently called *ridge regression* [Montgomery 2012]:

$$\mathbf{W}^{\text{out}} = \mathbf{D}\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{X}^{\mathrm{T}} - \beta\mathbf{I})^{-1} \tag{3.51}$$

The regularization coefficient β is introduced to reduce the risk of overfitting the output weights, which will quickly lead to diverging predictions when feeding the output of the ESN back into the input. Effectively β becomes another hyper-parameter, which has to be tuned in order to obtain good results. A derivation of Tikhonov Regularization can be found in the Appendix A.2.

3.4 Hyper-parameter Optimization

Before a machine learning model can be trained, certain decisions regarding the architecture of the network have to be made. This involves parameters such as the number of layers in feedforward networks, kernel sizes in convolutional nets or, as in the case of the ESN, sparsity and spectral radius of the reservoir. These parameters are called *hyper-parameters* (HP) and they fill a compact space $\mathbf{X}$. The goal of HP optimization is to find a configuration $\mathbf{x_i}$ that maximizes the performance of the network:

$$\arg\max_{\mathbf{x_i} \in \mathbf{X}} f(\mathbf{x_i}) \tag{3.52}$$

The function f is not known and very expensive to evaluate as one evaluation amounts to the creation of a trained NN. Often this optimization problem is solved by handily tuning each HP until a *reasonably* good $\mathbf{x}_{\text{opt}}$ is found, but there are a few methods that can be applied to automate this process. The two simplest methods are *grid searches* and *random searches* of the HP space. First, the space of valid HPs is defined. For example, a number of units $N = (10, 20, 30, ..., 100)$ and a learning rate $\eta = (0.1, 0.01, 0.001, ...)$. Grid search then performs an exhaustive search of all parameters, which is very simple to implement but suffers from the curse of dimensionality. Random search randomly samples $\mathbf{x_i}$ from the HP space, which does not have the problem of needing to perform an exhaustive search and is widely applied in practice, as it is still very simple to implement.
The next section describes an algorithm called *Bayesian Optimization*, which also samples the next $\mathbf{x_i}$ but incorporates the knowledge of already evaluated points in the HP space. It utilizes Bayes' Theorem which, adjusted to this problem, states that the *posterior* probability distribution of a model M (over the HP space $\mathbf{X}$) given a number of already evaluated samples $A \in \mathbf{X}$ is:

$$P(M|A) \propto P(A|M)P(M), \tag{3.53}$$

where $P(M)$ is the *prior* probability distribution over X, and $P(A|M)$ the *likelihood* of the samples given M.

3.4.1 Bayesian Optimization

The following description is largely based on article [Brochu, Cora, and De Freitas 2010] and will briefly introduce the concept of Gaussian Processes and how they are used in Bayesian Optimization as introduced by [C. K. Williams and C. E. Rasmussen 1996]. In summary, the Bayesian Optimization algorithm tries to maximize an objective function f by balancing exploration (evaluating areas where the true values of f are very uncertain) and exploitation (which tries to evaluate f where it is expected to be high). It rests upon the concept of Gaussian Processes (GP), which can be used to define a distribution over f. From the GP it is possible to calculate an acquisition function, which is used to efficiently sample the next $\mathbf{x}_i$ that should be evaluated. There are different acquisition functions that emphasize either exploration or exploitation. The method of Bayesian Optimization yields good results with only few evaluations and is very likely to perform well on objective functions with local maxima.

Gaussian Processes

A GP is defined by its mean function $m(\mathbf{x}_i)$ and its covariance function $k(\mathbf{x}_i, \mathbf{x}_j)$. This means that to each argument $\mathbf{x}_i$ we assign a random variable $f(\mathbf{x}_i)$. In analogy to the Normal distribution, a GP is formally written as:

$$f \sim GP(m, k) \tag{3.54}$$

where $m(\mathbf{x}_i)$ can be any function but typically is just zero, and a common covariance matrix is created with a Gaussian kernel:

$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(\frac{-||\mathbf{x}_i - \mathbf{x}_j||^2}{2}\right). \tag{3.55}$$

This kernel is close to one for values close to each other and approaches zero as the values grow further apart, which implies that close values are highly correlated. With this definition, a GP is equivalent to a multivariate Gaussian $\mathcal{N}(\mu, \Sigma)$ with mean vector μ and a covariance matrix Σ:

$$\mu_i = m(\mathbf{x}_i) \tag{3.56}$$

$$\Sigma_{ij} = k(\mathbf{x}_i, \mathbf{x}_j) \tag{3.57}$$

Given mean and covariance we can use the GP as a prior for the function f of which we want to find the maximum. By choosing a zero mean, the only actual prior information is a smoothly operating f, implied by the Gaussian kernel. The interesting part is how to update this prior with information that is gained by evaluating f a certain point to obtain a posterior. Assuming that we already have n observations $\{\mathbf{x}_{1:n}; f(\mathbf{x}_{1:n})\}$ we can find the covariance matrix $\mathbf{K}$ with Eq. 3.55. To find the probability distribution of an unobserved point $\mathbf{x}_{n+1}$, the conditional probability of f_{n+1} given the previous observations (also called predictive distribution) is needed:

$$P(f_{n+1}|\mathbf{x}_{1:n}; f(\mathbf{x}_{1:n})) = \mathcal{N}(\mu_p(\mathbf{x}_{n+1}), \sigma_p^2(\mathbf{x}_{n+1})), \tag{3.58}$$

where μ_p and σ_p are the predicted mean and standard deviation at an unknown point. They can be calculated thanks to the properties of the GP, which state that the observations and the arbitrary point $\mathbf{x}_{n+1}$ are jointly Gaussian:

$$\begin{bmatrix} f(\mathbf{x}_{1:n}) \\ f(\mathbf{x}_{n+1}) \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} \mathbf{K} & \mathbf{k}^T \\ \mathbf{k} & k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) \end{bmatrix}\right) \tag{3.59}$$

$$\mathbf{k} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_{n+1}) & \dots & k(\mathbf{x}_n, \mathbf{x}_{n+1}) \end{bmatrix} \tag{3.60}$$

From this it is possible to derive that:

$$\mu_p(\mathbf{x}_{n+1}) = \mathbf{k}^T \mathbf{K}^{-1} f(\mathbf{x}_{1:n}) \tag{3.61}$$

$$\sigma_p(\mathbf{x}_{n+1}) = k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k} \tag{3.62}$$

The resulting predictive distribution is typically very cheap to evaluate as the number of observations is low. With this distribution it is possible to find the so called *acquisition function*, which enables an educated guess of the next best point of evaluation of the objective function.

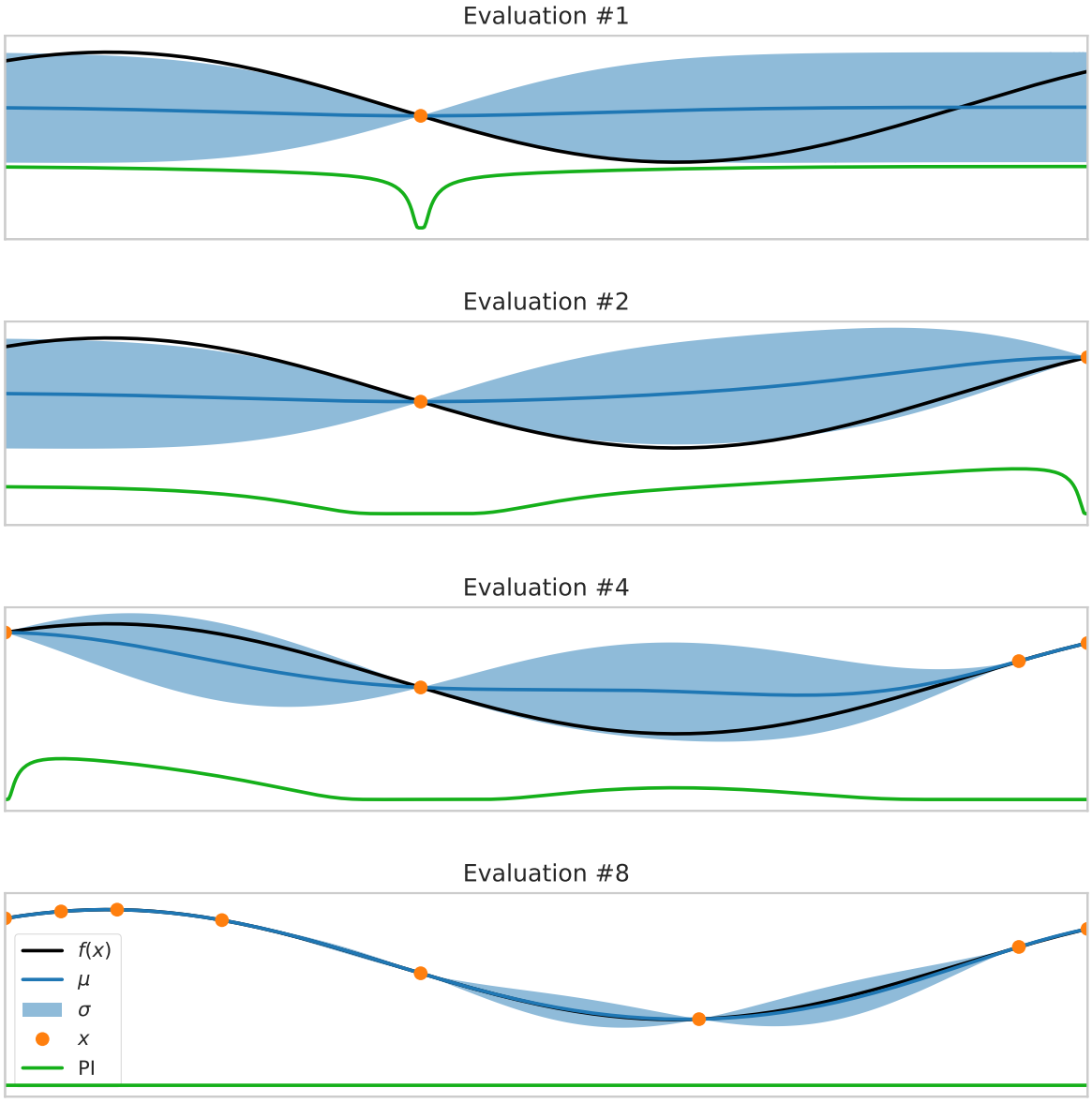


Figure 3.16: *A timeline of Bayesian Optimization. The plots show how the GP approximates the objective function better and better after each iteration. The maximum of the green acquisition function indicates where f will be sampled next. The uncertainty of already observed points is zero in this case, as there is no noise incorporated in the example. However, noisy objective functions can be dealt with by only slight additions to the described algorithm. An in depth description of Gaussian Processes and how they can be applied to ML can be found in [C. E. Rasmussen 2004].*

Acquisition function

The acquisition function is obtained from the predictive distribution of the GP and is defined such that it is high where the objective function f is *potentially* high. The probability of large objective values is high where either the uncertainty or the mean (or both) of the GP are large. Maximizing the acquisition function amounts to sampling f at

$$\mathbf{x}_{n+1} = \arg\max_{\mathbf{x_i} \in \mathbf{X}} \text{PI}(\mathbf{x}_i). \tag{3.63}$$

The function PI is an example of an acquisition function called the *probability of improvement*:

$$\begin{aligned} \text{PI}(\mathbf{x}_i) &= P(f(\mathbf{x}_i) > f(\mathbf{x}^+)) \\ &= \Phi\left(\frac{\mu_p(\mathbf{x}_i) - \mu_p(\mathbf{x}^+)}{\sigma_p(\mathbf{x}_i)}\right) \end{aligned} \tag{3.64}$$

Here Φ is the cumulative distribution function. The emphasis of PI clearly lies on exploitation, as samples with a mean lower than the current maximal mean will never reach PI values over 0.5. More sophisticated acquisition functions than PI, which are not purely driven by exploitation include *expected improvement*. A description of various acquisition functions and Bayesian Optimization as a whole can be found in [Brochu, Cora, and De Freitas 2010].

Chapter 4
Implementation

This chapter provides an overview over the problem we are trying to solve and shows how it was translated into a Tensorflow model. It highlights the most important technical considerations that were made during the implementation of the anomaly detection, which was described theoretically in the previous chapters. The different parts of the model, from input pipeline, over the ESN architecture to the hyper-parameter and weight optimizations, are illustrated with some brief pseudocode snippets.

4.1 Development

The code for this thesis was written in Numpy, Scipy, and Tensorflow, and is implemented as a Python package named `torsk`. Tensorflow [Abadi et al. 2016] is the most commonly used framework for implementing ML models. It is an open source library that is maintained mostly by Google engineers. In addition to implementing the most common GD algorithms, from plain GD to more advanced algorithms such as Adam, Tensorflow takes care of parallelizing and distributing the large matrix operations that make ML so powerful. The core design principle of Tensorflow is to separate the build and execution phases of the model. First, the desired model architecture is constructed by defining the *computational graph*. This graph is then optimized and during model execution certain nodes in the graph can be requested for evaluation. Tensorflow will then only execute the parts of the model that are necessary to evaluate the requested node. Some of the more intricate solutions that are presented (especially in Sec. 4.3.3) might be obsolete by the time of reading, as Tensorflow was under heavy development. It had not reached a stable API during the main development phase of the code, which was written mainly in Tensorflow versions 1.6-1.9.

4.1.1 The Journey

The implementation of the algorithm that was described theoretically in the previous chapters is the result of a search for a suitable RNN architecture and training method that has required numerous trial and error attempts. The result is a carefully orchestrated model that allows the prediction of high-dimensional, spatio-temporal, chaotic systems. Although it should be quite straight forward to implement a functioning ESN, with the theoretical background that was laid in the two previous chapters, the model is quite easy to break. Slightly defective training procedures can lead to the reproduction of the current time frame instead of a prediction of the future. Once the network is correctly implemented, it still is not trivial to make it produce sensible predictions for different datasets due to the large number of hyper-parameters that have to be set. Erroneous hyper-parameter setups easily result in predictions that quickly diverge, or do nothing but decay to the mean value of the input sequence. Especially the last error is very hard to fix without a good intuition of the internal dynamics of the ESN. Tensorflow's rather opaque RNN programming model does its part in slowing down the resolution of these problems.

At the beginning of this thesis it was not yet clear that the final approach would involve reservoir computing, which does not need to make use of the very convenient gradient and automated differentiation methods that Tensorflow offers. However, the automatic parallelization of the network still comes in handy. Additionally, Tensorflow comes with a tool called TensorBoard, which makes it easy to visualize the network architecture (Fig. 4.2) and the learning performance during development. A reevaluation of the advantages and drawbacks of Tensorflow and a second ML framework called *PyTorch*, that is currently gaining popularity, would most probably lead to the choice of PyTorch over Tensorflow. The dynamic features that are needed to implement RNNs are accesible in a much more pythonic way in PyTorch. Additionally, PyTorch seems to be faster at RNN tasks than Tensorflow, but this claim was not veryfied personally. That being said, PyTorch is still in an even earlier stage of development than Tensorflow, and was not as popular in the beginning of this work.

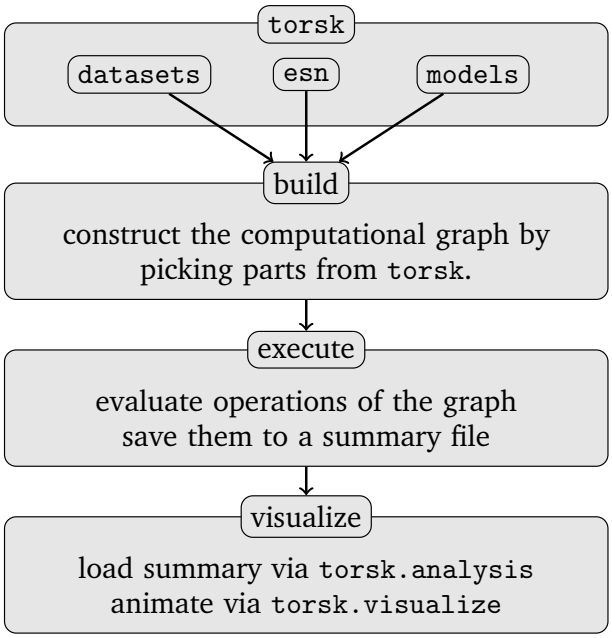


Figure 4.1: *Flow chart of the execution of* `torsk` *as it is realized with its run scripts.*

4.2 The `torsk` Python Package

The code that was developed in this work can be found on GitHub [`torsk` 2018], code snippets that are presented here are partially simplified to pseudocode to highlight only the important parts of the implementation and reduce the amount of cited boilerplate code. The parts where this is done are recognizable through inline comments.

The package that implements the anomaly detection is subdivided into three main parts: `torsk.datasets` (containing functions that create input pipelines for the different datasets), `torsk.esn` (where the ESN implementation itself is located), and `torsk.models` (containing different network architectures). In addition, there are two small visualization and analysis submodules to inspect the output of the Tensorflow models.

Fig. 4.1 shows a flow chart of the anomaly detection execution, to provide an overview over how it works. To realize the anomaly detection in Tensorflow, it is broken down into three phases: First, the build phase, where the computational graph is constructed. Second, the execution phase, which evaluates the graph in an appropriate order to optimize the ESN and make predictions. And third, the visualization of the results. If an online learning model (powered by GD algorithms) is run, then the execution and visualization phases can also be unified.

Once the graph is constructed it is, almost trivial to pick out the operations that are desired and evaluate them using Tensorflow primitives. As the execution and visualization phases of the algorithm are rather straight forward and can be examined more closely in the examples of the `torsk` repo, the rest of this section will focus on the build phase of the network.

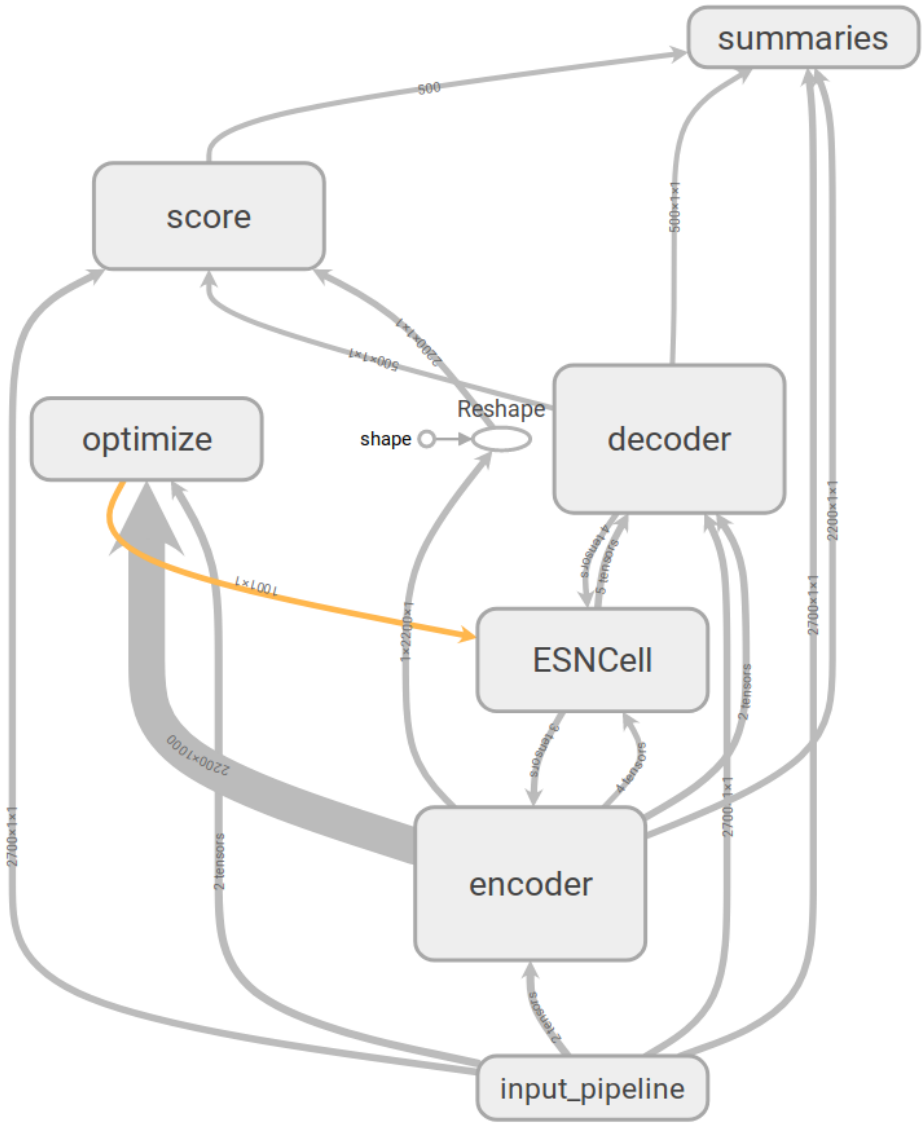


Figure 4.2: *Network graph as created by TensorBoard. The nodes represent the tensors that hold the actual values of input frames, weight matrices, etc. The edges represent the operations, such as multiplications/additions, which specify the data flow through the graph. The specified operations are evaluated during the execution phase of the network within a* `tf.Session`.

4.3 Building the Computational Graph

Before getting into the concrete implementation details of the graph, let us restate what we are trying to achieve: Based on an input sequence **u** we want to create a prediction **y** and compare this prediction to the truth **d** so that we can identify an anomaly based on the deviation of **y** and **d**.
This problem can be broken down in a few steps to gradually build up the complete computational graph (Fig. 4.2), which are listed below. Reminder: A frame of the input sequence **u** at time t is denoted by $\vec{u}_t$ (analogous for **y** and **d**).

1. **Read** the input sequence **u**. Implementation described in (Sec. 4.3.1) and represented

by the *input pipeline* node in the network graph. The input pipelines are defined in `torsk.datasets` and essentially convert Numpy arrays into Tensorflow's tensors.

2. **Feed** the input sequence **u** to the ESN. The ESN cell itself, with all its weights is represented by the *ESNCell* node. The *encoder* node takes care of looping over the input and feeding it frame by frame to the ESN, as well as recording the created internal states $\vec{x}_t$ and ESN outputs $\vec{y}_t$.

3. **Train** the ESN. This is done by the *optimizer* node, which adjusts the output weights of the ESN such that the outputs $\vec{y}_t$ match $\vec{u}_{t+1}$ as closely as possible.

4. **Predict** the sequence **y**. Similarly to step two the *decoder* feeds the necessary data to the ESN. The only difference is that now the output of the cell is fed back to its input. This node implements the *freely running* ESN.

5. **Detect** the eventual anomalies based on **y** and **d**. Described in Sec. 4.3.5 and represented by the *score* node. The sliding score is implemented both in Tensorflow and Numpy in `torsk.score`.

6. Evaluate the detected anomalies. This requires human interaction.

The steps 2.-4. are implemented in the `torsk.models` submodule and represent most of the logic of the network and the ESN implementations themselves reside in `torsk.esn`. In the following sections we will revisit the main nodes of the network to describe some specific implementation details that are worth noting.

4.3.1 Input Pipeline

The goal of the input pipeline is to create an `inputs` and a `labels` sequence that represent the series of inputs $\vec{u}_t$ and the desired outputs $\vec{d}_t$. The input pipeline takes care of reading slices of the input sequence to disk and hands them over to the parts of the NN architecture where it is needed. There are several ways of doing this in Tensorflow, but the recommended one is to use the *Dataset API* (`tf.data.Dataset`). If the data is not stored in the TFRecord format a dataset can be created from a Python generator. In the case of climate model output they are typically NetCDF files. The generator reads in the data, normalizes the values, and yields time slices of a desired length (Lst. 1). The generator is now converted into a dataset, which calls the generator as often as necessary to apply the defined transformations. The only significant operation that is applied to the SSH images is resampling them to a desired size. To comply with the Tensorflow *Estimator API* the pipeline returns a dictionary of all features and a single Tensor of labels. The current input pipeline is a very simple implementation that was not optimized with regard to speed, as this was not a bottleneck yet. It might have been sufficient to just work with `tf.placeholder` constructs instead. But because the final goal was to create an automated detection algorithm that can skim through a lot of data it seemed feasible to start using the slightly more complicated Dataset API. In addition, the input functions that are created with the Dataset API can be used in conjunction with the Estimator API.

```python
def normalize(data, vmin=None, vmax=None):
    if vmin is None or vmax is None:
        vmax, vmin = data.max(), data.min()
    return (data-vmin)/np.abs(vmin-vmax)

def make_generator(sequence_length, xslice, yslice):
    time, array = read_kuro(xslice, yslice)
    array = normalize(array)
    tottime = array.shape[0]
    for i in range(tottime-sequence_length):
        seq = array[i:i+sequence_length]
        t = time[i:i+sequence_length]
        yield t, seq

def pipeline(nr_prev_frames, nr_prev_frames):
    sequence_length = nr_prev_frames + nr_next_frames + 1
    def generator():
        return make_generator(sequence_length, slice(0, 100), slice(0, 100)):
    dataset = tf.data.Dataset.from_generator(
        generator=generator, ...)
    # some transformations like shuffling, batching and resizing ...
    iterator = dataset.make_one_shot_iterator()
    time, series = iterator.get_next()
    inputs, labels = series[:nr_prev_frames], series[nr_prev_frames:]
    features = {'inputs':inputs, 'time':time}
    return features, labels

if __name__ == '__main__':
    features, labels = pipeline(10, 10)
    inputs = features['inputs']
    with tf.Session() as sess:
        tf.global_variables_initializer().run()
        sess.run(inputs) # ==> retrieves first input series.
        sess.run(inputs) # ==> retrieves second input series.
```

Listing 1: *Simplified input pipeline for sea surface height (SSH) data. The lower part of the snippet shows the build phase of a very simple graph and how it is executed within a* `tf.Session`.

4.3.2 Echo State Network Cell

The central part of the anomaly detection algorithm is the ESN cell, as governed by the state space equations, which were introduced in Sec. 3.2.1. It is implemented both in a dense representation in the `ESNCell` and a sparse respresentation in the `SparseESNCell`. To create an RNN in Tensoflow's computational graph, it has to be unrolled along the time axis. For standard RNN tasks the graph creation is conveniently handled by the `tf.nn.dynamic_rnn` function, which only needs a `tf.nn.rnn_cell.RNNCell` object and some inputs to work. The ESN cell is implemented as a subclass of the `tf.contrib.rnn.LayerRNNCell`, which in turn is a subclass of the RNNCells to make them act like proper `tf.Layer` objects. An RNNCell needs to implement the `state_size` and `output_size` properties, as well as a `build` and a `call` method. The build method initializes all the weights that are needed in the call method, which defines the operations that are executed at each time step of the

```
class ESNCell(LayerRNNCell):
    def __init__(self, num_units, **kwargs):
        super(ESNCell, self).__init__(trainable=False, name=name, **kwargs)
        self._num_units = num_units
        self._activation = activation or math_ops.tanh
        self._spectral_radius = spectral_radius or 0.95
        self._density = density or 0.01
        # define initializers that are called in self.build

    @property
    def state_size(self):
        return self._num_units

    @property
    def output_size(self):
        return (self._out_units, self._num_units)

    def build(self, inputs_shape):
        self._reservoir = self.add_variable(
            name="reservoir", shape=[self._num_units, self._num_units],
            initializer=self._reservoir_init,
            trainable=False)
        # initialize all other weights ...
        self.built = True

    def call(self, inputs, state):

        x_input = math_ops.matmul(inputs, self._input_weights, name="Win_u")
        x_state = math_ops.matmul(state, self._reservoir, name="W_x")
        new_state = self._activation(x_input + x_state + self._input_biases)

        ext_state = array_ops.concat([new_state, inputs], axis=1)
        output = math_ops.matmul(ext_state, self._output_weights)
        return (output, new_state), new_state
```

Listing 2: *Pseudo code for the densely represented ESN cell. The full implementation is located at* `torsk.esn.esn_cell`.

RNN. In most Tensorflow RNN cells the output weights are defined outside the cell class as they are not part of the recurrency. The reasons for including them, as well as the state variable that is returned twice by the call method, will become clear during the section on the encoder and decoder of the network (Sec. 4.3.3). Pseudo code for the ESNCell is given in Lst. 2.

The initialization of the ESN reservoir is done via a custom `ESNReservoirIntitializer`. It inherits from the standard Tensorflow `Initializer` class and essentially calls a function that creates a reservoir matrix in Numpy and converts it to a `tf.Tensor`.

4.3.3 Encoder & Decoder

The encoders and decoders of an RNN take over the task of feeding the right inputs to the network during the training (encoder) and prediction (decoder) phases. The names

```
1 def prediction_helper(initial_inputs, params):
2     def initialize_fn():
3         finished = tf.tile([False], [1,])
4         next_inputs = initial_inputs
5         return finished, next_inputs
6 
7     def sample_fn(time, outputs, state):
8         """unnecessary for our task..."""
9         return tf.constant([0])
10 
11    def next_inputs_fn(time, outputs, state, sample_ids):
12        """Creates next inputs to ESNCell.call(inputs, state)
13        Params:
14            time: current time step
15            outputs: previous cell output. here: (prediction, state)
16            state: previous cell state
17            sample_ids: unused
18        """
19        finished = time >= params["nr_next_frames"] - 1
20        next_inputs, _ = outputs # pick only prediction, discard state
21        return finished, next_inputs, state
22 
23    helper = tf.contrib.seq2seq.CustomHelper(
24        initialize_fn=initialize_fn,
25        sample_fn=sample_fn,
26        next_inputs_fn=next_inputs_fn)
27    return helper
28 
29 def build_decoder(init_dec_input, init_dec_state, params)
30    with tf.variable_scope("decoder"):
31        dec_helper = prediction_helper(
32            init_dec_input, init_dec_output, params)
33        decoder = tf.contrib.seq2seq.BasicDecoder(
34            cell=cell, helper=dec_helper, initial_state=init_dec_state)
35        dec_output, dec_state, _ = tf.contrib.seq2seq.dynamic_decode(
36            decoder=decoder, output_time_major=None)
37    dec_output, dec_states = dec_output.rnn_output
38    return dec_output, dec_states
```

Listing 3: *Functions that create the prediction helper and decoder to feed the output of the ESN back to the input.*

encoder and decoder stem from the primary application of RNNs to natural language processing tasks. In language processing every word needs to be encoded into a (typically binary) vector representation before it can be fed to the network, hence the name. In our case the only thing the encoder does is to feed the true input $\vec{u}_t$ to the network and to record the corresponding output $\vec{y}_t$ for every frame in the input sequence. This is normally done by the `tf.nn.dynamic_rnn` function. The decoder is used during the prediction phase of the network. It takes care of feeding the current prediction $\vec{y}_t$ back to the ESN as the next input $\vec{u}_{t+1}$, which is not possible with the dynamic RNN encoder. Instead, the `tf.contrib.seq2seq.dynamic_decode` (line 35 of Lst. 3) routine can be used. It unrolls the ESN based on a decoder object, which in turn needs a helper to define the next inputs

and the state of the ESN. Unfortunately, the `dynamic_decode` function only returns, apart from the prediction, the final state of the ESN and none of the intermediate states. This is a major inconvenience for ESN validation, which is why the custom ESN cells (Sec. 4.3.2) return the state variable twice. This way they are treated like outputs, collected at every step, and returned together with the cell outputs in the end. Unfortunately, the practical realization of this idea is a slightly tedious task in Tensorflow that includes creating a custom *helper*, which, at the time of writing, are rather sparsely documented. However, it can be done as shown in Lst. 3. The `prediction_helper` function defines three functions that are needed to create a custom helper, which define the behaviour of the helper during the unroll of the ESN. Custom helpers provide quite some flexibility, including sampling next inputs from either labels (desired outputs) or predicted outputs, but this is not needed here, so `sample_fn` just returns an arbitrary value. The `next_inputs_fn` function is called at each prediction step with the previous cell output as one of its input arguments. The important thing is that on line 20 of Lst. 3 only the current prediction $\vec{y}_t$ is picked as the next input. Unfortunately, the extraction of all the intermediate states from the decoder involves changing the return statement of the ESN cells, which breaks them for the usage with `tf.nn.dynamic_rnn`. To fix this, Tensorflow's decoder features are used as encodeers by defining another dynamic decoding helper to return inputs instead of predictions. The few minor changes that have to be made to the helper can be found in the repository [`torsk` 2018].

4.3.4 Weight Optimization

The actual learning of the ESN is carried out by optimizing only the output weights $\mathbf{W}^{\text{out}}$ of the cell, which can be achieved via the pseudo-inverse method or Tikhonov Regularization (Sec. 3.3.6). The implementation of either of the two approaches is quite straight forward and can be found in the two functions `pseudo_inverse` and `tikhonov` of `torsk.models.lms`. Alternatively, the `torsk.models.grad` submodule implements the network with Tensorflows gradient optimizers.

4.3.5 Anomaly Detection

The anomaly detection, meaning the calculation of the anomaly score, can either be done online within the network graph, or offline by reading execution summaries and processing them. The former needs to be implemented in Tensorflow and is found in `torsk.score.tfscore`. The latter was done in Numpy, and resides in `torsk.score.npscore`. It is a simple function that receives the prediction errors and a `large_window_size` and a `small_window_size`, from which the normality score can be calculated according to Eq. 2.13.
If the anomaly detection is to be performed online, it needs to be included into Tensorflow's computational graph. The anomaly score is based on a sliding error window, so we need a loop. To prevent a redundant addition of operations to the graph Tensorflow provides the `tf.while_loop` function to create loops. It needs to be supplied with a conditional function that defines the stopping criterion and a loop body. The loop body computes the anomaly score. Note lines 21-22 where the computed anomaly score is injected into the

```
1 def sliding_score(errors, small_window, large_window, nr_prev_frames):
2     index = tf.constant(0)
3     shape = [errors.shape[0].value - nr_prev_frames] + errors.shape[1:]
4     scores = tf.zeros(shape)
5
6     def cond(i, errors, scores):
7         return i < shape[0]
8
9     def body(i, errors, scores):
10        # index at ith prediction (i+nr_prev_frames'th label)
11        j = i + nr_prev_frames
12        small_errors = errors[j:j+small_window]
13        large_errors = errors[j-large_window:j+small_window]
14
15        mu, var = tf.nn.moments(large_errors, axes=[0])
16        small_mu = tf.reduce_mean(small_errors, axis=0)
17
18        x = tf.abs(mu - small_mu) / tf.sqrt(var)
19        s = qfunction(x)
20
21        scores = tf.concat([scores[:i], [s], scores[i+1:]], axis=0)
22        scores.set_shape(shape)
23        return i+1, errors, scores
24
25    _, _, scores = tf.while_loop(cond, body, [index, errors, scores])
26    return tf.identity(scores, name='score')
```

Listing 4: *Tensorflow normality score implementation.*

`scores` variable by slicing and concatenation. This needs to be done this way because the `tf.assign` operation does not work as expected in a `tf.while_loop`. The shape of the `scores` must be reset after the concatenation in order to convince Tensorflow that the shape is invariant over the whole loop.

4.4 Hyper-Parameter Optimization

The choice of Hyper-parameters of a NN can either be done manually or with a number of different HP optimization algorithms (it is not a part of a node in 4.2). The most commonly used approach is probably Bayesian Optimization (as described in Sec. 3.4.1). The package 'scikit-optimize' provides a convenient implementation, whose usage is sketched in Lst. 5. The optimization is launched in line 38 by the `skopt.gp_minimize` (Gaussian Process minimize) function. The essential arguments to this call are the fitness function, the search dimension specification, and the initial values for each HP. At each step, the GP minimizer samples new evaluation parameters from the specified HP dimensions. They are chosen based on one of three acquisition functions (PI, EI, LCB), which is indicated by the argument `"gp_hedge"`. The algorithm probabilistically chooses one of the three available acquistion functions. The resulting `sample_params` are passed to the fitness function, which optimizes the output weights and creates ten test predictions. The prediction error over all

```python
import skopt
from skopt.space import Real

opt_steps = 100
output_dir = "hpopt"
dimensions = [
    Real(low=1e-5, high=1.0, name="esn_sparsity", prior="log_uniform"),
    # some more of those ...
]
initial_values = [
    0.1,    # esn_sparsity
    # ...
]
params = {
    # defines all default network parameters
}

@skopt.utils.use_named_args(dimensions=dimensions)
def fitness(sampled_params):
    params.update(sampled_params)

    features, labels = ds.input_fn(
        params["nr_prev_frames"],
        params["nr_next_frames"],
        shuffle=True)
    inputs = features['inputs']

    lms.build_model(inputs, labels, params)
    leaves = lms.train(params)
    # make 10 predictions and calculate their mean prediction error as a 'metric'

    if not np.isfinite(metric):
        metric = 1e10
    return metric

if __name__ == "__main__":

    search_result = skopt.gp_minimize(
        func=fitness, dimensions=dimensions, acq_func="gp_hedge",
        n_calls=opt_steps, x0=initial_values)

    skopt.dump(
        search_result, os.path.join(output_dir, "result.pkl"),
        store_objective=False)
```

Listing 5: *Bayesian optimization script based on the package [`skopt` 2018].*

predictions is averaged to obtain a metric that reflects the performance of the currently evaluated HPs. In the case that the predictions diverge the metric is set to a large finite value.

Chapter 5

Results

This chapter will analyze the practical capabilities of the echo state network for anomaly detection in time series. First we analyze the *memory capacity* (MC) of the ESN and show how well the ESN can predict a scalar chaotic time series. A tradeoff between networks with large MC and good capability to predict non-linearities is discussed before moving on to two input images and sea surface height prediction.

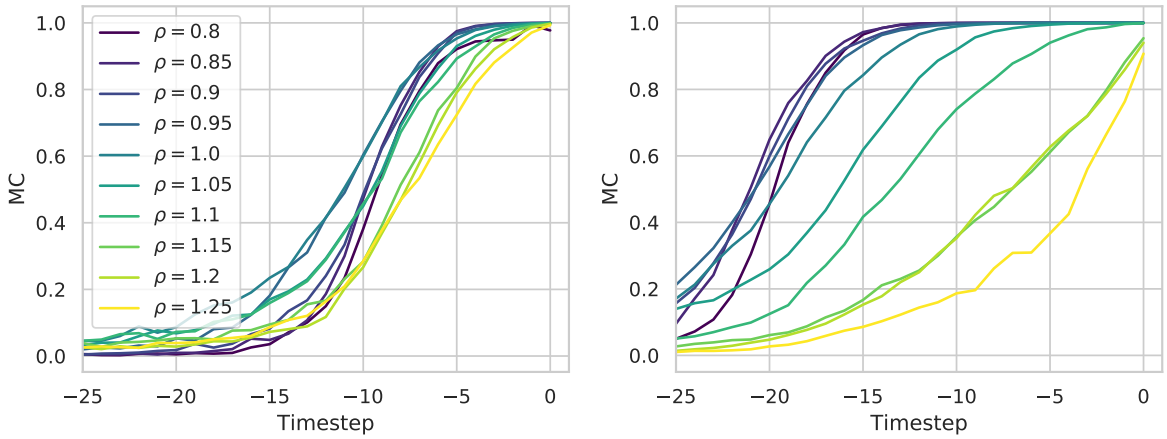


Figure 5.1: *Determination coefficient over time for various spectral radii of the reservoir and two different learning algorithms. On the left the network was trained with GD and on the right with pseudo-inverse regression. The SGD trained network does not show a clear depedence on spectral radius, while in the LMS network it is clearly visible.*

5.1 Short-term Memory

The short-term memory capacity (MC) of the internal state can be estimated through the so-called coefficient of determination R^2 and a simple experiment. R^2 is the squared correlation coefficient. For two variables X and Y it is defined by:

$$R^2 = \text{detCoeff}(X,Y) = \frac{\text{cov}^2(X,Y)}{\sigma^2(X)\sigma^2(Y)} \tag{5.1}$$

The network receives a random sequence $\mathbf{u}$ created from a uniform distribution and is trained to extract the last n inputs from the internal state $\vec{x}_t$ with 20 units. After training, the network extracts the last 40 time steps only from the last internal state (Fig. 5.2). By creating m sequences and collecting them in a matrix U

$$U = \begin{bmatrix} u_0^{t=-n} & \dots & u_m^{t=0} \\ \vdots & \ddots & \vdots \\ u_0^{t=-n} & \dots & u_m^{t=0} \end{bmatrix} \tag{5.2}$$

and the reconstructions of the network in a corresponding matrix Y, the coefficient of determination can be calculated for each time step by treating the columns of U and Y as the variables X and Y. The memory capacity can then be estimated by:

$$MC = \sum_{-n<i<0} \text{detCoeff}(U_i, Y_i), \tag{5.3}$$

The determination coefficient between extracted and targeted values is one if they are equal and zero if there is no correlation between them at all.
To find the MC of the ESN it was trained with batches of $m = 2000$ random sequences. Once with the Adam algorithm and once with the pseudo-inverse method (which of course only need a single batch of random sequences to be trained). After the training phase,

```
{
  "activation": "<function tanh at 0x7ffa8fddb2a8>",
  "esn_num_units": "30",
  "esn_spectral_radius": "---",
  "esn_density": "0.2",
  "in_weight_init": "0.01",

  "input_len": "50",
  "batch_size": "2000",
  "output_len": "50",

  "tikhonov_beta": "False",
  ...
}
```

Listing 6: *ESN setup parameters for the memorization task. The* `"False"` *value of the Tikohnov regularization parameter β means that the pseudo-inverse method was used. The spectral radius is varied from experiment to experiment.*

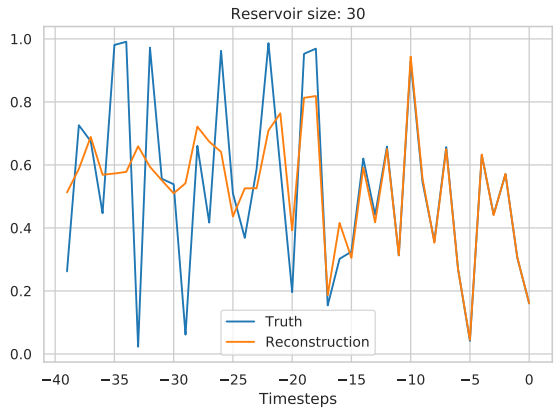


Figure 5.2: *True vs. reconstructed random time series. Most recent time step at $t = 0$.*

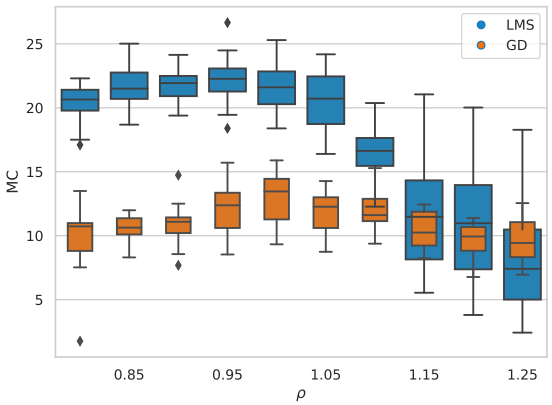


Figure 5.3: *MC over spectral radius ρ for the LMS and GD trained networks.*

another batch is fed to the network for evaluation. An exemplary reconstruction is shown in figure 5.2. It becomes evident that the most recent time steps can be reconstructed almost perfectly and the memory of the network becomes weaker further back in time. With this setup one can try to find the optimal spectral radius, which maximizes the memory capacity. Both plots in Fig. 5.1 show the coefficient of determination over the last 25 time steps at varying spectral radius ρ. Interestingly, training the ESN via a least mean squares approach (LMS, pseudo-inverse in this case) seems to be much more efficient than training with Adam in this case. In addition, scaling ρ has only a very small effect in the Adam-optimized network, while it is clearly visible in the LMS ESN, but this is probably due to the worse performance the SGD algorithm. Figure 5.3 suggests that MC increases as ρ approaches one and then quickly degenerates. This is thoroughly investigated computationally in a paper by [Farkaš, Bosák, and Gergel' 2016], which in essence confirms the conjectures that were made before.

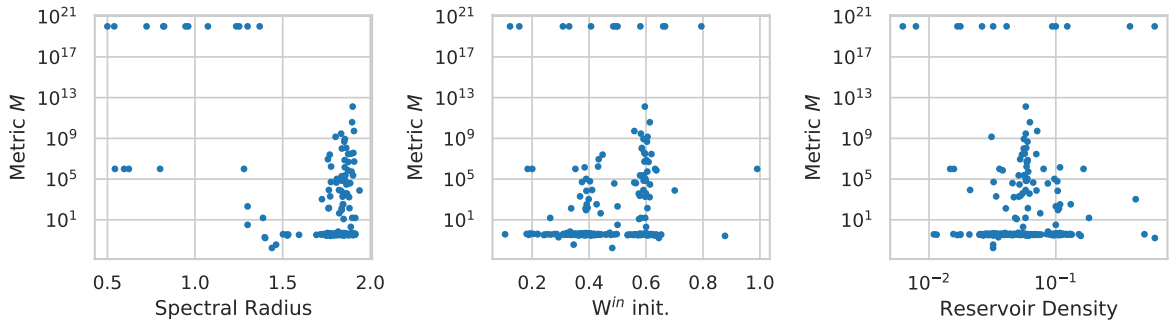


Figure 5.4: *Hyper-parameters over the squared error. The very large values are caused by diverging predictions that were clipped to a maximal* RMSE $= 10^{20}$*. The weight initialization and the sparsity parameters yield good performance in a certain range, the spectral radius only allows good performance for values larger than one.*

5.2 Mackey-Glass System

The first actual prediction task that will be solved by the ESN is the one of the Mackey-Glass system that was described in section 1.6. The reservoir is initialized with 500 units, which should be more than enough given that the period of the considered time series is below 100 time steps, and that the ESN should have a memory capacity slightly smaller than 500 steps. The ESN is fed with a sequence of length 2200, which creates the same number of internal states. To avoid transients in the internal state, the first 200 steps are discarded. The remaining 2000 internal states can be used to train the readout layer $\mathbf{W}^{\text{out}}$ via the pseudo-inverse method. Three hyper-parameters (HP) remain to be set: spectral radius, reservoir density, and input weight initialization parameter, which are found with the help of Bayesian Optimization (BO). For this purpose, the package `scikit-optimize` was used to minimize the RMSE

$$\text{RMSE} = \frac{1}{N}\sum_{i=0}^{N} ||d_i - y_i||_2 \tag{5.4}$$

over several predictions. The scikit algorithm needs an interval for every parameter to form an HP space that it can sample from, which where chosen as specified in table 5.1. The resulting optimal HPs were obtained over five BO runs. Each BO run evaluated 100 trained ESNs at different points in the HP space (Fig. 5.4). A surprising result is that the optimal spectral radius is significantly larger than one. In fact, the predicted signal just converges towards a value close to the mean of the series if the spectral radius is close to one.

Parameter	Best	Min	Max	Prior
Spectral radius	1.40	0.5	2.0	uniform
Weight init.	0.48	0.1	1.0	uniform
Density	0.03	0.01	1.0	logarithmic

Table 5.1: *Hyper-parameter results obtained via Bayesian Optimization*

This is peculiar, as section 5.1 showed a spectral radius close to one to be optimal for the memorization of sequences. It suggests that the crucial ingredient to predicting chaotic time

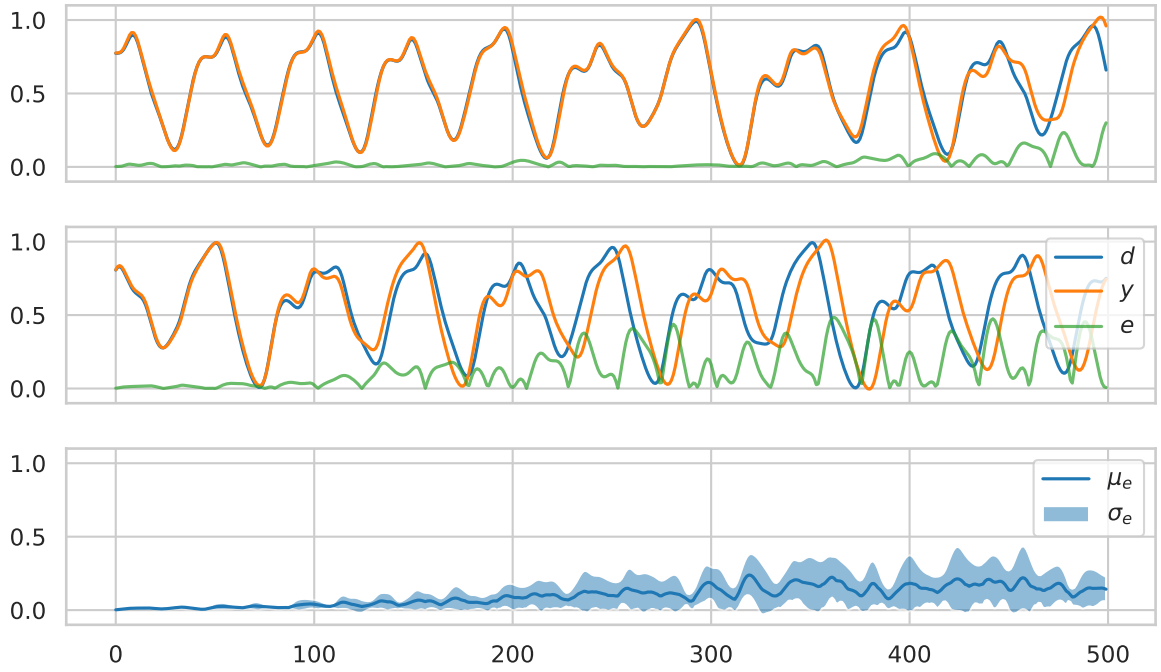


Figure 5.5: *The best (top) and worst (middle) predictions from 20 randomly chosen starting points of the Mackey-Glass series. All predictions were made by an ESN with optimal hyper-parameters. The target values d are shown in blue, prediction y in yellow, and the error $e = |d - y|$ in green.The plot in the bottom shows the variability of e over different predictions.*

series might not only be a very good knowledge of the past, but something else. In fact this is were the memory non-linearity tradeoff that was mentioned in section 3.3.5 comes into play. The more non-linear a prediction task becomes the higher must the spectral radius of the reservoir be to perform well. Unfortunately a large spectral radius degrades the MC of the ESN. A mathematically sound framework for reasoning about the memory non-linearity tradeoff is yet to be developed, but a rigorous computational analysis of the problem is carried out in an article by [Verstraeten et al. 2010]. For the purpose of an anomaly detection it is enough that we found good parameters for the prediction task. The quality of the predictions is depicted in Fig. 5.5. The predictive power of the ESN varies slightly over different parts of the Mackey-Glass system. By randomly choosing different starting points in the sequence and creating predictions for the next 500 frames, we can evaluate the average error that the ESN makes at each frame. Fig. 5.5 shows the best and worst predictions in the two upper plots and the average deviation of prediction and target in the bottom. An anomaly detection might be feasible within the first 100 frames, an anomaly that is caught beyond that point might as well be caused by a degraded prediction.

At this point, *finally*, all necessary parts to try and find artificially introduced anomalies in the MG system are in place. The anomalies are created by varying the parameter γ of the MG Eq. 1.3. By default it has a value of $\gamma = 0.1$. Every 400 steps it is decreased by an amount δ set back to its initial value after 50 steps. Depending on the magnitude of δ this creates rather smooth looking anomalies. Easily visual anomalies are created with a value of $\delta = 0.05$. To search the outliers, the trained network receives a sliding window of 300

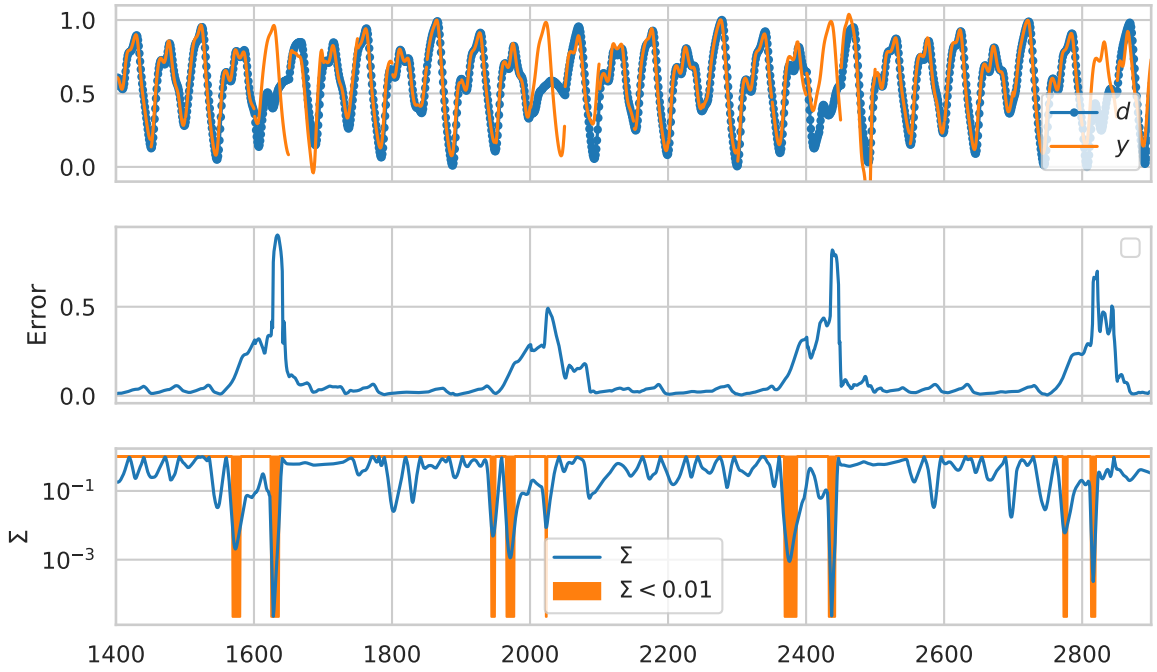


Figure 5.6: *Mackey Glass system with artificially introduced anomalies at every 400th step with $\delta = 0.05$. The anomaly score catches the irregularly high error regions slightly too early, because the prediction error goes up as soon as the prediction window reaches into the anomalous regions.*

steps of the anomalous sequence. The predictions for the next $N_p = 50$ steps into the future are collected and the *mean prediction error* (MPE) is calculated for every sequence that was fed to the ESN:

$$\text{MPE} = \frac{1}{N_p} \sum_{t=1}^{N_p} |d_t - y_t|. \tag{5.5}$$

The MPE is recorded for the whole dataset and the normality score Σ (Eq. 2.13) is computed with a large window size of 100 steps and a small window size of 5 steps.. The result is shown in Fig. 5.6. With delight we can find that all anomalies are easily caught. To demonstrate the capabilities of the ESN to efficiently find outliers in chaotic systems like the MG system, Fig. 5.7 depicts how the algorithm performs on anomalies that would, most probably, not be spotted by a human.

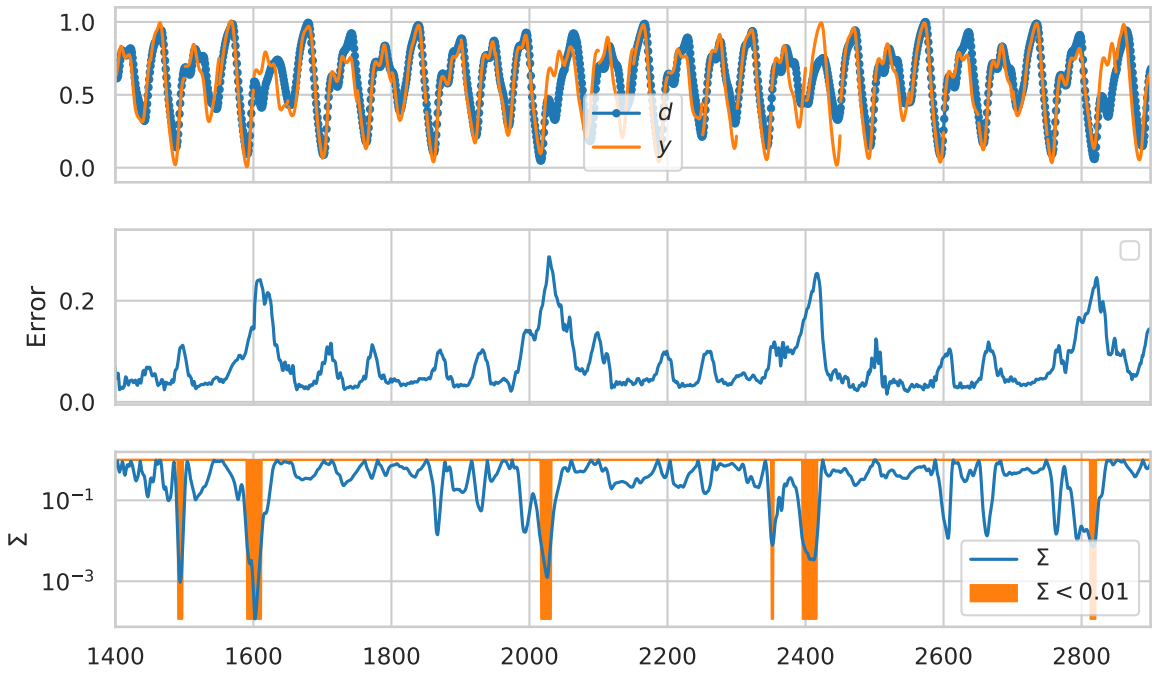


Figure 5.7: *Outliers that would have probably not been caught by humans. Created with* $\delta = 0.03$. *One false positive at* $t = 1500$.

```
{
  "esn_cell": "<class 'esn.cell.esn_cell.ESNCell'>",
  "activation": "<function tanh at 0x7f6090a53c08>",
  "esn_num_units": "500",
  "esn_spectral_radius": "1.40",
  "esn_density": "0.03",
  "in_weight_init": "0.48",

  "nr_equilibration_frames": "200",
  "nr_prev_frames": "2200",
  "nr_next_frames": "500",

  "tikhonov_beta": "False",
  ...
}
```

Listing 7: *ESN setup parameters for MG prediction.*

```
{
  "esn_cell": "<class 'esn.cell.esn_cell.ESNCell'>",
  "activation": "<function tanh at 0x7f4e1ef96d70>",
  "esn_num_units": "500",
  "esn_spectral_radius": "1.74",
  "esn_density": "0.10",
  "in_weight_init": "0.01",

  "nr_equilibration_frames": "200",
  "nr_prev_frames": "1500",
  "nr_next_frames": "200",

  "tikhonov_beta": "5e-06",

  "input_image_dims": "(2, 1)",
  ...
}
```

Listing 8: *ESN setup parameters for DO event detection. The hyper-parameters were found via Bayesian Optimization.*

5.3 Dansgaard-Oeschger Events

The second dataset consisting of ice-core records that reach back 100k years has two components. The δ^{18}O sequence, which is directly connected to surface temperature and the Ca^{2+} series, that measures dust content and shows similar patterns. Both sequences lack a clear reappearing pattern, which was clearly visible in the Mackey-Glass series, despite the chaotic nature of the MG system. This makes a one time training over a part of the dataset that contains most of variability impossible. It should however still be possible to project several dozen years ahead by moving to an online learning mode. In the online mode, the output weights are recomputed with every new time slice that is fed to the network. The resulting short-term predictions y are shown in Fig. 5.8. They were achieved by training the ESN for 1500 input steps and predicting the next 200 steps (the time step of the series is one year). Due to the very irregular dataset the predictions are not very good, but at points where the sequences change abruptly they deviate much more from the target values than they do otherwise. This can be exploited to detect the DO events which are located exactly at these abruptly changing points. Apart from the online weight adaption, the procedure is the same as with the MG system. The score is calculated from the MPE and thresholded, resulting in the detection of anomalies at the yellow colored regions of Fig. 5.8. Abrupt changes after steady periods are easily detected by the algorithm. DO events that lie close to each other tend to slip through. This is due to the fact that the ESN learns from the previous DO events and as it is trained on these anomalies, they are considered normal.

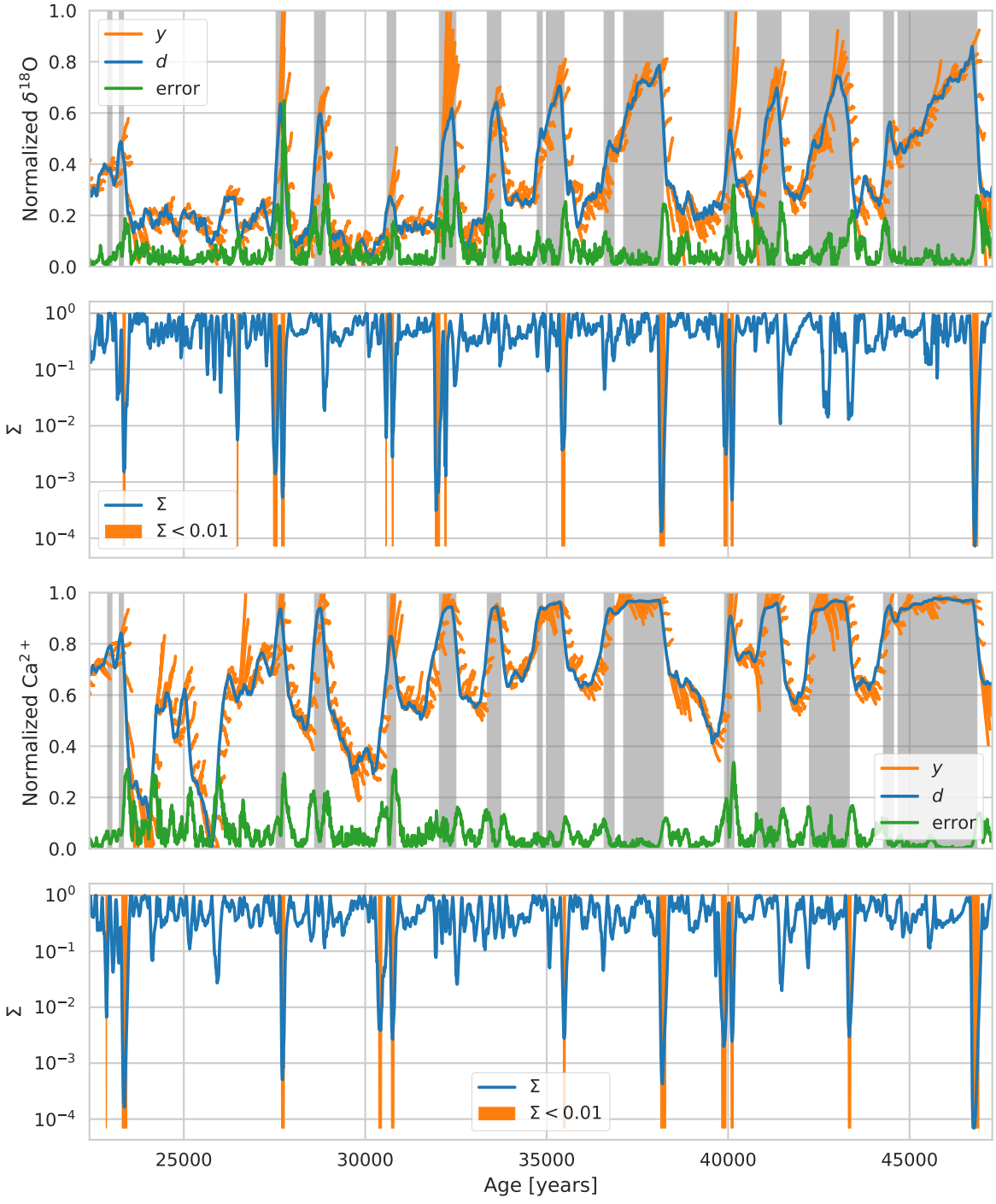


Figure 5.8: *Detected anomalies in the DO dataset.*

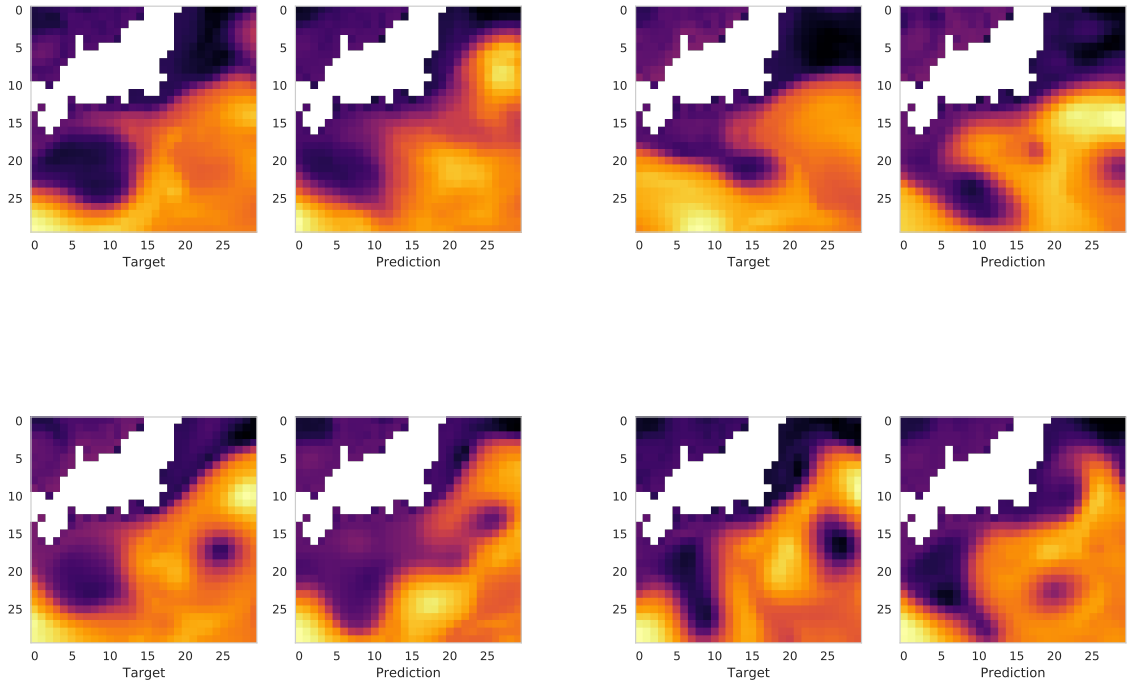


Figure 5.9: *Four exemplary predictions and corresponding desired outputs in the Kuroshio region. Shown is the 150th predicition frame out of 200.*

5.4 Kuroshio

The prediction of the Kuroshio region involves significantly more data that needs to be processed by the ESN. This means that the reservoir size has to grow appropriately. As a rule of thumb, the size of the internal state $\vec{x}_t$ should be at least as big as the period of the dataset is long. The most obvious period in climate simulations is typically the annual signal which, in our case of 3-day means, is 122 steps long. To be able to remember whole video sequences, this number should be multiplied by the number of pixels in each frame to obtain the state size. Even if we consider raw inputs of 100 x 100 pixels and resample them to a size of 30 x 30, this would still result in a state size greater than 100 000, which would result in a memory use of more than 40 GB of the internal square reservoir matrix alone. This is clearly unacceptable. Luckily, the pixels are highly correlated and we can hope that the ESN is able to compress some of the information of the input images it receives. Sizing the reservoir down to 10000 units and keeping it very sparse will decrease the memory usage of the network to an acceptable amount of a few hundred MB.

Apart from using a sparsely represented ESN reservoir, the approach is again the same as before. The ESN receives 1300 input frames and predicts the next 200, which amounts to a prediction of roughly two years. Exemplary predictions and the corresponding desired target outputs are shown in Fig. 5.9. The figures 5.10 and 5.11 show the 25th row of two more exemplary target and prediction evolutions over time. The prediction error that is shown is quite low for the predicted two years, but just to be sure only the first year,

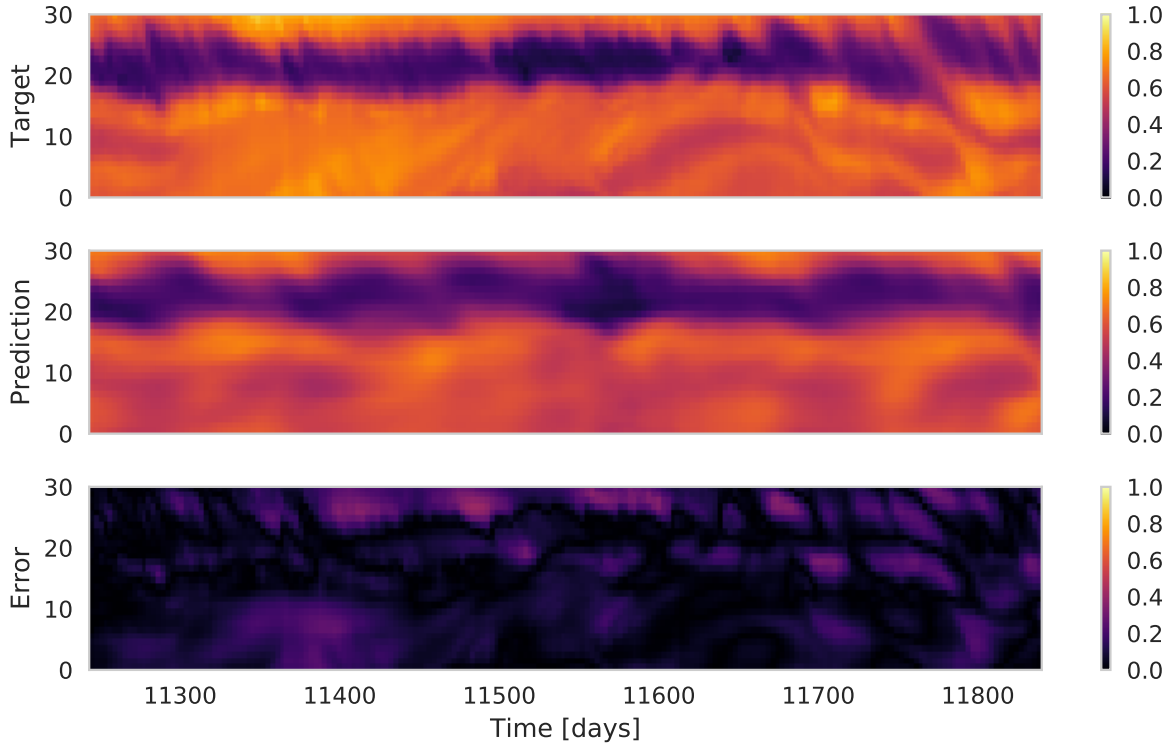


Figure 5.10: *Exemplary prediction 200 steps into the future.*

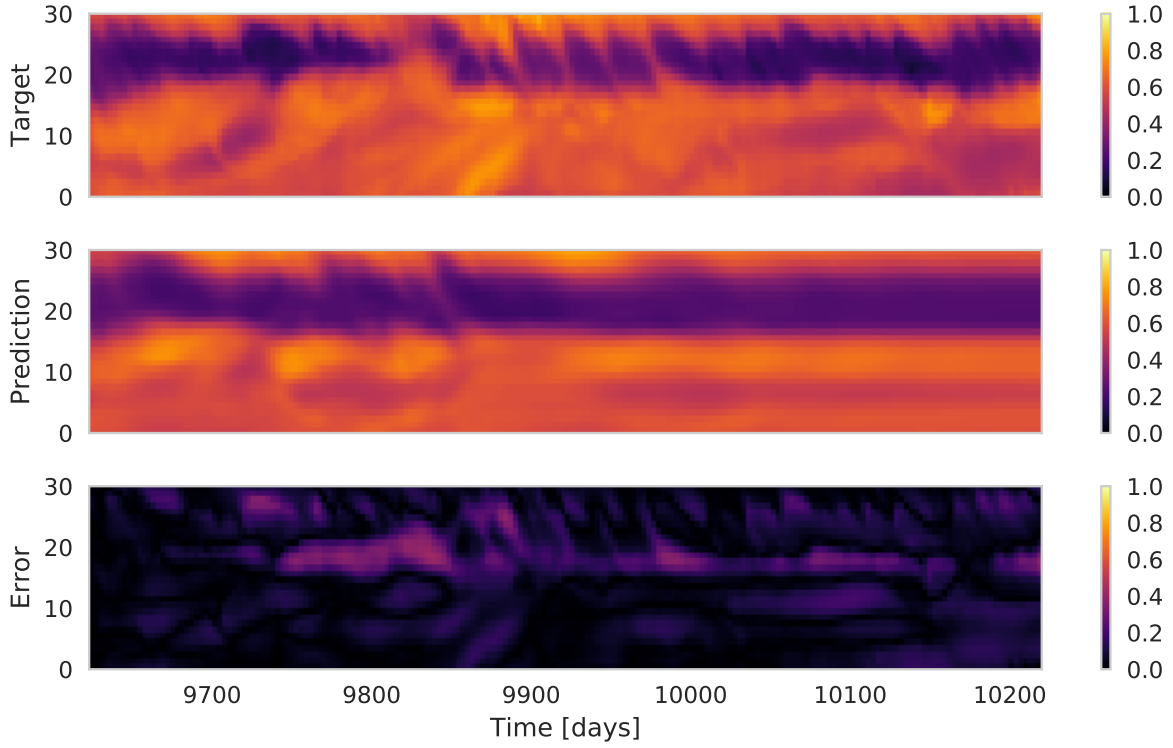


Figure 5.11: *Another prediction example. Here it looks like the internal state of the ESN has reached a fixed point, so the prediction remains constant after day 10100. This fixed point interestingly produces predictions that are very similar to the mean contracted state of the Kuroshio.*

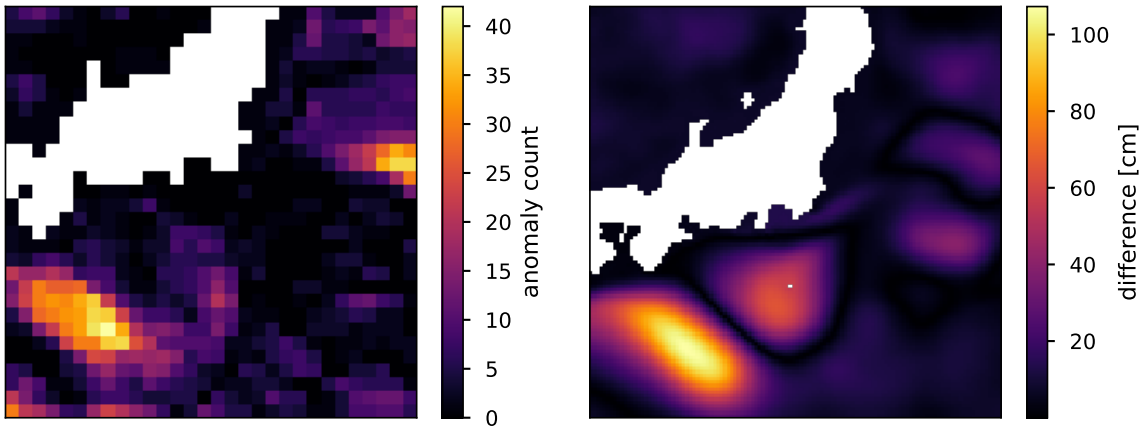


Figure 5.12: *Count of all pixels with a normality score that is lower than 0.01, compared to the assumed true form of the Kuroshio anomaly (obtained from the difference of its two states).*

meaning the first $N_p = 100$ frames, will be used for the subsequent anomaly detection.
Fig. 5.13 compares the target and prediction values for the last prediction step ($N_p = 100$) that is considered for the anomaly detection. The error plot shows the MPE for each prediction. A large deviation of prediction and target is visible around day 15500. The normality score (plotted in a logarithmic color scale) that was calculated from the MPE sequence is very low in this region as well. It seems to be a promising candidate for the Kuroshio anomaly. By summing up all instances where the anomaly score $\Sigma < 0.01$ we can create a map of outliers. The result is shown in Fig. 5.12. It shows a region with a lot of outliers just where we expect them to be. The similarity between the anomaly count and the difference of elongated and contracted states is obvious and the result will be regarded as a successful detection of the Kuroshio anomaly.
The second region of high anomaly counts is where the Kuroshio turns towards the North Pacific basin. This anomaly is caused by a large unpredicted eddy that enters the analyzed area from the East. This anomalous region would probably vanish if the analysis area is shifted eastward, away from Japan. This would be the obvious next step of the outlier search, but unfortunately a thorough analysis of the whole world is out of the scope of this work and will remain as one path that a further studies of this problem could take.

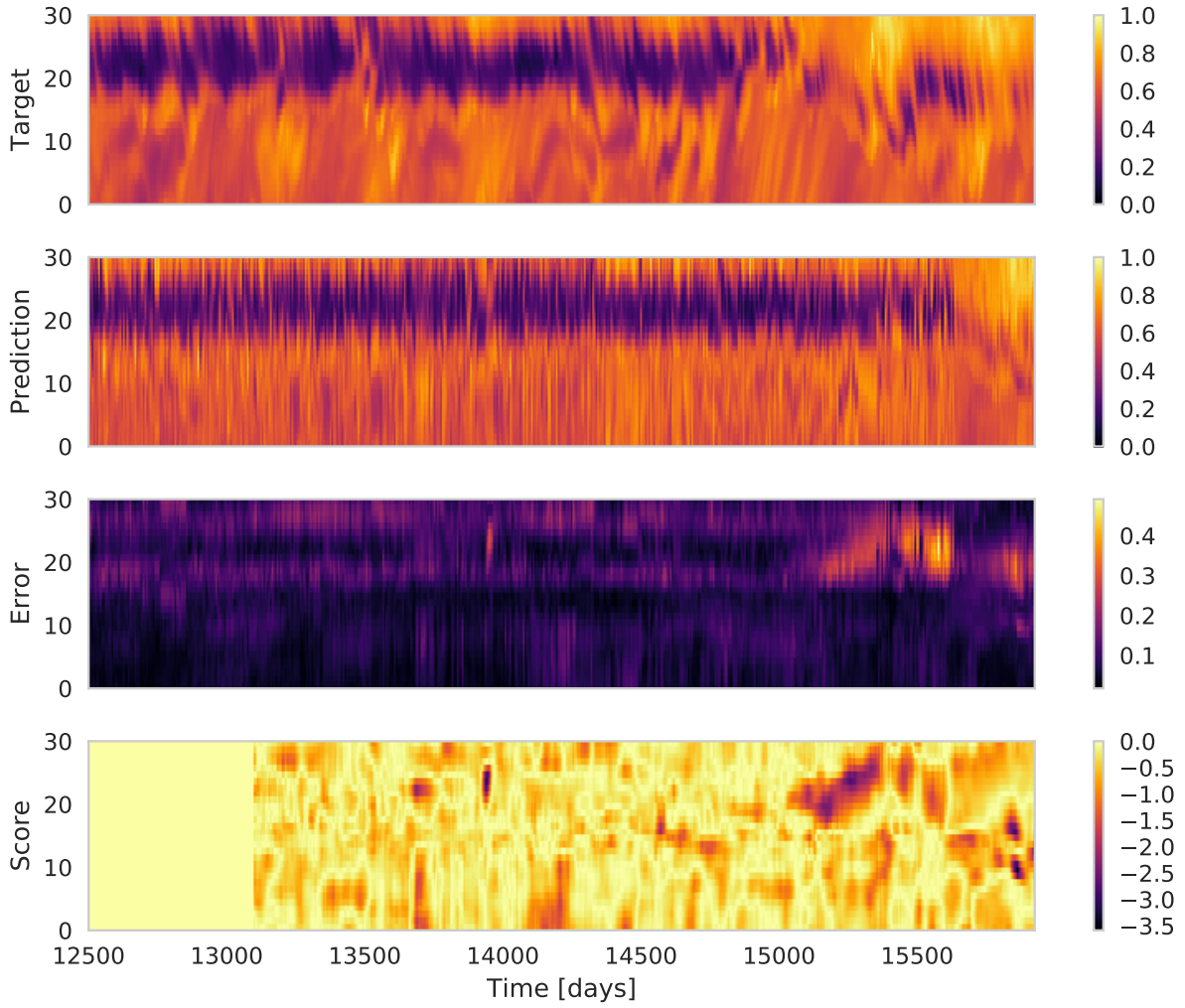


Figure 5.13: *The prediction plot shows the true values of one row of the SSH frames that were fed to the ESN. In the target plot we can see the prediction that was made based on the internal state from 100 steps before. The error plot depicts the mean error of one prediction sequence and in the bottom we can see the resulting anomaly score on a logarithmic color scale.*

```
{
  "esn_cell": "<class 'esn.cell.esn_cell.SparseESNCell'>",
  "activation": "<function tanh at 0x7f1c4d457d70>",
  "esn_num_units": "10000",
  "esn_spectral_radius": "2.0",
  "esn_density": "0.001",
  "in_weight_init": "1.0",

  "nr_equilibration_frames": "300",
  "nr_prev_frames": "1300",
  "nr_next_frames": "200",

  "tikhonov_beta": "False",

  "input_image_dims": "(30, 30)",
  ...
}
```


Listing 9: *ESN setup parameters for Kuroshio anomaly detection. The Bayesian Optimization resulted a large range of parameters with similar performance as long as the spectral radius was larger than 1.3 and the weight initialization parameter larger than 0.8. The chosen hyper-parameters reflect the need for a sufficiently non-linear reservoir.*

Chapter 6

Conclusions & Further Work

This chapter draws come conclusions from the practical as well as theoretical considerations that were made during this work. Additionally it provides an outlook on how the developed model could be improved with respect to prediction and computational performance.

6.1 Conclusions

After a considerable amount of thought was put into the understanding of the dynamical nature of ESNs and recurrent neural networks in general I successfully implemented a framework that is able to predict high-dimensional, spatio-temporal, chaotic time series. It can therefore be concluded, that a general, automated anomaly detection of anomalies in chaotic systems is possible. The problem of predicting a chaotic system long enough to detect anomalies in it was solved without making any assumptions about the underlying physics. In addition, the application of the ESN to the prediction problem has lead to a very cost efficient search algorithm, which is quite remarkable, considering that it is able to create said chaotic predictions.

The only adjustments that have to be made from one dataset to the next are a few hyper-parameters. They can be set approximately by hand by reasoning about the specific requirements of the given problem. Both the state size and the spectral radius of the ESN can be chosen within narrow bounds by examining the requirements of the problem. The state size should be at least as large as the period of the examined sequence and the spectral radius must be large enough to capture its non-linearity. However, to achieve the best results the exact choices of hyper-parameters are best left to algorithms such as Bayesian Optimization. This reduces the amount of tweaking that has to be done to apply the model to new datasets to a minimum.

The second goal of finding new physical behaviour in the available climate model output remains to be achieved. It will require some house keeping in the analysis part of the code, but all the necessary parts are in place.

With this proof of concept we have shown that the concepts of machine learning can be of much use in the field of oceanography and climate physics as a whole. A wider application of the concepts from artificial intelligence and machine learning promises to make the full potential of climate data exploitable in completely new ways.

6.2 Future Work

The ultimate goal of this work was to create an algorithm that skims through huge ocean simulation datasets and returns interesting areas to search for new physical behaviour. To achieve this, the created anomaly detection, which currently can process small windows, has to be scaled to work on the large, global domain. This can be achieved two different ways: The most straight forward approach is to apply the algorithm by shifting the window and analyzing each one of them independently. Another way would be to create overlapping predictions, that are merged and then fed back to the network.

Of course, also the current prediction model can be improved. There is still much room for improvement by increasing the complexity of the ESN. This would, most probably, reduce the amount of false positives of the detection. Additionally, the performance of the ESN could benefit tremendously from implementing it with a different framework than Tensorflow.

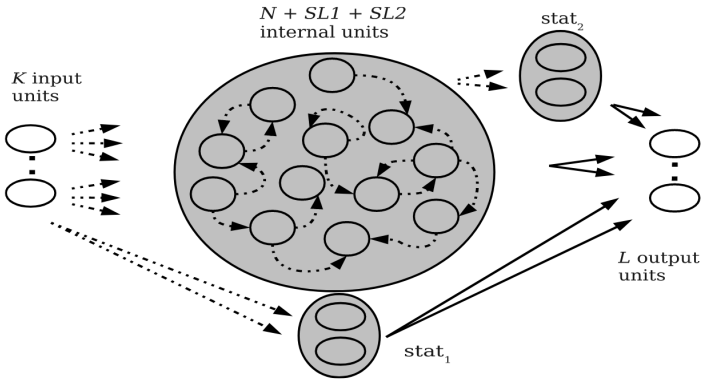


Figure 6.1: *Reservoir with random static projections [Butcher et al. 2013]. The $stat_1$ and $stat_2$ layers are static feedforward layers, that take the part of the non-linear transformation. The recurrent internal units represent the memory of the network. Their reservoir matrix can be initialized with a spectral radius close to one to maximize memory without the need of introducing non-linearities.*

6.2.1 Scaling to the Global Domain

By applying the detection algorithm to small windows of the model output, many independent ESNs can be run in parallel. This can be scaled until the memory limitations of the available computing resources are reached. This spatial parallelization would enable the analysis of the full, global domain.

The parallel ESNs can either operate independently within their window, or the predictions can be processed into one large super-prediction. For this the windows must, of course, overlap, which introduces a considerable amount of communication into the problem. The super-prediction can subsequently be divided back into the windows that are fed back to the ESN during the prediction phase.

6.2.2 Predictor Improvements

With the rather basic setup of a single ESN cell as the prediction network, there is still much room for improvement of the prediction model. The most obvious improvement would be to add a number of convolutional layers in front of the ESN cell. This could lead to significant improvements by leveraging the spatial information of the input as described in Sec. 3.1.1. Unfortunately, this would make the use of convolutional layers and ain turn an expensive GD algorithm for the weight optimization necessary, because the convolutional kernels have to be learned. By sacrificing the convolutional layer for a standard dense layer the GD optimization could be avoided, because it is possible to construct untrained feedforward layers in a similar fashion like the reservoir of the ESN. Networks that contain untrained feedforward layers are called *extreme learning machines* and were introduced by [Huang, Zhu, and Siew 2006]. The combination of the two concepts was introduced by [Butcher et al. 2013] and termed *reservoir with random static projections* (R^2SP). This architecture promises to solve the memory non-linearity tradeoff by separating non-linear transformation and internal memory of the network and could thus be much more powerful.

Another interesting approach to improve the performance of the network would be to em-

ploy *evolutionary optimization* techniques [Fogel 1997] to find a good reservoir matrix **W** which could be reused for the whole ocean dataset. This would eliminate fluctuations in prediction performance from run to run.

6.2.3 Performance Improvements

Probably the greatest advantage of the presented approach is its low computational demand. Creating predictions of a few hundred frames for a window of 30x30 pixels does not take longer than a few seconds on a standard laptop. However, for a truly automated application of the model to very large datasets with long sequences it would be interesting to implement it in another framework. Tensorflow does not excel in RNN computations and it would be quite simple to implement the network, for example, in Numpy and parallelize it with Bohrium [Kristensen et al. 2013]. This would get rid of a lot of overhead that is caused by the dynamic graph creation that Tensorflow is doing.
Additionally, the ESN, due to its static nature, holds the potential of being compiled to an FPGA, which could improve performance even further.

6.2.4 Understanding Recurrent Networks

To date, the general understanding of the ESN and RNNs in general is not very good. It is assumed that an RNN classifier needs as many fixed points as it has features to separate, but fundamental understanding is still lacking. A further investigation of the influence of fixed-points and period cycles within the internal state evolution on the memory and predictive capabilities of RNNs would be very interesting. This could lead to a more solid understanding of how the hyper-parameters of the ESN are influencing its performance.

Chapter A

Derivations

A.1 Spectral Radius

The spectral radius is a key parameter that determines some properties of the echo state network weight matrix, which we will here denote as $\mathbf{A} \in \mathbb{C}^{n\times n}$. If $\mathbf{A}$ has eigenvalues $\lambda_1, ..., \lambda_p$, where $p \leq n$, the spectral radius is defined as

$$\rho(\mathbf{A}) = \max\{|\lambda_1|, ..., |\lambda_p|\}. \tag{A.1}$$

Thus, to obtain the spectral radius we somehow have to find a reasonably close estimate for the absolute largest eigenvalue. For very large matrices, it obviously becomes infeasible to calculate all eigenvalues which has a computational complexity of $O(n^3)$. One method of quickly computing the largest eigenvalue is called **inverse iteraion** (or inverse power method), but before diving into the exact algorithm we will go through a quick recap of the linear algebra that is necessary for inverse iteration.
If the matrix $\mathbf{A}$ is diagonalizable, we can find a matrix $\mathbf{S}$ such that

$$\mathbf{A} = \mathbf{S}^{-1}\Lambda\mathbf{S}, \tag{A.2}$$

where Λ is a diagonal matrix containing the eigenvalues $\lambda_1, ..., \lambda_p$. From Eq. A.2 follows that:

$$\mathbf{AA} = (\mathbf{S}^{-1}\Lambda\mathbf{S})(\mathbf{S}^{-1}\Lambda\mathbf{S}) = \mathbf{S}^{-1}\Lambda^2\mathbf{S}, \tag{A.3}$$

which means that for any continuous function one can write:

$$f(\mathbf{A}) = \mathbf{S}^{-1}f(\Lambda)\mathbf{S}, \tag{A.4}$$

because every continuous function can be approximated by a polynomial function. In our case it will turn out to be very useful, that

$$(\mathbf{A} - \sigma\mathbb{I})^{-1} = \mathbf{S}^{-1}(\Lambda - \sigma\mathbb{I})^{-1}\mathbf{S}, \tag{A.5}$$

because only the eigenvalues (on the diagonal of Λ) are affected by the inversion, but the eigenvectors stay exactly the same.

Power Method

Suppose $\vec{x} \in \mathbb{C}^n$, that can be written as the sum of eigenvectors and eigenvalues:

$$\vec{x} = \lambda_1\vec{e}_1 + ... + \lambda_p\vec{e}_p, \tag{A.6}$$

where $|\lambda_1| > |\lambda_2| > ... > |\lambda_p|$ it follows that:

$$\mathbf{A}^j\vec{x} = \lambda_1^j\vec{e}_1 + ... + \lambda_p^j\vec{e}_p, = \lambda_1^j\sum_{i=0}^{p}\left(\frac{\lambda_i}{\lambda_1}\right)^j\vec{e}_i, \tag{A.7}$$

which means that for large enough j Eq. A.7 converges to:

$$\vec{\mu} = \mathbf{A}^j\vec{x} \to \lambda_1^j\vec{e}_1. \tag{A.8}$$

Eq. A.8 is called the **power method** for finding eigenvalues. From the estimate $\vec{\mu}$ of the largest eigenvector one can easily obtain the estimate σ of the corresponding eigenvalue by applying the Rayleigh quotient:

$$\sigma = \frac{\vec{\mu}^* \mathbf{A} \vec{\mu}}{||\vec{\mu}^* \vec{\mu}||} \tag{A.9}$$

From Eq. A.7 one can quickly see that it has an error of $O(|\lambda_2/\lambda_1|^j)$, which means that it converges very slowly if the two largest eigenvalues are very close to each other.

Inverse Iteration

The inverse iteration method aims to solve the problem of slow convergence of the power method by manipulating the eigenvalues of the iterated matrix favourably.
Suppose we can find a reasonably close estimate σ of an eigenvector λ_i of $\mathbf{A}$, then the matrix $(\mathbf{A}-\sigma\mathbb{I})$ is almost singular, because on of the elements of its diagonalized counterpart is close to zero. [1] This means, according to Eq. A.5 that $(\mathbf{A}-\sigma\mathbb{I})^{-1}$ has one very large eigenvalue, which we can exploit for a fast conversion of the power method. Starting from a random vector $\vec{x}_0$, the iterative update equation for the inverse power method reads:

$$\vec{x}_{i+1} = (\mathbf{A}-\sigma\mathbb{I})^{-1}\vec{x}_i, \tag{A.10}$$

The shift σ is updated at each iteration by using the Rayleigh quotient Eq. A.9. This results in an increasingly good estimate of the desired eigenvalue, which in turn speeds up the convergence of the inverse iteration. This only becomes a problem once the calculated shift hits the exact value of an eigenvalue of $\mathbf{A}$. Then the shifted matrix becomes singular.
Eq. A.10 can be rewritten in terms of a linear system solver:

$$\vec{x}_{i+1} = \text{linearSolve}(\mathbf{A}-\sigma\mathbb{I},\ \vec{x}_i) \tag{A.11}$$

in order to exploit the speedup of solving a linear system compared to an inverse matrix computation, which results in especially large speedups as soon as the matrix $\mathbf{A}$ is very sparse.
The last remaining problem to solve is a reasonable estimate for the largest eigenvalue of $\mathbf{A}$, preferably and upper bound in order to make sure that the iteration does not converge to the second largest eigenvalue. The most straight forward thing to do here would be taking the Frobenius norm of $\mathbf{A}$, which provides an intuitive upper bound of the spectral radius:

$$|\lambda|^k||\vec{e}|| = ||\lambda^k\vec{e}|| = ||\mathbf{A}^k\vec{e}|| \leq ||\mathbf{A}^k|| \cdot ||\vec{e}|| \tag{A.12}$$

$$|\lambda^k| \leq ||A^k|| \tag{A.13}$$

A faster way of doing this is based on the so called Gershgorin disks Gershgorin 1931. If a_{ij} is an element of the complex matrix $\mathbf{A}$ we can define $R_i = \sum_{j\neq i}|a_{ij}|$ as the radius of a disk $D(a_{ii},R_i)$ centered at a_{ii}. Gershgorin's circle theorem states that the eigenvalues of $\mathbf{A}$ have lie within the Gershgorin disks. Picking the the largest R_i in combination with the largest a_{ii} as an upper bound thus provides an upper bound for the largest eigenvalue in much less than $O(n^3)$ (complexity of the Frobenius norm).

[1] The estimate σ is often called shift, which is why this method is also known as the *inverse shift method*.

This iterative inverse method typically converges in a few steps, which makes it vastly superior to calculating every eigenvalue and picking its maximum in order to find the spectral radius.

A.2 Tikhonov Regularization

The goal of the optimization of the ESN output weights is to find a $\mathbf{W}^{\text{out}}$ such that for a given state x_t we can produce a prediction of the next time step:

$$y_t = \mathbf{W}^{\text{out}} x_t \tag{A.14}$$

By collecting a number of states $\mathbf{X} = (x_1, ..., x_T)$, that are produced by the inputs $u_1, ..., u_T$, we can write

$$\mathbf{D} = \mathbf{W}^{\text{out}} \mathbf{X}, \tag{A.15}$$

with $\mathbf{D} = (d_1, ..., d_T)$ as the conactenated desired outputs. The optimal output weights can then be found by simply solving the overdetermined system:

$$\mathbf{W}^{\text{out}} = \mathbf{D}\mathbf{X}^{\text{T}}(\mathbf{X}\mathbf{X}^{\text{T}})^{-1} \tag{A.16}$$

With this simple least squares solution, the output weights are very prone to overfitting, which is why a regularized version, also known as Tikhonov Regularization, often yields more general results, that let the ESN perform much better in the freely running mode.
Tikhonov Regularization minimizes the function Φ:

$$\Phi(\mathbf{W}^{\text{out}}) = ||\mathbf{D} - \mathbf{W}^{\text{out}}\mathbf{X}||^2 + \beta^2 ||\mathbf{W}^{\text{out}}||^2, \tag{A.17}$$

where the first term represents the *misfit* of the outputs to the target and the second term introduces a the regularization. A larger coefficient β will favor smaller output weights in the solution, which prevents them from overfitting.
Eq. A.17 can be written as:

$$\Phi(\mathbf{W}^{\text{out}}) = \left|\left| \begin{pmatrix} \mathbf{D} \\ 0 \end{pmatrix} - \mathbf{W}^{\text{out}} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix} \right|\right|^2, \tag{A.18}$$

which can be solved by:

$$\begin{pmatrix} \mathbf{D} \\ 0 \end{pmatrix} = \mathbf{W}^{\text{out}} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix} \tag{A.19}$$

$$\begin{pmatrix} \mathbf{D} \\ 0 \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix}^{\text{T}} = \mathbf{W}^{\text{out}} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix}^{\text{T}} \tag{A.20}$$

$$\begin{pmatrix} \mathbf{D} \\ 0 \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix}^{\text{T}} \left[\begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \beta\mathbf{I} \end{pmatrix}^{\text{T}} \right]^{-1} = \mathbf{W}^{\text{out}} \tag{A.21}$$

$$\mathbf{W}^{\text{out}} = \mathbf{D}\mathbf{X}^{\text{T}}(\mathbf{X}\mathbf{X}^{\text{T}} + \beta^2\mathbf{I})^{-1} \tag{A.22}$$

Bibiliography

[1] Martín Abadi et al. “Tensorflow: a system for large-scale machine learning.” In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] Charu C Aggarwal. *Outlier analysis*. eng. New York, NY: Springer, 2013.

[3] Subutai Ahmad et al. “Unsupervised real-time anomaly detection for streaming data”. In: *Neurocomputing* 262.Supplement C (2017). Online Real-Time Learning Strategies for Data Streams, pp. 134–147. ISSN: 0925-2312. DOI: *https://doi.org/10.1016/j.neucom.2017.04.070*. URL: *http://www.sciencedirect.com/science/article/pii/S0925231217309864*.

[4] Pierre Baldi and Kurt Hornik. “Neural networks and principal component analysis: Learning from examples without local minima”. In: *Neural Networks* 2.1 (1989), pp. 53–58. ISSN: 0893-6080. DOI: *https://doi.org/10.1016/0893-6080(89)90014-2*.

[5] Atilim Gunes Baydin et al. “Automatic Differentiation in Machine Learning: a Survey”. In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43. URL: *http://jmlr.org/papers/v18/17-468.html*.

[6] GEP Box and GM Jenkins. *Statistical Models for Forecasting and Control*. 1970.

[7] Eric Brochu, Vlad M Cora, and Nando De Freitas. “A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning”. In: *arXiv preprint arXiv:1012.2599* (2010).

[8] J.B. Butcher et al. “Reservoir computing and extreme learning machines for non-linear time-series data analysis”. In: *Neural Networks* 38 (2013), pp. 76–89. ISSN: 0893-6080. DOI: *https://doi.org/10.1016/j.neunet.2012.11.011*. URL: *http://www.sciencedirect.com/science/article/pii/S0893608012003085*.

[9] Eugen Diaconescu. “The use of NARX neural networks to predict chaotic time series”. In: 3 (Mar. 2008).

[10] Kenji Doya. “Bifurcations in the learning of recurrent neural networks”. In: *Circuits and Systems, 1992. ISCAS’92. Proceedings., 1992 IEEE International Symposium on*. Vol. 6. IEEE. 1992, pp. 2777–2780.

[11] Izharwallach Michael Dzamba. “AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery”. In: *arXiv* (2015).

[12] Bruno Eckhardt and Demin Yao. “Local Lyapunov exponents in chaotic systems”. In: *Physica D: Nonlinear Phenomena* 65.1-2 (1993), pp. 100–108.

[13] Igor Farkaš, Radomír Bosák, and Peter Gergel'. "Computational analysis of memory capacity in echo state networks". In: *Neural Networks* 83 (2016), pp. 109–120.

[14] Chrisantha Fernando and Sampsa Sojakka. "Pattern Recognition in a Bucket". In: *ECAL*. 2003.

[15] David B Fogel. "Evolutionary algorithms in theory and practice". In: *Complexity* 2.4 (1997), pp. 26–27.

[16] Kenichi Funahashi and Yuichi Nakamura. "Approximation of dynamical systems by continuous time recurrent neural networks". In: *Neural Networks* 6.6 (1993), pp. 801–806. ISSN: 0893-6080. DOI: *https://doi.org/10.1016/S0893-6080(05)80125-X*.

[17] S. Gershgorin. "Über die Abgrenzung der Eigenwerte einer Matrix." Russian. In: *Bull. Acad. Sci. URSS* 1931.6 (1931), pp. 749–754.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[19] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. "Extreme learning machine: theory and applications". In: *Neurocomputing* 70.1-3 (2006), pp. 489–501.

[20] Herbert Jaeger. "Short term memory in echo state network". In: *Technical Report GMD Report 152* (2002).

[21] Herbert Jaeger. "The echo state approach to analysing and training recurrent neural networks-with an erratum note'". In: 148 (Jan. 2001).

[22] Herbert Jaeger and Harald Haas. "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication". In: *Science* 304.5667 (2004), pp. 78–80. ISSN: 0036-8075. DOI: *10.1126/science.1091277*.

[23] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: *1412.6980*. URL: *http://arxiv.org/abs/1412.6980*.

[24] Mads RB Kristensen et al. "Bohrium: unmodified NumPy code on CPU, GPU, and cluster". In: *Python for High Performance and Scientific Computing (PyHPC 2013)* (2013).

[25] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[26] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[27] Edward N Lorenz. "Deterministic nonperiodic flow". In: *Journal of the atmospheric sciences* 20.2 (1963), pp. 130–141.

[28] Mantas Lukosevicius and Herbert Jaeger. "Reservoir computing approaches to recurrent neural network training". In: *Computer Science Review* 3.3 (2009), pp. 127–149. ISSN: 1574-0137. DOI: *https://doi.org/10.1016/j.cosrev.2009.03.005*. URL: *http://www.sciencedirect.com/science/article/pii/S1574013709000173*.

[29] Wolfgang Maass, Thomas Natschläger, and Henry Markram. “Computational models for generic cortical microcircuits”. In: *Computational neuroscience: A comprehensive approach* 18 (2004), p. 575.

[30] The Mainichi. “Kuroshio Current curves for the first time in 12 years, various marine effects expected”. In: *The Mainichi* (Oct. 2017).

[31] Douglas C. Montgomery. *Introduction to linear regression analysis*. eng. 5th ed.. Wiley series in probability and statistics ; 821. Hoboken, NJ: Wiley, 2012. ISBN: 9780470542811.

[32] Michael Mozer. “A Focused Backpropagation Algorithm for Temporal Pattern Recognition”. In: 3 (Jan. 1995).

[33] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[34] C. Olah. *Neural Networks, Manifolds, and Topology*. http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/. 2014.

[35] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[36] Jaideep Pathak et al. “Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach”. In: *Physical Review Letters* 120.2 (2018), p. 024102.

[37] Joseph Pedlosky. *Ocean circulation theory*. Springer Science & Business Media, 2013.

[38] Mads B. Poulsen, Markus Jochum, and Roman Nuterman. “Parameterized and resolved Southern Ocean eddy compensation”. In: *Ocean Modelling* 124 (2018), pp. 1–15.

[39] Carl Edward Rasmussen. “Gaussian processes in machine learning”. In: *Advanced lectures on machine learning*. Springer, 2004, pp. 63–71.

[40] Sune O Rasmussen et al. “A stratigraphic framework for abrupt climatic changes during the Last Glacial period based on three synchronized Greenland ice-core records: refining and extending the INTIMATE event stratigraphy”. In: *Quaternary Science Reviews* 106 (2014), pp. 14–28.

[41] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), p. 533.

[42] Hava T Siegelmann and Eduardo D Sontag. “Turing computability with neural nets”. In: *Applied Mathematics Letters* 4.6 (1991), pp. 77–80.

[43] `skopt`. *Scikit-Optimize - Simple and efficient minimization of black-box functions.* https://scikit-optimize.github.io/. 2018.

[44] J.C. Sprott. “A simple chaotic delay differential equation”. In: *Physics Letters A* 366.4 (2007), pp. 397–402. ISSN: 0375-9601. DOI: https://doi.org/10.1016/j.physleta.2007.01.083. URL: http://www.sciencedirect.com/science/article/pii/S0375960107002848.

[45] Steven H Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. CRC Press, 2018.

[46] Ilya Sutskever, James Martens, and Geoffrey E Hinton. “Generating text with recurrent neural networks”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 1017–1024.

[47] `torsk`. *Echo State Network Implementation*. `https://github.com/nmheim/esn`. 2018.

[48] David Verstraeten et al. “Memory versus non-linearity in reservoirs”. In: *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE. 2010, pp. 1–8.

[49] Wikipedia. *Convolutional neural networks*. `https://en.wikipedia.org/wiki/Convolutional_neural_network`. Accessed: 2018-07-10. 2018.

[50] Christopher KI Williams and Carl Edward Rasmussen. “Gaussian processes for regression”. In: *Advances in neural information processing systems*. 1996, pp. 514–520.

[51] Ronald J Williams. “Training recurrent networks using the extended Kalman filter”. In: *Neural Networks, 1992. IJCNN., International Joint Conference on*. Vol. 4. IEEE. 1992, pp. 241–246.

[52] Ronald J Williams and David Zipser. “Experimental analysis of the real-time recurrent learning algorithm”. In: *Connection Science* 1.1 (1989), pp. 87–111.

[53] Peter R Winters. “Forecasting sales by exponentially weighted moving averages”. In: *Management science* 6.3 (1960), pp. 324–342.