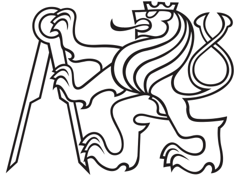
Bachelor Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Reservoir Computing Framework in Apache Flink

Hynek Noll

Supervisor: Sebastián Basterrech, MSc., Ph.D.
Field of study: Software
May 2020

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Noll** Jméno: **Hynek** Osobní číslo: **425233**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Reservoir Computing Framework v Apache Flink

Název bakalářské práce anglicky:

Reservoir Computing Framework in Apache Flink

Pokyny pro vypracování:

Apache Flink is an open source framework and distributed processing engine for manipulations of unbounded and bounded data streams. The framework is ideal for operating large streaming time-series. At the beginning of 2000s, a new computational concept for designing and training Neural Networks was introduced and is popularly known with the name of Reservoir Computing (RC).
A RC model is designed with basic sparse matrices operations. In spite of its simplicity is very powerful for modeling time-series. In this thesis, the student should implement in the Apache Flink framework the Echo State Model (ESN) that is the canonical and most popular RC method. The developed framework should be evaluated over well-known real and simulated time-series provided by the supervisor.

Seznam doporučené literatury:

[1] Apache Flink tutorials available at: https://ci.apache.org/flink
[2] H. Jaeger and H. Haas, “Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication,” Science, vol. 304, no. 5667, pp. 78–80, 2004.
[3] H. Jaeger, M. Lukosevicius, D. Popovici, and U. Siewert, “Optimization and applications of Echo State Networks with leaky-integrator neurons,” Neural Networks, vol. 20, no. 3, pp. 335–352, 2007.
[4] D. Verstraeten, B. Schrauwen, M. D’Haene, and D. Stroobandt, “An experimental unification of reservoir computing methods,” Neural Networks, vol. 20, no. 3, pp. 287–289, 2007.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Sebastian Basterrech, MSc., Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **20.09.2020**

Sebastian Basterrech, MSc., Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my parents (especially my mom) for supporting me throughout the whole studies, both mentally and financially.

Furthermore, I would like to thank my supervisor for investing a lot of time and patiently guiding me through the whole work. And for explaining concepts to me in an easily understandable manner.

Declaration

I hereby declare that I have worked on this thesis independently and specified all the used information sources in accordance with the Methodical guidelines about following ethical principles during the preparation of university theses.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstract

Stream processing for Machine Learning has become popular with the need to analyze large amounts of data in real-time. The focus is shifting to scalable solutions using clusters and processing the data in real-time. Recurrent Neural Networks are expensive to train (require large amounts of data). These requirements are reduced when using the Reservoir Computing framework. Reservoir Computing introduces a specific paradigm that the first part (reservoir) of a Recurrent Neural Network is left untrained, and the second part (readout) focuses on linear modelling.

Apache Flink is a scalable stream processing framework. Flink can provide fault-tolerance guarantees (such as exactly-once semantics) and low latency thanks to processing the stream records individually.

We've developed a new, extensible Reservoir Computing library in Apache Flink. In this work, we present the theory behind it and the performed experiments.

Keywords: Apache Flink, Recurrent Neural Networks, Data Streams, Reservoir Computing, Real-Time Processing, Echo State Networks, Time Series Prediction, Stochastic Gradient Descent, Linear Regression

Supervisor: Sebastián Basterrech, MSc., Ph.D.
Department of Computer Science, FEI, VSB-TUO, Ostrava, Czech Republic
Sebastian.Basterrech@vsb.cz

Abstrakt

Zpracovávání datových proudů je populární metodou pro strojové učení díky potřebě analyzovat velké množství dat v reálném čase. Perspektivními se stávají škálovatelná řešení používající počítačové clustery a zpracovávání dat v reálném čase. Rekurentní neuronové sítě vyžadují velké množství dat k trénování. V případě Reservoir Computing frameworku jsou tyto nároky snížené. Reservoir Computing přináší specifický způsob konstrukce rekurentních neuronových sítí, kdy první část (reservoir) je netrénovaná a druhá část (readout) typicky využívá lineárního modelování.

Apache Flink je škálovatelný framework zaměřený na stream processing (zpracovávání datových proudů). Flink umožňuje zvýšenou odolnost vůči chybám (např. pomocí tzv. "exactly-once semantics") a nízkou latenci díky zpracovávání prvků datového proudu individuálně.

V rámci této práce jsme vyvinuli novou, rozšiřitelnou knihovnu Reservoir Computing funkcionalit pro Apache Flink. Představíme teorii v pozadí a provedené experimenty.

Klíčová slova: Apache Flink, umělé neuronové sítě, lineární regrese

Contents

1 Introduction 1
Thesis Organization 2
Nomenclature 2

2 Data Streams 5
2.1 Basic Definitions 5
2.2 Time Series Data 5
2.3 Bounded and Unbounded Streams 6
2.4 Stream Processing Frameworks . . 7

3 Linear Modelling 9
3.1 Machine Learning Context 10
3.2 Training 10
3.2.1 Algebraic Approach 11
3.2.2 Numerical Approach 12
3.2.3 Summary 14
3.3 Testing 14

4 Reservoir Computing 17
4.1 Overview of Recurrent Neural Networks 17
4.2 Reservoir Computing Models . . . 18
4.3 Echo State Network 19
4.3.1 Echo State Property 20
4.3.2 Memory Length 21
4.3.3 Cyclic Reservoirs 21
Cycle Reservoir with Jumps 22

5 Reservoir Computing Framework in Apache Flink 25
5.1 Introduction to Apache Flink . . . 25
5.2 Characteristics of Apache Flink . 26
5.2.1 Concepts 26
5.2.2 Flink Program Anatomy 27
5.2.3 Common Characteristics of the Core APIs 28
Supported Data Types 28
Copying Behavior 29
5.2.4 Operations of the Core APIs 29
Sources 29
Transformations 29
Iterations 30
5.3 Differences of DataStream and DataSet API 31
5.3.1 DataStream API 31
5.3.2 DataSet API 32
5.3.3 Specific Differences 33
5.4 Features 33
Decompression of Input Files 33
Configuration Functions 33
Passing Parameters 34
5.4.1 Notions of Time 34
Event Time Watermarks 34
5.4.2 Windows 35
5.4.3 Keyed Collections 35
5.4.4 State 35
5.4.5 Broadcasting 35
5.4.6 Distributed Cache 36
5.4.7 Debugging 36
5.4.8 Semantic Function Annotations 36
5.4.9 Parallel Computing in the Context of Apache Flink 37
5.4.10 Using DataStream for Matrix Representation 37
Demonstrating the Problems on Matrix Multiplication 37
5.4.11 Efficient Matrix Representation 38
5.5 Implementation Description 38
5.5.1 Project Structure 39
5.5.2 Data Representation 39
5.5.3 Linear Model Functions (Readout) 39
5.5.4 Reservoir 40
5.5.5 Higher-Level Examples 40
Null Values 41
5.5.6 Default Configuration 41

6 Experimental Results 43
6.1 Linear Regression 44
6.1.1 Glaciers 44
6.1.2 CO_2 Emissions 46
6.1.3 $PM_{2.5}$ Outdoor Air Pollution 52
6.1.4 Limitations of Linear Models 54
6.2 Reservoir Computing 54
6.2.1 Glaciers 55
6.2.2 CO_2 Emissions 56
6.2.3 $PM_{2.5}$ Pollution 58
6.3 Sensitivity Analysis of Reservoir Parameters 58
6.3.1 Reservoir Size 59
6.3.2 Spectral Radius 64
6.3.3 Reservoir Topology 65
6.4 Controlling Spectral Radius 70

7 Conclusions and Future Work **73**
7.1 Conclusions 73
7.2 Recommendation for Future Extensions 73
Bibliography **75**
Acronyms **83**
Acronyms **83**

Figures

4.1 Simple Cycle Reservoir 21
4.2 Cycle Reservoir with Jumps 23
4.3 Cycle Reservoir with Jumps (Randomized Weights) 23

5.1 Flink Dataflow 27
5.2 Flink Iterations Diagram 31

6.1 Glaciers Meltdown (Training Data) 45
6.2 Glaciers Meltdown 45
6.3 Glaciers Meltdown (80% data for training) 46
6.4 CO_2 Emissions By Nation (Training Data) 47
6.5 CO_2 Emissions By Nation LR .. 47
6.6 CO_2 Emissions of United Kingdom (Training Data) 48
6.7 CO_2 Emissions of United Kingdom LR 49
6.8 CO_2 Emissions of Norway (Training Data) 49
6.9 CO_2 Emissions of Norway LR .. 50
6.10 CO_2 Emissions of Czech Republic (Training Data) 50
6.11 CO_2 Emissions of Czech Republic LR 51
6.12 CO_2 Emissions of Mainland China (Training Data) 51
6.13 CO_2 Emissions of Mainland China LR 52
6.14 $PM_{2.5}$ Pollution in Seattle Area (Training Data) 53
6.15 $PM_{2.5}$ Pollution in Seattle Area LR 53
6.16 'Enhanced Identity' (Combined) 54
6.17 Glaciers Meltdown 55
6.18 Glaciers Meltdown 56
6.19 CO_2 Emissions of CHINA (MAINLAND) 56
6.20 CO_2 Emissions of United Kingdom 57
6.21 CO_2 Emissions of Norway 57
6.22 CO_2 Emissions of Czech Republic 58
6.23 $PM_{2.5}$ Pollution in Seattle Area 58
6.24 Analyzing the impact of reservoir size using CO_2 Emissions of UNITED KINGDOM data. 60
6.25 Analyzing the impact of (larger) reservoir size using CO_2 Emissions of UNITED KINGDOM data. 60
6.26 Analyzing the impact of reservoir size using Glaciers Meltdown data. 61
6.27 Analyzing the impact of (larger) reservoir size using Glaciers Meltdown data.................. 61
6.28 Analyzing the impact of reservoir size using Mackey-Glass Time Series data. 62
6.29 Analyzing the impact of (larger) reservoir size using Mackey-Glass Time Series data. 62
6.30 Analyzing the impact of reservoir size using $PM_{2.5}$ Pollution in Seattle Area data. 63
6.31 Analyzing the impact of (larger) reservoir size using $PM_{2.5}$ Pollution in Seattle Area data. 63
6.32 Analyzing the impact of spectral radius using Mackey-Glass Time Series data 64
6.33 Analyzing the impact of spectral radius using CO_2 Emissions of UNITED KINGDOM data. 64
6.34 Analyzing the impact of spectral radius using Glaciers Meltdown ... 65
6.35 Analyzing the impact of spectral radius using $PM_{2.5}$ Pollution in Seattle Area data. 65
6.36 Analyzing the impact of changing the pattern of connectivity using Mackey-Glass Time Series data ... 66
6.37 Analyzing the impact of changing the pattern of connectivity using CO_2 Emissions of UNITED KINGDOM data. 66
6.38 Analyzing the impact of changing the pattern of connectivity using Glaciers Meltdown 67

Tables

6.39 Analyzing the impact of changing the pattern of connectivity using $PM_{2.5}$ Pollution in Seattle Area data. 67
6.40 Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using Mackey-Glass Time Series data 68
6.41 Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using CO_2 Emissions of UNITED KINGDOM data. 68
6.42 Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using Glaciers Meltdown 69
6.43 Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using $PM_{2.5}$ Pollution in Seattle Area data. 69
6.44 CRJ with $\alpha = 0.1$ 70
6.45 CRJ with $\alpha = 0.4$ 70
6.46 CRJ with $\alpha = 0.6$ 71
6.47 CRJ with $\alpha = 0.9$ 71

Chapter 1

Introduction

In today's world, there is a huge amount of data (called *big data*) that can't be processed in a traditional way. At the same time, an increasing amount of computing power is available, which allows for ambitious new paradigms and architectures to arise. One such paradigm, stream processing, is about processing data that arrives rapidly in real-time. (Near) real-time processing is achieved by processing individual (small) *records* of data. This approach may require benevolence in terms of data orderliness (allowing records to arrive out-of-order and late within some fixed time threshold, with the actual order given by timestamps associated with each record). Typically, stream processing uses parallel and distributed computing (running on clusters and multiple threads) to provide scalability. This fact then imposes further requirements on the system such as solid fault tolerance.

In the area of machine learning, recurrent neural networks are used for various prediction and classification tasks. They mimic brain in basic contours and might be difficult to configure properly. A simpler way of constructing recurrent neural networks was introduced under the name Reservoir Computing [1, p. 2]. It allows for a faster learning with the paradigm being more accessible. Therefore, both reservoir computing and stream processing is suitable for problems where we need the results quickly.

Apache Flink is a stream processing framework. It is made to be scalable, provide low-latency and be fault-tolerant. Flink has historically had a machine learning library entitled *FlinkML*. The library includes some algorithms for supervised (such as multiple linear regression) and unsupervised learning, data preprocessing, etc. It is written in Scala and only for the DataSet API. This library is no longer a part of the distribution since version 1.9 [2]. The development was mostly halted and the library wasn't keeping up with the rest of the framework.

We've developed a library for Apache Flink (1.8) focusing on Reservoir Computing, specifically Echo State Networks that are probably the most widely used type. The library is implemented in the popular Java language, and compatible with both DataSet and DataStream API. Although we'll demonstrate the library usage only on time series predictions, it can potentially be used in many other different areas. The present parameters of the model are fully configurable. In our implementation, we've also included some

popular extensions or modifications of the standard Echo State Network. Namely, we've focused on selected deterministic topologies of the reservoir like "Cycle Reservoir with Jumps" [3]. The linear readout can be trained by either Stochastic Gradient Descent or Ridge Regression (using Moore-Penrose inverse (pseudoinverse) with Tikhonov Regularization). Apart from example datasets that were used for experiments and processed solely as historical data (mainly for convenience, as it was slightly easier to produce graphs and other output), we've also attempted to simulate a real-time streaming application, by periodically writing data to and reading it from file. We present a performance analysis of the model over the developed examples, analyzing the impact of individual hyperparameters.

A JavaDoc is included with the developed library. The whole project can be found on GitHub[1] with plotting scripts in a separate repository[2].

Thesis Organization

The next Chapter presents the main concepts related to data streaming and time series. In Chapter 3, we thoroughly describe Linear Regression which is applied as a part of Reservoir Computing. In Chapter 4 we introduce the concepts of Reservoir Computing and put it into the context of Machine Learning. We then focus on Echo State Networks in Section 4.3. In Chapter 5, we'll first introduce the Apache Flink framework, and after that in Section 5.5, we'll zoom in on our implementation. In Chapter 6, we present the experiments testing the validity and possibilities of our implementation. Using both real-world and function-generated data. Lastly, we summarize the presented work and outline the possibilities of future extension in Chapter 7.

Nomenclature

Vectors are denoted by a bold, lowercase letter. Matrices are denoted by a normal, uppercase letter. We'll consider all vectors to be column vectors (by default) and omit their column dimension (which is 1). For example, an n-dimensional vector would be denoted as belonging to $\mathbb{R}^n$ instead of $\mathbb{R}^{n\times 1}$. We might write out a vector as an n-tuple, which will be considered equivalent to the matrix notation, e.g. $\boldsymbol{u} = (u_1, \ldots, u_n) = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \in \mathbb{R}^n$.

Some terms will appear several times, also in case of Apache Flink the terms can have a meaning that differs from a strict mathematical terminology. In order to avoid ambiguity, we specify them as follows:

- *Mapping*: used when referring to a "general type of function", i.e. $f : \mathbb{R}^m \to \mathbb{R}^n$.

[1] https://github.com/h4nek/flink-rc

[2] https://github.com/h4nek/flink-rc-plotting

- *Transformation*: a particular type of mapping where input and output domains are the same, i.e. $\mathbb{R}^n \rightarrow \mathbb{R}^n$. When applying a scalar function to a higher-dimensional structure (vector) and using it as a transformation, it is to be understood as an element-wise application.
- *Function*: by functions we are referring to mappings whose outputs are scalars, *i.e.* $\mathbb{R}^n \rightarrow \mathbb{R}$.

A list of used acronyms can be found at the end of thesis, after bibliography.

Chapter 2

Data Streams

2.1 Basic Definitions

Data is a collection of information in an electronic form that is being stored and used by a computing unit (we might say *data point*, *event* or *data sample* when referring to a single piece of data) [4]. A *computing unit* is a group of resources with close physical relationship that is capable of executing a task. We can think of it as a generalization of a computer [5, p. 112].

Sometimes we might want to impose order on the data, and it then becomes *sequential* [6]. For example, a set of natural numbers can be ordered by the numbers' values and if we were to process it as a stream (see below), we would go from the lowest to the highest. *Time series data* is a subcategory of sequential data where the order of the data is dependent on their timestamp, i.e. the time of their creation or time of entering a data processing application. In practice, we are only able to process discrete data and any continuous sequence (e.g. real numbers) has to be discretized.

Then, a *stream of data* (also known as *data stream* or simply stream) is a sequence of data that is meant to be transferred [7]. It is typically (roughly) ordered by a timestamp marking the creation of each data point, so it can be seen as a type of time series data. *Data streaming* means transferring a stream of data in a continuous flow from one place to another [8, 9]. *Stream processing* is simply processing a data stream by some computing unit [10]. This stands in contrast to *batch processing*, which means processing a collection of data in "batches" and is preceded by storing and aggregating the data. *Real-time data processing* means processing the data as soon as it arrives and producing an output almost instantaneously [11, 12]. Stream processing is often associated with real-time processing since at the heart of it, there is no need to wait for additional data when some data arrives, and it therefore allows for a lower latency [13].

2.2 Time Series Data

More formally, a time series data is a sequence of events obtained through measurements (observations) over time [14, section 2.1]. We typically receive

timestamped data with a fixed time gap in-between individual data points (events) (i.e. receiving a data sample every $k > 0$ units of time). Most commonly, the event at time t may have an impact on the events at time $t + ik$ $(k > 0, i \in \mathbb{N})$, i.e. the events are dependent on each other [15]. For the purpose of our implementation, let's formally define a data stream as an ordered set of values (vectors) $\{\boldsymbol{x}(1), \boldsymbol{x}(2), \ldots, \boldsymbol{x}(n)\}$. Having another such set $\{\boldsymbol{y}(1), \boldsymbol{y}(2), \ldots, \boldsymbol{y}(n)\}$, we can say that their elements form input-output pairs $(\boldsymbol{x}(i), \boldsymbol{y}(i))$. The purpose of this pairing can be seen later when talking about Linear Regression (3) and Reservoir Computing (4).

Usage. The most common problems in the area of Reservoir Computing that involve time series data are:

- Classification: where the system receives a stream, and the goal consists of classifying each element by assigning it to some class (category, label). This is commonly used in areas such as speech recognition (assign a voice to a person), medicine or intrusion detection [16, p. 116].

- Forecasting (Prediction): we typically want to *predict* (estimate) future data given data from the past (and present). This process is called *multi-step ahead time series prediction* (a special case would be the basic 1-step ahead) [17, pp. 765-766]. (We'll refer to it as *x-time-step ahead* in the implementation.) Symbolically

$$\boldsymbol{x}_{t-p}, \ldots, \boldsymbol{x}_t \rightarrow (\boldsymbol{x}_{t+1}, \ldots, \boldsymbol{x}_{t+tsa-1},)\ \boldsymbol{x}_{t+tsa}, \tag{2.1}$$

  where $\boldsymbol{x} \in \mathbb{R}^n$ is a data vector, $t \in \mathbb{N}$ is the current time-step, p is the number of past data used for the prediction and $tsa \in \mathbb{N}$ is the number of time-steps ahead we want to predict.

  This can be used in a wide variety of fields, such as agricultural productivity, stock prices or electricity consumption [17, p. 765].

In general, we want to predict some information that will (or might) be known to us in the future. While learning to predict through already available information from the past.

2.3 Bounded and Unbounded Streams

Since stream is a sequence of data, it has a sense of orderliness of its elements. A bounded data stream is simply a stream that has a defined start and end [18]. Similarly, an unbounded data stream is a stream that doesn't have a defined end, but it still does have a defined start. This is understandable considering how stream processing works. We have to start processing at some point (typically rooted in time) and work onward from there. Therefore, what was before that point (if anything) becomes unavailable. Yet, we can theoretically process the data forever, or we might not know when we will want to stop for now. Unbounded streams are also called never-ending due

to this characteristic. There are some practical implications that come with processing a potentially infinite amount of data. First of all, we can't wait until all of the data arrives, so we have to process them on the run. We have to produce some partial results, typically periodically. Second of all, we can't just store all of the historical data, as it would grow in size infinitely. We have to manage what we keep and potentially discard data that are older than some chosen period of time. Having the data relatively ordered might also be crucial in order to have a sense of completeness of the partial results as well as responsibly manage old data and create reasonable backups. On the other hand, bounded data streams can be processed after aggregating all data. They can also be sorted as a whole afterwards and therefore can be allowed to arrive in a chaotic fashion.

2.4 Stream Processing Frameworks

If we look at open source frameworks working with big data using streams, we are offered tools like Apache Spark, Apache Flink, Apache Kafka, Apache Storm and other less popular ones [19, p. 1] [1]. All of the mentioned solutions are distributed frameworks focused on cluster processing.

Apache Flink is slightly newer, therefore slightly less developed and more experimental big data processing framework compared to Apache Spark. It makes use of true streams (incoming data are immediately queued for stream processing) and is generally faster (lower latency) [20, section 5: Results]. Apache Spark processes the given data in groups of elements (micro-batches) [21].

Apache Kafka is an earlier started project focused on distributed cluster processing, where distributed coordination is done with nodes being equal (consumers form consumer groups) and therefore it has to have a solid fault tolerance [22]. In Flink, on the other hand, there is a master-slave hierarchy, although fault tolerance is also an important aspect [23, section Distributed Execution]. Kafka is often used as a source that creates streams and then forwards them to other stream processing frameworks [24].

We'll work with Apache Flink, which is in active, ever-growing development, whilst already having a stable release.

[1] A collective infographic of big data solutions can be found here: `http://mattturck.com/wp-content/uploads/2018/07/Matt_Turck_FirstMark_Big_Data_Landscape_2018_Final.png` (Accessed: 20. May 2020)

Chapter 3

Linear Modelling

Linear regression (LR) (the most common way of linear modelling) tries to find the best *linear model* (LM) according to some criteria (typically using the *least squares method*, described later) [25, p. 48]. A linear model takes input variables x_i (observations, independent variables), uses them in a linear transformation (more specifically, a function that is linear with respect to its coefficients α_i) and produces the output variables y_i (dependent variables), or more precisely their predictions (estimations) $\hat{y}_i$ that are as close as possible (have the lowest chosen error). In other words, there has to be a linear relationship between the input and the output variables for the method to be applicable [26, pp. v, 2-3]. (In the joint input-output space, the set of input-output pairs generally represents a(n affine) hyperplane.)

We'll narrow the view to only having one output variable y and predicting the respective scalar $\hat{y}$. More specifically, we'll work with time series (described in the above section 2.2), where the input and output variables can be viewed as functions of time t. Note that we'll use t here as a label, denoting that $(\boldsymbol{x}(t), y(t))$ is an input-output pair where in reality, $y(t)$ is typically from a later time. The *linear predictor function* (representing the theoretical linear model) has the following form:

$$y(t) = \alpha_0 + \alpha_1 x_1(t) + \ldots + \alpha_n x_n(t) + \epsilon, \quad t = 1, \ldots, m \tag{3.1}$$

In practice, the following equation is used instead:

$$\hat{y}(t) = \alpha_0 + \alpha_1 x_1(t) + \ldots + \alpha_n x_n(t), \quad t = 1, \ldots, m \tag{3.2}$$

which can be shortened using a matrix notation into:

$$\hat{\boldsymbol{y}} = X\boldsymbol{\alpha} \tag{3.3}$$

where $\hat{\boldsymbol{y}} \in \mathbb{R}^m, \boldsymbol{\alpha} \in \mathbb{R}^{n+1}, X \in \mathbb{R}^{m \times (n+1)}$. While the first equation (3.1) includes an unobservable error ϵ and shows us the actual relationship between input and output variables, the second equation (3.2) gives us a computable estimate $\hat{y}(t)$ of the real output value (we will call it the prediction) at one point in time. To be precise, we should also use $\hat{\boldsymbol{\alpha}}$, since it's an estimate of the theoretical model parameters, but we'll omit this notation for brevity.

(A verbal description of the function (3.2) is that in one fixed point in time t, it provides a scalar prediction (of an output variable) $\hat{y}(t)$ based on an

n-dimensional vector (of input variables) $\boldsymbol{x}(t)$, using an (n+1)-dimensional vector (of parameters/coefficients) $\boldsymbol{\alpha}$.) Note that $X = \begin{bmatrix}\mathbf{1} & \boldsymbol{x}_1 & \dots & \boldsymbol{x}_m\end{bmatrix}$, where $\mathbf{1}$ is a vector of 1s that will get multiplied by the so-called *y-intercept* (the constant) α_0. Y-intercept is the value of $\hat{y}(t)$ when $\boldsymbol{x}(t) = \mathbf{0}$ ($\forall t$). Because of accepting multiple input variables, this process is also more specifically called the *multiple linear regression* [26, pp. 2, 3]. If we'd like to predict multiple values at the same time t, we can just create a separate linear model for each of them.

3.1 Linear Regression in the Context of Machine Learning

In machine learning (supervised learning), there exists a *training* phase and a *testing* phase of a model. (We could also have a *validation* phase/set if we were choosing between multiple models [27, p. 7].) In the case of linear regression, the training phase aims to create the "best" (according to the chosen criteria) linear regression model by computing the optimal $\boldsymbol{\alpha}$ vector of parameters (*regression coefficients*) [26, p. v]. The testing phase is simply about evaluating the computed model's performance by applying it on input vectors $\boldsymbol{x}(t)$ (usually different from the training data), obtaining the corresponding prediction value $\hat{y}(t)$ and comparing it to the actual value $y(t)$ using the chosen criteria. In both phases, X and $\boldsymbol{y}$ are available to us. In production, the testing phase is substituted with (or a part of) the model usage and the real outputs $y(t)$ typically come to us with a delay, therefore we can evaluate the performance retroactively.

If we only have one set of data available at the beginning, the data should be split (e.g. in half) to have disjoint datasets for the training and testing phase. This can also prevent *overfitting*, because we might otherwise create a function that fits perfectly, but only for the set it was trained on. And this would not work so well in the future as the data probably follows a more complex/random pattern.

Taking the data through a linear model that is typically created using linear regression is the final phase of a reservoir computing framework (discussed in the next chapter) [1, pp. 3, 5, 20].

3.2 Training

Under ideal conditions, what we'd try to achieve is to create a model that outputs perfect predictions (such that $\hat{\boldsymbol{y}} = \boldsymbol{y}$).

In reality, we can hardly predict anything perfectly, let alone when requiring a linear relationship. Therefore, we need to choose some criteria which will tell us how good our model is. The most common approach is to minimize the distance between our predictions and the actual values (there are other

approaches which we won't discuss). This can be written mathematically as:

$$\arg\min_{\hat{\boldsymbol{y}}} ||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2 \tag{3.4}$$

It's an optimization problem called the *method of (linear) least squares*. One reason to square the norm (put it to the second power) is that the (euclidean) norm itself is defined as

$$||\boldsymbol{y}||_2 = \sqrt{\sum_{i=1}^{m} y_i^2} \tag{3.5}$$

where y_i are the coordinates of vector $\boldsymbol{y}$ (with respect to the standard basis). And squaring non-negative real numbers means applying a monotonically increasing function on them, which preserves the argument of minimum. Therefore squaring the norm just makes the computation easier while giving us the same result (since we're mainly interested in the argument of minimum, not the value (which we could easily compute afterwards)). Another reason will be apparent when using numerical methods in subsection 3.2.2: the squared norm gives us a smoother function. This in effect means that we'll get a better convergence overall (the function approaches minimum faster when farther from it and becomes smoothly slower when closer to it). So it might get us close to the minimum faster while also making us less prone to overshooting it. [28, p. 25]

Since we know what the function $\hat{\boldsymbol{y}}$ (3.3) looks like and that it depends solely on the $\boldsymbol{\alpha}$ vector of parameters (X is given to us in the training phase), we can rewrite the equation (3.4) as

$$\arg\min_{\boldsymbol{\alpha}} ||X\boldsymbol{\alpha} - \boldsymbol{y}||^2. \tag{3.6}$$

So in the training phase, our goal is to compute (get close to) the optimal $\boldsymbol{\alpha}$ (sometimes denoted as $\boldsymbol{\alpha}^*$). There are multiple ways to do that which can be divided into two major types. First type is an algebraic (analytical) approach and second type is a numerical one.

3.2.1 Algebraic Approach

An algebraic approach effectively means trying to solve a set of linear equations

$$\boldsymbol{y}\ (= \hat{\boldsymbol{y}}) = X\boldsymbol{\alpha}, \tag{3.7}$$

which aren't expected to have an exact solution.

Since the gradient of a (differentiable) function has to equal $\mathbf{0}$ at the argument of minimum, we get the following equations from the least squares method function (3.6):

$$||X\boldsymbol{\alpha} - \boldsymbol{y}||^2 = (X\boldsymbol{\alpha} - \boldsymbol{y})^T(X\boldsymbol{\alpha} - \boldsymbol{y}) = \boldsymbol{\alpha}^T X^T X\boldsymbol{\alpha} - 2\boldsymbol{y}^T X\boldsymbol{\alpha} + \boldsymbol{y}^T\boldsymbol{y}. \tag{3.8}$$

$$\nabla||X\boldsymbol{\alpha} - \boldsymbol{y}||^2 = \frac{\partial||X\boldsymbol{\alpha} - \boldsymbol{y}||^2}{\partial\boldsymbol{\alpha}} = 2X^T X\boldsymbol{\alpha} - 2X^T\boldsymbol{y} = \mathbf{0} \tag{3.9}$$

$$X^T X\boldsymbol{\alpha} = X^T\boldsymbol{y} \tag{3.10}$$

The equation (3.10) is called the *normal equation*.

Now suppose that X has linearly independent columns (its columns, viewed as vectors, form a linearly independent set). Then X^TX is invertible and we are able to write an equation for computing the $\boldsymbol{\alpha}$ directly using a *Moore-Penrose inverse* (specifically the left inverse of X):

$$\boldsymbol{\alpha} = (X^TX)^{-1}X^T\boldsymbol{y}. \tag{3.11}$$

But in general, there might be cases where X has linearly dependent columns or where X^TX is ill-conditioned (that means we're expected to meet with a great inaccuracy when computing its inverse). Then we can introduce a regularization factor $\lambda \in \mathbb{R}_0^+$ (using *Tikhonov regularization*) and have a new equation

$$\boldsymbol{\alpha} = (X^TX + \lambda I)^{-1}X^T\boldsymbol{y} \tag{3.12}$$

where $I \in \mathbb{R}^{n\times n}$ is the identity matrix. This is also known as *ridge regression*. λ is called the *hyperparameter* [29, p. 281], since we will not optimize it as part of the training but will choose it beforehand, usually by trial and error. We can see that for $\lambda = 0$, this new equation is the same as the previous (3.11). Otherwise if $\lambda > 0$, then $X^TX + \lambda I$ is always invertible (in theory). The larger our parameter λ is, the more well-conditioned the matrix becomes (and the model is less prone to overfitting). The smaller it is, the closer we are to the real solution (the global minimum argument). So we need to balance the λ to best fit our concrete problem.

3.2.2 Numerical Approach

In numerical approach, we use iterative methods to converge to the optimal $\boldsymbol{\alpha}$. One such method is the *gradient descent* (GD). The idea is to start at some initial $\boldsymbol{\alpha}_0$ and go in the direction of the negative gradient. Since the gradient vector points in the direction of the steepest ascent, its negative counterpart will lead us in the direction of the steepest descent, geometrically speaking. Therefore we'll approach a local minimum (or a saddle point) with each step (if the step we take is small enough). In practice, we don't ever expect to arrive exactly at the optimum (especially with the numerical approach), so the important condition for us to ensure is that the function value is decreasing with each step (by choosing the right step size) and we therefore get better results (lesser error).

Gradient descent can be further divided into *batch gradient descent* and *stochastic gradient descent* (SGD). Batch gradient descent takes the whole training dataset (batch) for each iteration. Conversely, stochastic gradient descent only uses one record from the training set in each iteration, eventually iterating over the whole set [30, p. 2]. This record might be chosen at random (by shuffling the training set first), hence the name "stochastic". The formula for one iteration of the batch gradient descent for the linear least squares minimization (3.6) is

$$\boldsymbol{\alpha}_{k+1} = \boldsymbol{\alpha}_k - \eta\nabla(||X\boldsymbol{\alpha}_k - \boldsymbol{y}||^2) \tag{3.13}$$

$$\boldsymbol{\alpha}_{k+1} = \boldsymbol{\alpha}_k - 2\eta X^T(X\boldsymbol{\alpha}_k - \boldsymbol{y}) \tag{3.14}$$

where $\eta \in \mathbb{R}_0^+$, called the *learning rate*, directly affects the step size. It should be chosen so that we approach the minimum as quickly as possible without "overshooting" and getting a larger function value than the one from the previous iteration. The constant 2 can be seen as part of the learning rate and ignored. Furthermore, it's common to divide the gradient by the number of samples (training set size), which we'll denote by m. This makes the change of $\boldsymbol{\alpha}$ independent of the size of the dataset, which would otherwise have to be accounted for by modifying the learning rate [31]. The edited formula would now be

$$\boldsymbol{\alpha}_{k+1} = \boldsymbol{\alpha}_k - \eta\frac{X^T(X\boldsymbol{\alpha}_k - \boldsymbol{y})}{m}. \tag{3.15}$$

We can adjust the learning rate in each iteration and control it so that we don't ever increase the function value. A good initial estimate of $\boldsymbol{\alpha}_0$ is also desirable to get close to the minimum more quickly.

If $\nabla(||X\boldsymbol{\alpha}_k - \boldsymbol{y}||^2) = 0$, it would mean we're at some stationary point. Generally, the stationary point might even be a (local) maximum, but if we ensure a decreasing tendency, then we'd only end at a maximum if we picked it as our starting point ($\boldsymbol{\alpha}_k = \boldsymbol{\alpha}_0$). Otherwise, the stationary point might be a saddle point or a (local) minimum. We can infer more properties in specific cases. If the function being minimized is convex and differentiable, then any stationary point is a global minimum [32, pp. 652, 653]. But we expect the function to be differentiable in order to compute its gradient. And then since (any) norm is convex (and power of 2 is convex and non-decreasing), our function (3.6) therefore has the needed properties.

The formula for one iteration of the stochastic gradient descent for the linear least squares minimization is

$$\boldsymbol{\alpha}_{k+1} = \boldsymbol{\alpha}_k - \eta\frac{(\boldsymbol{\alpha}_k^T\boldsymbol{x}(k) - y(k))\boldsymbol{x}(k)}{m} \tag{3.16}$$

where we've already left out the constant 2 and divided by the number of samples m. We don't need to have the whole dataset upfront. The model improves with every (individual) record it receives. Compared to the batch version, first of all, the updates are faster. And it's also useful for cases where the whole training dataset might not fit into memory. The disadvantage might be that the function (and therefore its $\arg\min$) keeps changing [30, p. 2].

Now the final question is when to stop iterating. We can set a proximity threshold and stop the algorithm when we're close enough. We can also set the maximum number of iterations (or do a combination of both). By doing a combination of both, we can try to get satisfactorily close while not getting stuck in an endless computation. By default, we'll just iterate over the whole dataset once.

Setting the Learning Rate. The simplest approach is tuning the value by "trial and error" while setting the learning rate η constant. If we wanted an

optimal learning rate (at each step), there exists what is called the *line search*, where we basically compute the global minimum of a real-valued function ($f : \mathbb{R} \to \mathbb{R}$). In practice, this would mean computing the derivative at each iteration, instead of just applying a learning rate "blindly", and is typically too expensive.

An interesting approach that might provide a good compromise between precision and computational cost is to introduce a specified *decay*. That is, an initially chosen learning rate gets smaller according to some criteria. This cooperates nicely with the fact that as we get closer to a minimum, we'll need to perform smaller steps to prevent overshooting. One type of decay, which we implemented, is the *step decay*, where we simply lower the value every couple of steps. The questions of how often (*decayGranularity*) and how much (*decayAmount*) should the learning rate change, can again be approached through (perhaps informed) experimentation and set manually. We apply the formula

$$\eta = \eta_{previous} * (1 - decayAmount) \tag{3.17}$$

every $\lceil \frac{numberOfSamples}{decayGranularity} \rceil$ steps, where $numberOfSamples$ is the size of the training set.

3.2.3 Summary

We've chosen to implement two representative methods to compute the optimal $\boldsymbol{\alpha}$ parameter. First one (algebraic, offline) is by computing the formula (3.12), where we have to set the hyperparameter λ (regularization factor). We used Moore-Penrose inverse with Tikhonov regularization. Second one (iterative, online) is by repeatedly computing the formula (3.16), where we set the hyperparameter η (learning rate). We used stochastic gradient descent.

3.3 Testing

Testing the model (3.2) (created in the training phase) is about first applying it on input vectors $\boldsymbol{x}(t)$ and getting predictions $\hat{y}(t)$. Then we compare the corresponding predictions $\hat{y}(t)$ to the real output values $y(t)$. To realize all that, we need a testing dataset (preferably one that is disjoint from the training dataset, as mentioned earlier). In other words, we're computing some sort of *error*.

There are multiple types of errors we can compute. One of the most common and basic ones is the *mean squared error* (MSE). As the name suggests, we square the difference and then also divide it by the number of samples m to get a result independent of the size of the used dataset.

It has the following formula:

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}(i) - y(i))^2 \tag{3.18}$$

which can be written in a matrix notation as

$$\text{MSE} = \frac{(\hat{\boldsymbol{y}} - \boldsymbol{y})^T \cdot (\hat{\boldsymbol{y}} - \boldsymbol{y})}{m} = \frac{||\hat{\boldsymbol{y}} - \boldsymbol{y}||^2}{m}, \tag{3.19}$$

where m is the number of samples (testing dataset size). We can assume that the lower the value of MSE is, the more precise our model is.

Chapter 4

Reservoir Computing

4.1 Overview of Recurrent Neural Networks

To discuss reservoir computing, we should first put it in a context and define the encompassing terms. The broadest relevant area could be the so called "artificial intelligence". Basically, *artificial intelligence* (AI) can mean any behavior done by machines that resembles the natural human intelligence.

A more specific and more precisely defined field is called *machine learning* (ML). ML strives to make computing units complete tasks (and do them with gradually better results) by learning them in a given bounded domain with specified properties, instead of following the traditional explicit set of instructions (that would be given by the programmer). More exactly, a ML *task* is about finding a relation between an input and an output based on given input-output pairs [1]. This relation can then be used to "predict" the output from only the given input (e.g. receiving a sequence of (not completely random) 0s and 1s and predicting what the next number is going to be). A ML task can be dependent on the input history (*temporal*) or just the input in the current moment (*non-temporal*) [1, pp. 4, 5]. Non-temporal task is thus suitable for problems where individual input-output pairs are unrelated.

In the broadest sense, ML can be seen as an approach to create and train AI, even if just within a very narrow domain (e.g. playing chess). But possibly a better look at it would be that ML is an approach that might get us closer to developing a real general-purpose AI. ML can be divided into supervised and unsupervised. In *supervised learning*, a set of pairs with inputs and desired outputs is given at the very beginning. A specific model is then trained on these pairs whilst the machine knows that it should give out results as close as possible to the desired outputs when it receives the paired input. A common specification of "being close" is having the lowest possible mean squared error (described later; generally, we try to minimize any chosen type of error). Supervised learning is standard in the type of neural networks we're interested in.

An *artificial neural network* (ANN) (commonly just *neural network*) is a system that tries to mimic the biological neural network (neural circuit). In a nutshell, we can see that ANNs are an attempt to make machines simulate human (and animal) brain. Again, we can divide ANNs in subcategories.

One criteria is whether or not they allow cycles. If they do, they're called a *recurrent neural network* (RNN). If not, (they form a directed acyclic graph) they're called *feedforward neural networks*.

We'll be interested in RNNs, since reservoir computing is used to train them. Moreover, it's apparent from the above description that feedforward networks are just a subset of RNNs, so they'll technically not be left out [33].

Some well-known families of RNNs are:

- Vanilla RNN: are the original type of RNN that was used in the past. It had some problems like the *vanishing/exploding gradient*, where over time, thanks to the nature of the used activation (typically sigmoid) functions, the gradient could either vanish (go to 0) or explode (go to ∞) [34, p. 1]. Which meant the network would stop learning or get to NaN values. This basically happens because the weight matrix that the gradient is being repeatedly multiplied by has typically values either < 1 or > 1 [35, p. 194].

- *Long Short-Term Memory* (LSTM): tries to eliminate the vanishing and exploding gradient problems by using gates with logistic functions on top of the basic RNN [36, p. 3]. So it's generally more complex. It also enables forgetting past information through one of the gates (modify the memory length). LSTM is used for problems like speech recognition [36, p. 2].

- Reservoir Computing: started training only the output layer of RNN in a linear fashion, leaving the hidden/input layer untrained (its weight matrices are instantiated before the model starts training and fixed). This all can save us a lot of unnecessary computation. The gradient problems are avoided by operating near the "edge of stability" through scaling the spectral radius of the hidden weights matrix (discussed in 4.3.1) [37].

4.2 Reservoir Computing Models

Reservoir Computing (RC) is an approach to creating RNNs that addresses the problems of classical approaches using mainly a gradient descent method (GD described later). It (typically) imposes the following properties on the training [1, pp. 2, 3]:

First, a *reservoir* (input layer and hidden layer of the RNN, mainly referring to the latter) is created randomly. It then guides the input through nonlinear transformations which can be affected by the previous inputs and "lifts it" into a higher dimension.

Second, a *readout* (output layer) is trained, typically through linear regression to form a linear model. For each output of the reservoir, a linear transformation is applied on it to create the final RNN output. While we could allow the readout output to belong to $\mathbb{R}^n$ in general, we'll only consider it to be a single real value for simplicity (the general output could then be created by putting n real-valued outputs into a vector).

In comparison to classical RNN training methods, RC appears to be less computationally expensive by not having to train the reservoir (only updating its state $\boldsymbol{x}(t)$).

By doing this, it managed to outperform past RNN architectures in various types of prediction and classification tasks [1, p. 3]. For the future, RC mainly showed that we don't yet know of a good way to train RNNs [1, p. 25].

Two major types of RC are echo state networks and liquid state machines. Due to their similarities, the term "reservoir computing" was introduced. *Liquid State Machine* (LSM) follows a more biologically realistic path and finds its usage in the computational neuroscience. It uses *spiking neurons* in the readout. Spiking neurons produce an output ("fire a spike") only once a sum of specific input signals has reached a given threshold [38, pp. 6, 7]. *Echo State Network* (ESN), on the other hand, typically has linear readout. It uses classic neurons, which accept an input and produce an output at each iteration of the network.

4.3 Echo State Network

There isn't one way to implement or express an RC model. We'll be using a simple RNN with one hidden layer, and express the formulas in matrix notation (presented as a state-space model) as is done in [1, pp. 6, 5]. For the reservoir computation (also called an *expansion*), we have

$$\boldsymbol{x}(t) = f(W_{\text{in}}\boldsymbol{u}(t) + W\boldsymbol{x}(t-1)), \quad t = 1, ..., T, \tag{4.1}$$

where $\boldsymbol{u}(t) \in \mathbb{R}^{N_\text{x}}$ is the input vector in time t, T is the size of the processed dataset ($T \to \infty$ for data streams), $W_{\text{in}} \in \mathbb{R}^{N_\text{x} \times N_\text{u}}$ is a matrix of input weights, $W \in \mathbb{R}^{N_\text{x} \times N_\text{x}}$ is a matrix of weights of internal network connections, f is some chosen nonlinear transformation (e.g. hyperbolic tangent) and $\boldsymbol{x}$ is the *state vector* (reservoir output).

We can interpret this as doing a linear mapping of the input data and a linear mapping of the reservoir outputs from the previous time instance. Then adding them together and finally computing some nonlinear transformation of it. Due to this recursive definition, $\boldsymbol{x}(t)$ keeps a nonlinear expansion of the whole input history. We expect that $N_\text{x} \gg N_\text{u}$ [1, p. 5]. Also, $\boldsymbol{u}(t)$ is typically included as part of $\boldsymbol{x}(t)$. (Hence why we also call it an *expansion* - we're expanding $\boldsymbol{u}$ by adding more dimensions to it.) Effectively, the reservoir consists of matrices W_{in} and W.

Both matrices W_{in}, W are instantiated randomly or according to some predefined pattern (the reservoir is untrained). Typically, their elements come from a uniform distribution on the interval $[-0.5, 0.5]$ (more generally an interval symmetric around 0). While W_{in} is dense (with virtually no 0s), W is typically sparse, with sparsity around 80% or more.

We could also work with an extended model

$$\boldsymbol{x}(t) = f(W_{\text{in}}\boldsymbol{u}(t) + W\boldsymbol{x}(t-1) + W_{\text{ofb}}\boldsymbol{y}(t-1)), \quad t = 1, ..., T, \tag{4.2}$$

where W_{ofb} is an "output feedback" weight matrix and $\boldsymbol{y}(t-1)$ is the readout output at previous time instance. So the expansion gets affected by the previous final output value as well.

For the readout computation, we have

$$\boldsymbol{y}(t) = W_{\text{out}}\boldsymbol{x}(t), \tag{4.3}$$

where $W_{\text{out}} \in \mathbb{R}^{N_{\text{y}} \times N_{\text{x}}}$ is the matrix of trained output weights and $\boldsymbol{y}(t) \in \mathbb{R}^{N_{\text{y}}}$ is the final output of RC in time t. Another nonlinear transformation could also be applied on $\boldsymbol{y}(t)$ but that's beyond our scope of interest. Note that we could also view this as a *linear basis function model*, where $\boldsymbol{x}$ would be a vector of results of the basis functions applied on the vector $\boldsymbol{u}$.

To introduce an *intercept* value (also called *bias*), we add a constant (1) as the first coordinate of $\boldsymbol{x}$. If we then set all (other) coordinates of $\boldsymbol{x}$ to 0, the value of the output vector is determined by the 1st column of W_{out}. Geometrically, for function $y : \mathbb{R} \to \mathbb{R}$ dependent on input $x \in \mathbb{R}$, plotted in Cartesian plane, this would be the point at which the line intercepts the y-axis. (The input vector would actually be $\boldsymbol{x} \in \mathbb{R}^2$ when considering the intercept.) Thanks to this, our model doesn't have to map to a linear subspace (doesn't have to include origin in it's range) and is thus more flexible.

We can see that this is a temporal task, even though we do not remember the original input vector $\boldsymbol{u}$. Instead, we keep its nonlinear expansion $\boldsymbol{x}$.

As mentioned before, we'll consider $N_{\text{y}} = 1$ for our implementation.

The most popular approach to computing the readout (more specifically the W_{out} matrix) is using linear regression, which was discussed in detail in chapter 3.

4.3.1 Echo State Property

A desirable feature of an ESN to have is the *echo state property*. It represents a condition that the effect of older states $\boldsymbol{x}(t)$ and inputs $\boldsymbol{u}(t)$ should be getting smaller as time goes by [1, p. 11]. That is, they would have an infinitely small impact on the state $\boldsymbol{x}(t+k)$, where $k \to \infty$.

Typically (but not necessarily), this property is satisfied when the spectral radius of W is less than 1 [1, p. 11].

Spectral radius of a (square) matrix is defined as the largest absolute value of its eigenvalues. Formally

$$\rho(M) = \max\{|\lambda_1|, \dots, |\lambda_n|\}, \tag{4.4}$$

where $\lambda_1, \dots, \lambda_n \in \mathbb{C}$ are the eigenvalues of matrix $M \in \mathbb{R}^{n \times n}$ (real-valued matrices are sufficient for our case).

To ensure that $\rho(W) < 1$ in an implementation, we can simply scale the original matrix (here denoted W^{old}):

$$W = \frac{\alpha}{\rho(W^{\text{old}})} W^{\text{old}}, \tag{4.5}$$

where $\alpha \in (0, 1)$ is a configurable(tonable) hyperparameter [3, pp. 4-5].

If we wanted a truly sufficient condition for ensuring the echo state property, we could replace the spectral radius with the largest singular value of W (singular values are non-negative, so no need to compute their absolute values) [1, p. 11].

4.3.2 Memory Length

Spectral radius also affects another property which is the length of the memory [1, p. 11, 39, p. 17]. *Memory length* of ESN roughly means how long can an information be kept (and be significant) inside the network. That is, for the computation of $\boldsymbol{x}(t+k)$, how large can k be for $\boldsymbol{x}(t)$ to have an impact on it? When the spectral radius is close to 0, we speak about short memory. When it is close to 1, it provides long memory. (Of course, what's short and long can be arbitrary.)

4.3.3 Cyclic Reservoirs

Apart from ensuring the echo state property, we might find other properties to be desirable. Like having cycles inside the reservoir (represented by the W matrix), or having a symmetric W.

Cycles enable that an information (from input $\boldsymbol{u}(t)$) can be stored forever, although typically with a diminishing impact. Otherwise, it has to be lost or "emitted" at some point.

Instead of creating the W randomly, we can do it deterministically. The most basic type is the *Simple Cycle Reservoir* (SCR) [40, p. 132]. Viewing it as a directed graph, we want to have connections going from the first node to the second, etc. and finally from last to the first. Creating one big cycle.

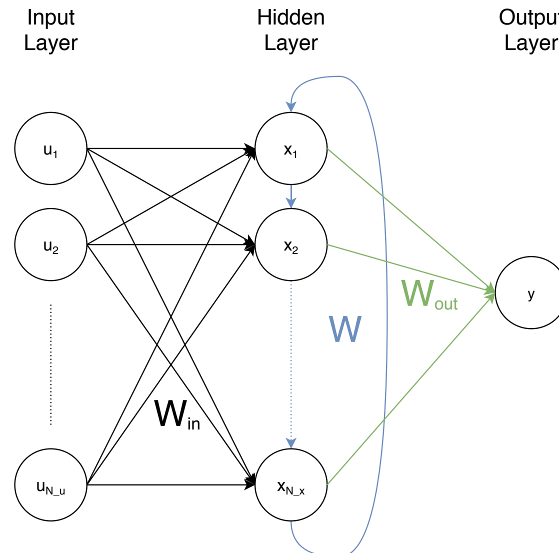


Figure 4.1: Simple Cycle Reservoir. W_{in} is assumed to be 100% dense, which corresponds to a fully connected hidden layer (layer of state vectors $\boldsymbol{x}(t)$) in the graph representation.

Weights are constant. That is, we generate a random weight r_c from some interval and assign it to every connection. This topology corresponds to having a matrix W with this non-zero weight on the subdiagonal, as well as in the top-right corner.

Generally, when considering $W \in \mathbb{R}^{N_x \times N_x}$, this corresponds to having N_x nodes in the reservoir. A non-zero element $W_{i,j}$ then corresponds to a directed edge from j-th to i-th node.

Now it's clear why we have $W_{i+1,i} = r_c \; \forall i$ and $W_{1,N_x} = r_c$ to create the unidirectional cycle.

Cycle Reservoir with Jumps

On top of the simple cycle, we can consider adding bidirectional *jumps* [3]. They effectively work as shortcuts, lowering our average path length. We need to specify the jump size j. The jumps will then connect nodes that are j places apart, starting from the 1st node - ideally forming one (bidirectional) cycle. And once again, we'll have a constant weight r_j for all our jump connections (different from the cycle weight). Then our first jump is represented by $W_{1,1+j} = r_j = W_{1+j,1}$. Other jumps simply follow from the reached nodes until we can't create any more. That is, for the general formula, $W_{[1+(k-1)j],1+kj} = r_j = W_{1+kj,[1+(k-1)j]}, \; k \in 1, \dots, \lfloor \frac{N_x - 1}{j} \rfloor$ and then the last jump that ideally goes to the first node ($W_{[1+N_x-j],1} = r_j = W_{1,[1+N_x-j]}$). This can be ensured by satisfying $j \mid N_x$. This is called the *Cycle Reservoir with Jumps* (CRJ)[1].

W will generally be a sparse matrix, so we don't have to worry about that property. However, due to the cycle, it will not be symmetric.

An obvious alteration that we'll support is to have individually randomized weights, instead of r_c and r_j.

We can think of creating a symmetric matrix with only jumps. In that case, we'll realize a jump from every node. That is, $W_{i,[(i+j) \bmod N_x]} = r_j = W_{[(i+j) \bmod N_x],i} \; \forall i$ and all other elements will be 0. This will automatically create one big (encompassing every node) bidirectional cycle or multiple small ones (*i.e.* when $j \mid N_x$).

[1]Note that we call this topology "Cyclic with Jumps" in the implementation and in the generated plots.

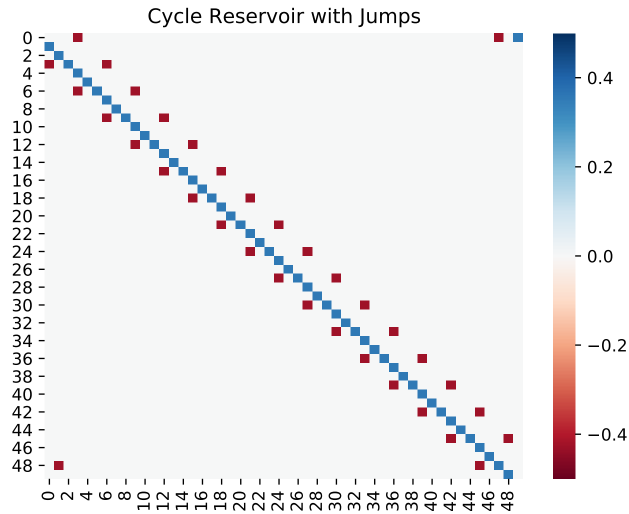


Figure 4.2: A visualization of $W \in \mathbb{R}^{50\times50}$ with random constant cycle weight of approx. 0.3569, jumps of size 3 and random constant jump weight of approx. -0.4235.

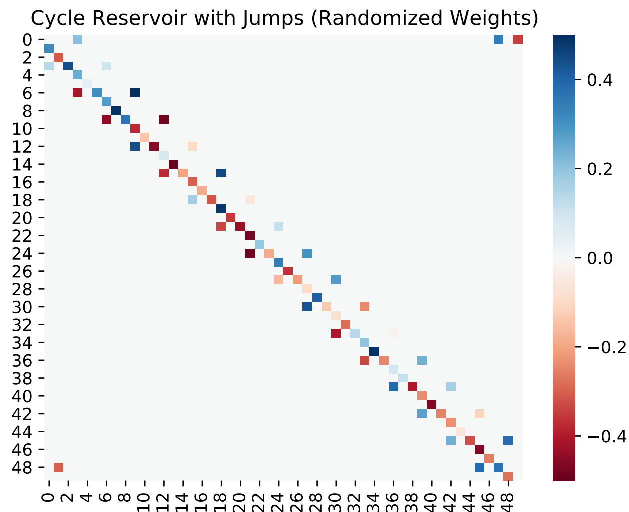


Figure 4.3: A visualization of $W \in \mathbb{R}^{50\times50}$ with jumps of size 3 and random individual weights.

In these figures, we've ignored the scaling hyperparameter α. The comparison of W's with different α values can be found in section 6.4.

The sparsity of such W's can be computed relatively easily. Given N_x nodes, we have N_x weights for the whole cycle and the number of jumps is

$\lfloor \frac{N_\mathrm{x}}{j} \rfloor$:

$$\frac{N_\mathrm{x} + 2\lfloor \frac{N_\mathrm{x}}{j} \rfloor}{N_\mathrm{x}^2} \times 100\ (\%) \tag{4.6}$$

Chapter 5

Reservoir Computing Framework in Apache Flink

5.1 Introduction to Apache Flink

Apache Flink is a framework for distributed stream (and batch) processing of data [18]. It mainly consists of a streaming engine and builds batch processing on top of it [41]. Flink has been developed by The Apache Software Foundation with the purpose of being capable of fast and reliable data processing on a large scale [42].

The data that Flink processes can be either bounded (finite) or unbounded (infinite, never-ending) streams (see Section 2.3 for further description), where bounded streams can be seen as a special case of unbounded streams (we just define an end). Flink provides DataSet API and DataStream API respectively to process them.

Although Flink is community-driven, a major support comes from a company called Ververica (formerly Data Artisans), which is composed of the original developers.

History and Future. Apache Flink's foundational concepts were laid out in 2008 [43]. The development of its direct predecessor named *Stratosphere* started in 2010 through a collaboration between three German universities. The goal of the project was to develop next generation big data analytics platform [44]. (It was built on top of a research project called *Nephele*, developed at the Technical University of Berlin [45]. It introduced building a dataflow through constructing a Job Graph by the users and then turning it into an Execution Graph by the framework.) In 2014, Stratosphere was accepted as an Apache Incubator project (and subsequently renamed to Apache Flink), which enabled it to expand into a more global frame and made it a part of the open-source-focused Apache Software Foundation [46].

On 8. March 2016, Flink 1.0 was released [47]. Flink has been in active development since then, with a major (1.x) release roughly every 4-6 months [48, section History].

The focus of future development is on the Table API and further unifying the batch and stream processing. Since DataStream API is more general, it

should fully incorporate the capabilities of the DataSet API in the future [49].

5.2 Characteristics of Apache Flink

Flink has been mainly developed for Java and Scala programming languages, which are both based on *Java Virtual Machine* (JVM). More recently (since 1.9) a limited support of Python for the Table API has been available [50].

Flink provides APIs on multiple *levels of abstraction* (LoAs) [51, section "Levels of Abstraction"].

The Core APIs of Flink consist of the DataSet API and the DataStream API. They are the second most descriptive LoA. Each of the APIs consists of a class of the same name and operations (transformations) defined on it.

The lowest level of abstraction is provided by `ProcessFunction` which is actually integrated into DataStream API so users can seamlessly switch between them. It gives the ability to create and customize stateful streaming.

A LoA one higher than the Core APIs is the Table API that utilizes the relational model used in database systems. It is related to the `TableEnvironment`.

The highest LoA is called SQL, which specifically provides the ability to write SQL query expressions. It is closely related to the Table API. It mainly provides a `sqlQuery` method called on a `TableEnvironment` which accepts SQL queries as a String parameter. The SQL API is considered incomplete (as of version 1.8) [52].

We'll focus on the Core APIs in this chapter from now on.

5.2.1 Concepts

A *function* is the most general concept, often meaning any executable sequence of instructions. Although from a mathematical point of view, a function should be deterministic, i.e. give the same output when receiving the same input arguments. This might not be true for OOP (Object-Oriented Programming) languages, hence why Java's "functions" are called *methods* and they are a part of some class. The word *function* has a special meaning in Flink, where it is the argument of an operation (operator function) [53]. Implementation-wise, we actually define a class to represent the function, and then pass an instance of it (object) as an argument to the called operation. The class can also be anonymous or a lambda expression [54]. The requirement for these classes is only that they implement the interface that corresponds to the given operation (e.g. `MapFunction` or `RichMapFunction` for map transformation). Inside the function class we have the actual methods that need to be implemented and specify the behavior of the concrete function. Even though the documentation uses this term in various contexts, we'll try to apply it more strictly only on cases that fit the mentioned Flink definition.

An *operation* in Flink is a method that's manipulating the data stream. The types of operations are: transformation, iteration, source and sink. Operations in Flink are lazy (utilize lazy evaluation), which means they are only executed when their result is needed. More specifically, an `execute` method of the

execution environment needs to be called. An exception to this are a few sink operations that cause eager execution when called on a DataSet. Namely `print`, `collect` and `count` [55, slide 35]. [51, 56] (Note that `printToErr` (for printing to standard error stream instead) and `printOnTaskManager`, an operation that can be used to add a String prefix to each printed element - a more general version of the `print` operation, are also evaluated eagerly [57].) An argument to the operation is either a function object or just some Java objects (like Integers) in simpler cases [53].

An *operator* is a representation of an operation. A way to look at it is: when building the Flink dataflow, we specify the operators (nodes) and the data streams (edges) that will later move (flow) between them in the specified direction. When the dataflow is executed, each operator invokes an operation on the data that is being transferred, creating a data streaming environment.

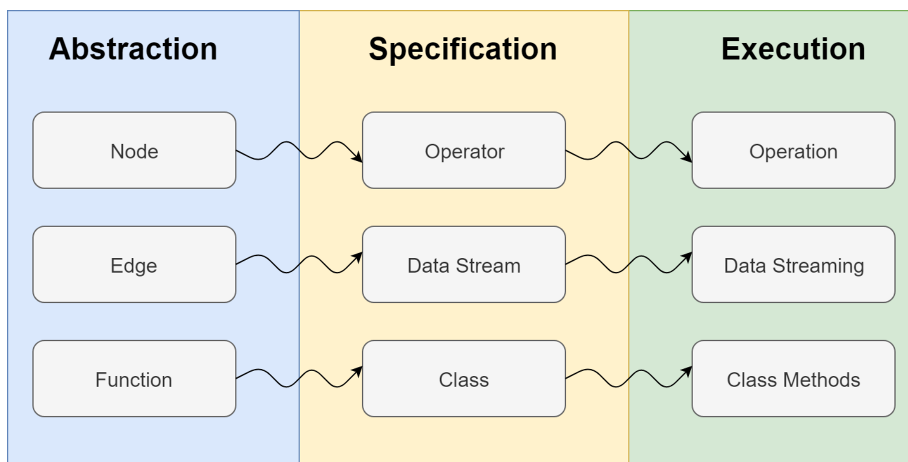


Figure 5.1: Flink Dataflow

5.2.2 Flink Program Anatomy

The basis of a Flink program is an execution environment (`ExecutionEnvironment` for DataSet, `StreamExecutionEnvironment` in case of using a DataStream, and `TableEnvironment` for Table API [58]). Calling a `getExecutionEnvironment()` method on the appropriate class will provide a local environment when the program is run as a regular Java program (from an IDE) or a remote environment (for cluster execution) when invoked from a command line as a JAR file [53]. The execution environment is used to set up the properties of the program (job) and execute it when it's ready [59, 60].

The execution environment provides a data source from which the DataStream/DataSet collection is created. This can be e.g. a file to read the elements from, a whole directory of files (we can also configure the environment to read files from nested directories [61]), a Java Collection whose elements get extracted or a connector to external systems like Apache Kafka. We then apply a series of transformations on the collections and output the results using a data sink (the destination to output the data - can be a

file, console or some other framework that processes them, like using the Kafka connector) [62]. These courses of action form a Flink dataflow and we can view them as building a directed graph topology. Finally, we start the execution of these dataflow processes by calling an `execute()` method on the environment. We can see that the operations are first set up and then executed lazily.

In case of a DataStream, the data are read from the specified sources periodically, so these sources need to be periodically checked. Analogically, they are then being periodically output through the specified sink [63]. On the other hand, sources/sinks of DataSet can be read/written to only once and never checked again.

5.2.3 Common Characteristics of the Core APIs

The two APIs (DataStream and DataSet) have a lot in common and many operations of the same name and functionality can be performed on both (discussed in the next Subsection 5.2.4).

The provided classes (`DataStream` and `DataSet`)[1] are immutable collections of data of the same type. This means that their elements can't be added or removed after the collection's instantiation [53]. Yet the elements themselves might be mutable and therefore modified in the process. The immutability allows for the "old" collection to be forked and sent to multiple operators, creating multiple different new collections [64]. They can also contain duplicates. Unlike typical Java Collections, their contents can't simply be "inspected". (In Java Collections like `List` and `Map`, we have an iterator available.)

Supported Data Types

Flink distinguishes between six categories of data types (of elements of the `DataStream` or `DataSet` collection). It handles the elements according to the type category they belong to.

The categories are:

1. Tuples (and Scala Case Classes) - A composite type storing a fixed number of fields of possibly different types.

2. POJOs - Plain Old Java Objects. To be in this category, a class must: be public, have a default constructor (public and without arguments) (it can then have other constructors), have all of its fields either public or accessible through appropriate getter and setter methods, and finally, it must be supported by the registered Flink serializer.

3. Primitive types - Or more specifically the wrapper classes that provide their object representation. (E.g. instead of int we use Integer, instead of double we use Double.)

[1] Note: We won't typeset these classes in further mentions for brevity.

4. General Classes - If a class is not identified as a POJO, it falls in this category. Can be e.g. any user-defined Java class. Flink treats these data types as "black boxes" and therefore might work with them less efficiently than in case of POJOs. The types in this category are serialized by Kryo - an external serialization framework.

5. Values - Define a custom serialization and deserialization by implementing the `org.apache.flinktypes.Value` interface and the appropriate `read` and `write` methods. It's useful in situations when a general serialization strategy would be deemed inefficient.

6. Hadoop Writables - Types that implement the `org.apache.hadoop.Writable` interface.

Finally, special types like the Scala's `Either`, `Option`, and `Try` are supported.

Copying Behavior

By default, the DataStream collection is copied upon every transformation. The DataSet collection is copied (new element objects are created) at the start of the job execution. An ability to lessen the amount of copying is provided by the function `enableObjectReuse()` of `ExecutionConfig` class. In particular, it brings the DataStream copying behavior on the level of the default DataSet behavior and in case of being called in bounded `ExecutionEnvironment`, it makes the DataSet operate in a "full reuse" mode where reused objects coming from an external system are used throughout the whole job [65]. This comes with some additional precautions that must be taken in order to avoid unexpected and undesired behavior (e.g. not modifying the input objects inside certain transformations) [61].

5.2.4 Operations of the Core APIs

Sources

Source can refer both to the (typically external) place that contains the data and to the "source function", an operation that reads the data from the specified place and creates a data stream from them [61].

A most general source operation for DataStream is `addSource` that accepts a custom function class implementing the `SourceFunction<T>` interface.

Transformations

A *transformation* is a type of operation that is performed on a data stream. The provided stream's data is modified (transformed) as a result of such operation and a new stream is returned (intermediate operation). Examples include filtering, mapping, joining, grouping, aggregating, updating state or defining windows [66, 67].

A basic transformation is `map`. It takes the collection's elements one by one and transforms each one into a new element according to the supplied function. The type of the resulting elements can be different.

A `flatMap` transformation offers a more general approach. Instead of outputting exactly one element for each input, it can output any number of elements each time through a `Collector` object.

A `filter` transformation is used to, as the name suggests, filter out elements and therefore shrink the collection (or in the "best" case, keeping it as is) while retaining the elements' type. A `FilterFunction` class that is supplied as an argument returns a boolean for each element indicating if it should stay (true) or be filtered out (false). For simpler filter functions, use of the lambda expressions is convenient. For an example, let' say we have a collection of integers and want to only keep the ones between 10 and 100 (exclusive). We achieve that by simply calling

```
.filter(x -> 10 < x && x < 100)
```

on the said collection.

A `project` transformation is only applicable on collections of Tuples. It outputs a new collection of Tuples each containing only the specified fields of the original Tuple and in the order they were specified in for the transformation.

Iterations

We can define a general iteration on a DataStream. An analogy for DataSet is the `BulkIteration`. Additionally for DataSet, a more complex `DeltaIteration` is available where efficiently only the necessary part of the solution is modified [68]. The `DeltaIteration` keeps a second DataSet (called the *WorkSet*) whose elements are used in the loop, while the DataSet of elements that are being transformed in each iteration and output at some point is called the *SolutionSet* [57].

An iteration is initiated with calling `iterate()` (`iterateDelta()`) on the collection whose elements you want to send through the iterations. Those functions accept as arguments the conditions on which the loop execution terminates (the conditions can be e.g. a max number of iterations or an empty WorkSet). This creates an initial iterative collection, which we should keep a reference of.

We then define all the transformations that are part of the loop, collectively called the *step function*. (In a parallel setting, one execution of the step function on all partitions is called the *superstep*. All partitions need to complete the superstep before the next superstep is started.)

When we're done, we supply this transformed collection as an argument into the `closeWith()` transformation called on the initial iterative collection (in case of a `DeltaIteration`, both the transformed WorkSet and a SolutionSet of which just the affected part where the elements have been changed ("delta") is needed). This final transformation gives us the iteration output (the transformed SolutionSet) in the form of a DataStream or DataSet, on which we can continue to define further transformations or start new loops.

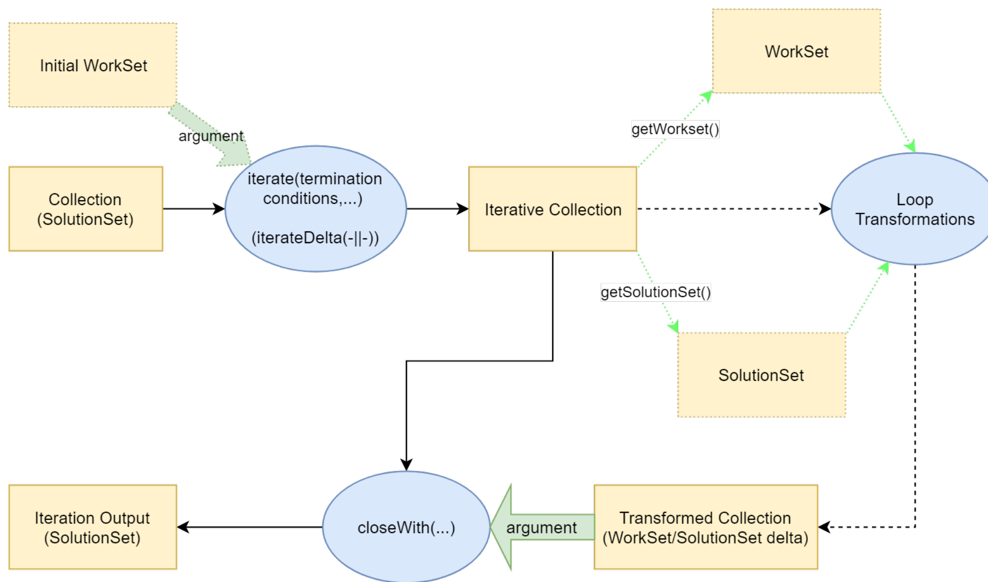


Figure 5.2: Flink Iterations Diagram. Visualizes how the general iteration works, pointing out the specifics of `DeltaIterations`.

DataSet also supports iterations, namely bulk and delta iterations. The bulk iteration can be initiated by calling the `iterate` method, where we can specify the maximum number of iterations. This creates an `IterativeDataSet` which we should save into a variable. After that, we apply all transformations that should be in the loop and pass the result of them into the `closeWith` method called on the saved `IterativeDataSet` that was created right after calling the `iterate` method. (There's an optional second parameter to `closeWith` - a DataSet which, if empty, causes the termination of the loop.) The second type of iterations - delta iterations - keeps a state in the form of another DataSet. Each subsequent iteration then accepts the new transformed DataSet (called the workset) produced from the previous one along with the update (delta) of the state (called the solution set). This type of iteration terminates when the workset is empty or when the maximum amount of iterations has been reached.

5.3 Differences of DataStream and DataSet API

5.3.1 DataStream API

A DataStream API is an interface for processing (possibly) unbounded data streams [63, 51]. It consists of the DataStream class and transformations (methods) that can be called on it. A DataStream is initialized from and run inside of a `StreamExecutionEnvironment` [59].

A source operator `socketTextStream` to read from a network socket is available only for DataStream [63].

5.3.2 DataSet API

A DataSet API is used to process bounded sets of data. Although a DataSet is internally handled as a data stream so we can talk about streams in all cases [51, section "Batch on Streaming"]. As mentioned previously, the DataSet API consists of the DataSet class and transformations defined on it. We can also view `ExecutionEnvironment` as being part of it [60]. The DataSet class is a bounded (finite) collection of data of the same type (but it doesn't implement Java `Collection` or `Iterable` interface). This API can be useful for processing historical data.

DataSet has additional tranformations not found in DataStream. One of them is a `mapPartition` function that transforms a whole parallel partition of the collection. The partition data with an Iterator are given as an input and a transformed partition with an arbitrary number of data is output [57]. A `reduce` function, gradually combining all elements (one-by-one) into one result, can be applied on the whole DataSet as well as on a grouped DataSet where it provides one result for each group [61]. (Unlike for DataStream, where reduce can be applied only on Windows, as the result of reducing the whole stream would never come.) Similarly a `reduceGroup` function works on a whole group of elements at once and returns any number of elements as a result (a transformed group). The "group" might be the whole DataSet. Another special function is `distinct`, returning only distinct elements by removing all duplicates of the DataSet (with respect to all or just selected fields of each element). It can be thought of as effectively creating what we know as a mathematical set. This and other following transformations closely resemble SQL operations and have a similar functionality. `xOuterJoin` transformations (left, right or full) combine two DataSets and produce the same output elements as an (inner) join operation. But an outer join preserves the elements that had no matching element in the other stream. It does so for the elements of the first/second/both stream(s) respectively. (In such case, the "missing" input element is replaced by a null value.) A `cross` operation creates a cross join (cross product) of two DataSets. In other words, it creates all possible pairs of elements. A user-defined function can be supplied to the follow-up `with` operations to define a custom logic of combining each pair of elements into one. Since this transformation can be very computationally expensive, one can use a `crossWithHuge` or `crossWithTiny` function to tell the optimizer that the second DataSet is much larger/smaller than the first one respectively. A `union` transformation simply combines two input DataSets into one containing all of their elements. Note that if the input DataSets contained duplicates, the output will contain them as well, thus it is a bit different from a mathematical set union. Some partition-level transformations are also present, like `rebalance`, which rebalances the load of each parallel partition. There are functions to hash-partition, range-partition or apply a custom partitioning of the whole DataSet on a given key. A `sortPartition` transformation sorts the partition based on a specified filed and in a specified order. (Similarly, a `sortGroup` transformation can be applied on a grouped DataSet.) It then makes sense to be able to use a `first` function that returns

the first n elements (n being specified as an argument).

So in summary, the DataSet API with the corresponding class can be applied to finite collections of data. It provides more functionality thanks to that property, but it's less general and can't deal with never-ending streams (while DataStream API can handle both).

Most functions that can be applied on the full or grouped DataSet, can be applied only on a windowed DataStream.

5.3.3 Specific Differences

DataStream and DataSet APIs have some interesting differences - mostly details that can be observed in code [69].

- To print elements of a DataStream in console with each having a chosen prefix, we can call `print()` with `String` argument. For DataSet, an operation of the same name is deprecated and we can instead call `printOnTaskManager()`. The interesting part is, that `printOnTaskManager()` is one of the few operations that trigger eager execution (see 5.2.1).
- Only the DataStream API gives an access to the low-level `ProcessFunction`, which exposes timestamps, etc.. It also supports windowed operations, which are essential for DataStreams but unneeded for DataSets.
- For DataSets of `Tuple`, transformations like `keyBy` give a convenient ability to specify keys by only typing the numbers of selected `Tuple` fields. In DataStreams we need to have a `keySelector` function as usual.
- A `join` transformation is called differently. For DataSet, we later call `with` to specify the function to be performed on the joined sets. If the return type can't be inferred, we have to subsequently call `returns` and specify it. For DataStream, the operation to invoke the function on joined streams is called `apply`, and it also accepts the types of output elements directly.

5.4 Features of Apache Flink

Decompression of Input Files

Flink supports decompression of certain types of extensions (e.g. .gzip, .deflate) for any type of readable input file. But the process isn't parallelizable, which might impact the overall job scalability [61].

Configuration Functions

By the term "calling function on a transformation", we'll mean the practice of calling a "special type" of function (method) directly after the given trans-

formation function. This “special type” will configure the transformation’s behavior in some way.

Passing Parameters

One case of the function call on a transformation is supplying parameters to the transformation, which can be done using the `withParameters()` method. The parameters are supplied in a `Configuration` object. An alternative is to pass the parameters into the transformation’s constructor. The first approach requires the implementation of a `RichFunction` interface, as the Configuration object is passed into the transformation’s implementation of the `open()` method. Another alternative is to pass the Configuration object into the job’s execution environment and therefore having it accessible globally from any transformation implementing the `RichFunction` interface. Passing in a custom class that extends the `ExecutionConfig.GlobalJobParameters` class is also possible.

5.4.1 Notions of Time

Flink has three notions of time associated with individual collection elements [70].

Processing time is the simplest notion of time, analogical to standard programs. It refers to the local time of the machine at an operator performing a time-based operation. Processing time mode might be useful for applications where low latency is prioritized over precision of results.

Event time is the time of creation of each event. This time is typically already associated with events when they arrive into Flink dataflow. It is represented by *timestamps* (`long` values), which are assigned to each event based on a custom strategy. Using event time allows for accurate and consistent results, whether recorded or real-time processing takes place.

Ingestion time is the time when an event enters the Flink dataflow, i.e. when it’s being processed by the source operator.

Event Time Watermarks

Event time needs special constructs to work. Apart from a strategy to assign timestamps, special elements of the stream called *watermarks* have to be created. This can all be done in a `assignTimestampsAndWatermarks` transformation or directly in a source operation.

Watermark can be thought of as a special element of the stream that contains a timestamp and tells the program that event time has progressed to that given timestamp. The purpose of watermark system is to define when to stop waiting for the arrival of earlier events. When a stream is being processed and a watermark t is encountered, the program doesn’t expect any further events from that stream with a timestamp $t' <= t$, and can for example close a past window. Common, simple types of automatic watermark generation include periodically generated watermarks with ascending timestamps, used in

cases where events in a given (source) task simply occur in an ascending order and their timestamps can therefore be used as watermarks, and "bounded-out-of-orderness" watermarking, where the watermarks are generated based on a fixed maximum amount of lateness of events [71].

When working with windows, an event is *late* when, at the time of its arrival, its event time is less than the time set by the most current watermark. Applications can allow (deal with) lateness, though unbounded lateness would mean unbounded (ever growing) state and is thus undesirable [72].

5.4.2 Windows

Window is a finite block of stream elements, over which we can make computations [73]. Windows have a defined scope: they can be time driven (aggregate data over 30 seconds) or data driven (aggregate 100 elements). Windows can also be distinguished based on mutual overlapping: tumbling, sliding and session windows.

5.4.3 Keyed Collections

The collection can be structuralized by grouping together the elements sharing some common characteristic. This is realized by defining keys on them (using a `keyBy()` on DataStream or a `groupBy()` on DataSet). A key is a function performed on each element of the collection and giving a value that determines which group of elements it belongs to. So all elements that have the same value assigned by the key function will be in one group. The subsequent operations are then applied on the keyed collection.

5.4.4 State

One of the major features is keeping a persistent state between processing individual elements in the given transformation. Flink provides this possibility and more by implementing a `RichFunction` interface. State can be used for both keyed and non-keyed collections. There are more types of state supported for the keyed collections (e.g. `ValueState`, `ListState`, `MapState`). We can use `CheckpointedState` for the non-keyed collections.

Passing initial parameters to the transformations can be realized in a standard way by using constructor or by other means. The configuration parameter in the `open()` function is obsolete and empty for the DataStream class.

5.4.5 Broadcasting

A broadcasting of one collection's elements to the function that processes another collection is available. For DataStream, this is done by calling `broadcast()`, which creates a broadcasted connected stream (two connected streams where the elements of one will be available when processing any element of the second one inside a specified transformation). For DataSet,

it means broadcasting the whole collection by calling `withBroadcastSet()` specifying the set and the name that it should be identified with inside the function. The function can then access it through its runtime context. In DataStream, it's called the broadcast state and in DataSet, the broadcast variable we access it with `getBroadcastState()/getBroadcastVariable()` respectively, specifying the state descriptor / broadcast variable name respectively [74].

5.4.6 Distributed Cache

We can register files (local and remote) in the execution environment to make them available in each node (parallel instance) of the distributed cache. They can then be accessed from the runtime context of any transformation that implements the `RichFunction` interface.

5.4.7 Debugging

When debugging, it is easier to run the program locally instead of it being distributed on a cluster. Furthermore, it is appropriate to test it on a smaller set of data. For this purpose, a `LocalEnvironment/LocalStreamEnvironment` (subclass of `ExecutionEnvironment/StreamExecutionEnvironment`) can be created and used for `DataSet/DataStream` testing. This should be used together with obtaining the "test data" from any Java Collection initialized directly in code, outputting the results to another Collection and setting breakpoints in a local-run IDE.

5.4.8 Semantic Function Annotations

One can use Java annotations to give Flink a hint about the behavior of a function. More specifically, to explicitly state what does the function do with its input/output fields. They can be specified before a function class. This feature might help optimizing the program. Based on the provided information, the optimizer can conclude that a sorting or partitioning of some data is preserved. This might mean that a data shuffle or sort is unnecessary.

One type is `@ForwardedFields`, which accepts field expressions to specify fields which remain unmodified by the function. Furthermore, if a field was unmodified but e.g. received as the 2nd field of the input Tuple and sent as the 4th field of the output Tuple, we'd write

```
"f1->f3"
```

as an argument. A wildcard `*` can be used to refer to all fields of the given Tuple. An alternative is to pass the arguments into a `withForwardedFields()` function called right after the respective transformation.

A `@NonForwardedFields` annotation is used to only specify all the non-forwarded input fields. All of the unspecified fields are then considered as being left unchanged and output in the same position as they've been input

in. The class function that this annotation is used on must have an identical input and output type.

A `@ReadFields` annotation, if used, has to declare all fields that are accessed and used by the respective function in some way. This does not include the forwarded fields. The optimizer then knows that all of the other (i.e. unspecified) input fields were either forwarded or not used at all.

5.4.9 Parallel Computing in the Context of Apache Flink

Parallel computing in general means simultaneous use of multiple resources to complete a computational task. Such task first has to be broken into discrete parts that can be processed concurrently [75, section "What is Parallel Computing?"]. This approach provides concurrency, allows for solving more complex and larger (i.e. requiring more memory) problems and can also significantly save time.

Flink supports parallel execution and has a specific conception of parallelism. A parallelism of an operator is the number of operator subtasks that are being executed in parallel. Together, these parallel instances form what is called a task. Each parallel instance processes a part of the task's input data [76]. The parallelism of a stream is determined by (and equal to) the parallelism of its producing operator [51]. The degree of parallelism can be specified for each operator individually, for the whole execution environment (affecting all operators it executes), through the client that represents a program, or at the system level, providing default parallelism for all execution environments it creates. The priority of the aforementioned settings goes from the highest to the lowest. E.g., setting a parallelism of an individual operator overrides everything else.

5.4.10 Using DataStream for Matrix Representation

When one wants to represent matrices using DataStream class, they'll stumble upon one fundamental problem. Since DataStream was made to process elements as they arrive, it wouldn't make sense to wait for all matrix elements as processing it whole can also be done by using DataSet API.

Demonstrating the Problems on Matrix Multiplication

Let's focus on matrix multiplication as a concrete operation we'd like to implement/use. For matrices $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$ we want to compute

$$AB = C. \tag{5.1}$$

By definition, we can compute each element of $C \in \mathbb{R}^{m \times p}$ using the equation [77]

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{5.2}$$

We can see that for any $c_{ij} \in C$ to be fully computed, we need to receive the whole corresponding row of A and the whole corresponding column of B. But we don't know when the sum is complete unless we are provided additional information. A sufficient information could be knowing the dimensions of the matrices we're working with beforehand (for dense matrices), or getting a signal when the row/column is 'complete' - meaning we received all of its elements (for sparse and possibly dense matrices; In Flink, this could be realized by using event time timestamps and watermarks or by some convention (wait for a given amount of time after the last element was received) or custom special elements that would tell the index of completed row/column). We can then compute the elements of C which have all the required row/column elements known, even before receiving the rest of the input matrices.

We could view each column of matrix C in (5.1) as a linear combination of column vectors of A where the individual scalars are represented by the corresponding column vector of B. If multiple chained multiplications are present, we could multiply the "scalars" first and do the linear combinations last.

5.4.11 Efficient Matrix Representation

We want to be able to perform basic matrix operations and some more specific ones, like computing the *spectral radius*. It is also desirable to be as efficient as possible (in speed, memory, etc.). There might be other considerable factors like scalability. There exist some solid established Java matrix libraries like *Jama* and *EJML*. A less known library called *ojAlgo* (oj! Algorithms) seems to be performing very well in Java matrix libraries benchmark that tests various properties using different matrix sizes and operations (even coming off as the best one) [78]. Note that this benchmark tool wasn't developed by ojAlgo creators (it was actually developed by the creator of EJML). The library is written purely in Java, supports multithreading, and the authors demonstrate its usage on creating artificial neural networks and solving mathematical programming problems [79]. Therefore it seems like a good choice for us. The disadvantage is in having almost no *JavaDoc*, which leads to the need for reading some external instructions or studying the source code in order to understand the exact functionality.

5.5 Implementation Description

The core of the implementation is the RC library itself. Of course, various testing classes, especially application examples are a large and important part of the project too.

5.5.1 Project Structure

The project folder is `flink-rc` and the library rests under `src/main/java`, which is a standard Maven structure. There are two parts: `lm`, representing the linear model (readout) implementation, and `rc_core`, representing the reservoir implementation. All the examples and other experimental, testing and test-supporting code can be found under `src/test/java`.

By default, the examples create plots when running. In order to properly run the plotting scripts (without rewriting paths in the code), the `python_plots` folder has to be in the same parent folder as the project. The plots are then generated in the `plots` subfolder.

5.5.2 Data Representation

Since our library is build using Flink's DataStream/DataSet API, it is designed to accept `DataSet` or `DataStream` (collectively referred to as *Collection* here for brevity) of `Tuple2<Long, List<Double>>`. as labeled inputs and Collection of `Tuple2<Long, Double>` as labeled outputs.

The `Long` value represents an index of the record. Input records and output records with the same index get paired together during the model training and testing/predicting. Note that their timestamps might be different, if used. `List<Double>` represents the vector of input features $\boldsymbol{x}(t)$, where t is basically the associated index. Each input feature $x_i(t)$

We could think about supporting different (numerical) data types. Java has a `Number` superclass that encompasses all the different classes like `Integer`, `Float`, `Double` or `BigDecimal`. But standard arithmetic operations are not defined on it, so it is practically unusable. An alternative would be to overload our function with different types of input parameters, but it might not be worth the hassle. For now, we decide to stick with `Double` class (and the corresponding double primitive type). Note that `Double` is just a wrapper class to represent the double as an object. This allows us to represent real numbers with a finite precision of at least 15 significant decimal digits [80].

5.5.3 Linear Model Functions (Readout)

We've created a `LinearRegression` class for online (SGD) training and testing (both DataStream/DataSet API) and a `LinearRegressionPrimitive` class that realizes the LR with a non-Flink approach, and can be mainly used for the offline (pseudoinverse) training.

A `fit` function for the input stream serves to train the linear model using Gradient descent and make it better with time. It returns a new optimal $\boldsymbol{\alpha}$ parameter with each incoming input-output pair. It accepts the `outputStream` parameter to make input-output pairs based on a common timestamp. It also accepts specifications for the training. An `alphaInit` parameter tells the function what should be the initial value of the $\boldsymbol{\alpha}$ vector. It's the value from which the Gradient descent attempts to converge towards a minimum.

Another parameter, `numIterations`, denotes the number of Gradient descent iterations for each input-output pair. A `learningRate` parameter influences the step size of GD. This means how far in the direction of negative gradient we want to go in each iteration. A smaller size is safer in the sense that we are less likely to "overshoot" and go past the minimum. A bigger size might mean faster convergence but a bigger chance of "overshooting".

A `predict` function predicts an output from the input stream using the fitted model. It is designed to build on the computations from the `fit` function. It's first parameter, `alphaStream`, is a `DataStream` of optimal $\boldsymbol{\alpha}$ vectors. When a list of alphas with a more recent timestamp arrives, it is considered more optimal and the current $\boldsymbol{\alpha}$ list is replaced by it. This allows for the "online" learning to take place. The second parameter is `alphaInit` and provides the list of alphas that are used in the initial model until the first element from the `alphaStream` arrives. The `alphaStream` can be viewed as a stream of instructions that modify the function. It is expected to be much "slower" than the input stream the function is called on (several input vectors will be processed before an updated $\boldsymbol{\alpha}$ vector arrives).

5.5.4 Reservoir

At first, we were trying to develop the Echo State Network reservoir using multiple approaches. But the only truly developed and up-to-date one is the `ESNReservoirSparse` using ojAlgo and `SparseStore` classes to represent the W_{in}, W matrices. This class extends the `RichMapFunction` and can therefore be called throu a `map` transformation.

5.5.5 Higher-Level Examples

The originally developed testing examples (for LM) were in separate classes and can be found in `lm` package under `test/java/`. A more elaborate example structure was developed later, called "higher-level" examples, to support much more convenient development and modification of individual examples. The most up-to-date core class that was used for all experiments is the `HigherLevelExampleBatch`. Its configurable fields and more is in the `HigherLevelExampleAbstract` abstract class.

All of the individual examples that use the complete RC model are in the `rc` package. All examples have the `*Example` suffix and serve mostly to store and modify the program configuration. `IdentityTest` is just for some basic validation. `HyperparameterAnalysis` class was developed to be able to conveniently analyze chosen hyperparameters and produce plots shown in the later Section 6.3.

The easiest way to learn how to properly create a custom RC model is by looking at the `HigherLevelExampleBatch` class and ignoring e.g. the plotting functionality. An outline of how to use DataStreams instead of DataSets is given in `HigherLevelExampleStreaming`.

Besides the RC parameters configuration specified in the following subsection, there are a couple of `boolean` variables to "switch" between behaviors of the example programs, they are:

- includeMSE – if we want to include and print MSE estimates during the LM training.
- plottingMode – if we want to plot the results
- debugging – if we want to print multiple values to console during execution
- lrOnly – if we want run only the LR (readout phase)

Null Values

Flink might have problems with `null` values inside a *Collection*. In particular it explicitly `doesn't` support `null` values when working with a *Collection* of `Tuples` and the built-in `TupleSerializer` will throw an `Exception` [81, section "Flink's TypeInformation class"]. Strangely, when reading *Collection* of `List<Double>` elements, it depends. If we use `DataSet`, everything seems to run smoothly. When using `DataStream`, we get an `Exception` from the `ListSerializer`.

The easy general solution is to never let any transformation (or source) output a null value and reserve a different value if we really need to signify an invalid record. E.g. if a `map` function originally outputs null values, turn it into a `flatMap` function and output nothing in that case.

5.5.6 Default Configuration

Here we provide the list of default values of all configurable parameters that are relevant to RC. All of them can be found specified in the `HigherLevelExampleAbstract` class.

- Reservoir
  - size of W (N_{x}): 10 (always required to specify when calling our ESN implementation)
  - topology of W: "Cyclic with Jumps"
  - range of W_{in}, W weights: 1
  - shift of W_{in}, W weights: 0
  - scaling parameter of W (α): 0.8
  - size of jumps: 2 (for deterministic topologies with jumps)
  - sparsity of W: 0.8 (80%) (for "Sparse" topology)
  - activation function (transformation): `Math::tanh` (hyperbolic tangent)
  - initial state vector ($\boldsymbol{x}(0)$): $\mathbf{0}$

- Readout
  - initial vector of regression coefficients ($\boldsymbol{\alpha}_0$): $\mathbf{0}$
  - learning rate (η): 0.01
  - step-based decay of η: active (true)
  - regularization factor (μ): 10^{-10}
  - bias constant (1): included
  - input vector ($\boldsymbol{u}$): included
  - decay granularity of η: 32
  - decay amount of η: 1/16
- Other
  - n-time-steps ahead prediction: 0 (disabled)
  - applying only linear regression (readout) part: disabled (false)

The "range" and "shift" parameters together result in having the weights from the interval $[-0.5, 0.5]$.

The scaling hyperparameter α that is equal to the spectral radius (ρ) of W was by default set to 0.8 instead of a more neutral 0.5, since values close to 1 are generally considered more optimal [82, p. 29].

To include the "bias" and "input", we perform concatenation of values at the end of reservoir phase, resulting in $(1, \boldsymbol{u}, \boldsymbol{x})$ as the new readout input ($\boldsymbol{x}$).

The step-based decay of learning rate was used as it generally shows a slight improvement over the constant version. The learning rate was updated by applying the previously introduced equation (3.17).

Chapter 6

Experimental Results

During implementation of some task, it seems reasonable to first focus on functionality (getting the expected result). And then on performance (highest speed, lowest memory possible). We were comparing the speed of some elementary implementations (that perform the same tasks). One, maybe obvious, but commonly mistaken thing that we checked was comparing implementation using Java 8 streams

```
double spectralRadius = eigenpairs.parallelStream()
    .map(x -> x.value.norm())
    .max(Comparator.naturalOrder()).get()
```

with a more native one

```
eigenpairs.sort(Comparator.comparing(x -> x.value));
double spectralRadius = eigenpairs
    .get(eigenpairs.size() - 1).value.norm();
```

here demonstrated on computing the spectral radius.

After a couple of "warmup" runs, the stream approach takes ~2.5ms, while the "native" approach takes only ~1.7ms. We probably don't need a more precise measurement to see the difference.

Even if we were using for loops, the cost of creating a stream is very high and can only get faster when processing a very large amount of data.

To test the linear models produced by linear regression (readout phase) as well as the complete "reservoir computing models", we have developed the following real-world examples: Mass Balance of Glaciers, Yearly CO_2 Emissions by Nation, $PM_{2.5}$ Pollution in Seattle Area. We've also used some datasets generated by mathematical functions for specific purposes.

These datasets were used for both offline (using Moore-Penrose inverse) and online (using Gradient descent) learning.

6.1 Linear Regression

With LR, we'll be modelling and plotting a linear relationship between an example-specific notion of time (x-axis) and the target (model output) value (y-axis). This mainly serves as a validation of our LR implementation as well as a demonstration of its capabilities. Note that the y-scales for training and testing (if training is shown) are probably different.

In our experiments, we've used the following configuration values (unless stated otherwise): The initial learning rate value η was chosen by trial-and-error. The offline version was fitted without regularization ($\mu = 0$).

We'll be splitting all datasets according to the simple, commonly applied 80:20 ratio (training:testing). Since our example datasets are rather small, we need to have enough data points to train the model. But we also want to avoid overfitting and perceiving models as too precise compared to how they would perform in production. We'll then visualize the fitting of testing datasets.

Note that we didn't normalize the data in this part, and therefore the MSEs of the experiments may vary based on the scale of values.

Apart from that, we've used the default configuration specified in Subsection 5.5.6 under "Readout".

6.1.1 Fitting Average Cumulative Mass Balance of Reference Glaciers Worldwide

The data used for this example is sourced from ([83], [84] [1]). Our input variable x has one feature, the *Year* column. And our output variable y is the *Mean cumulative mass balance* column (describing how much the mass has increased or decreased since "Year Zero" (1945 in this case); given in meter water equivalent).

[1]Data has been obtained via: `https://datahub.io/core/glacier-mass-balance` (Accessed: 4. December 2019)

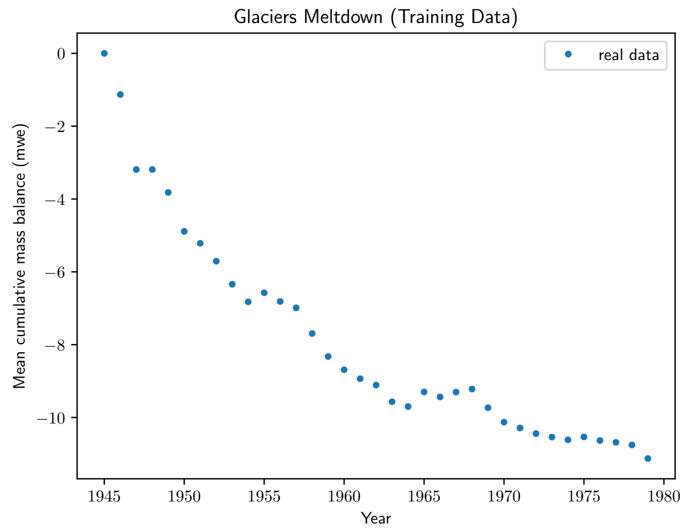


Figure 6.1: The data used for linear model training (50% of total).

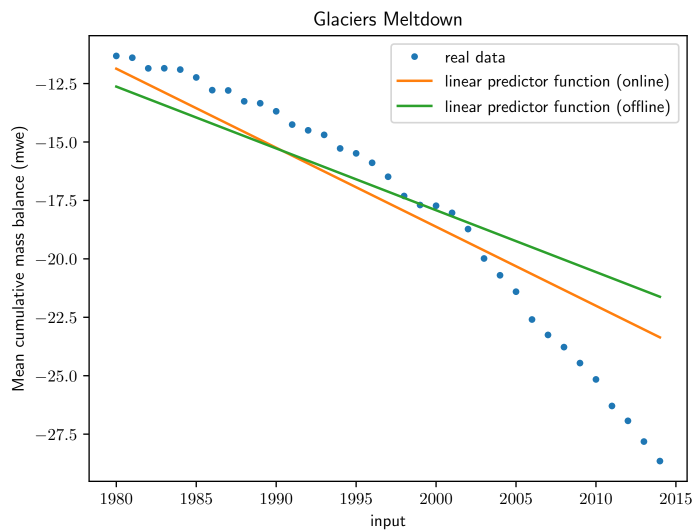


Figure 6.2: Using 50% of data for testing. Online: $\eta = 0.01$, MSE ≈ 4.3109. Offline: MSE ≈ 7.8349.

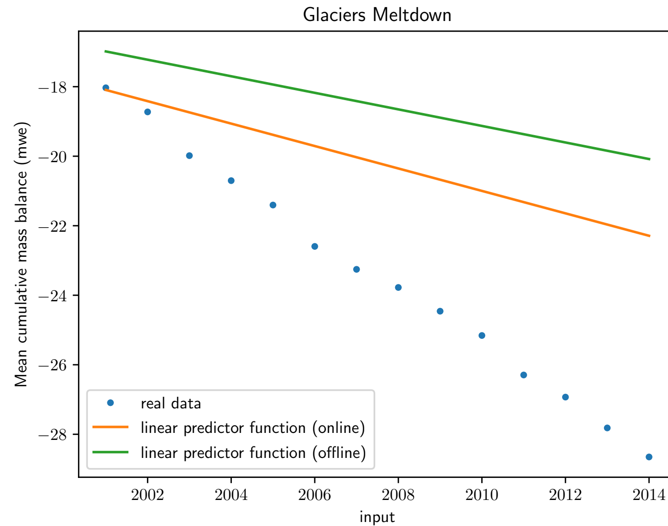


Figure 6.3: Using only 20% of data for testing. Online: $\eta = 0.01$, MSE ≈ 14.1336. Offline: MSE ≈ 28.9949.

We can see that for the 50/50 (training/testing) split in Figure 6.2, the fitted line looks more appropriate than in the case of 80/20 split (Figure 6.3). This might be simply because the glacier mass started decreasing more rapidly in the past few years, and therefore the historically trained model can't produce a good-fitting line.

6.1.2 Yearly CO_2 Emissions from Fossil Fuels by Nation

This dataset [85] [2] contains (estimated) annual CO_2 emissions of countries that existed (and produced emissions) at the time. The data spans from 1751 to 2014. The emissions are measured in thousand metric tons (kt) of CO_2. For the purpose of testing LR, we've chosen a subset of 4 countries (United Kingdom, Norway, Czech Republic and Mainland China). Note that e.g. the first entry for Czech Republic is from the year 1992 (at the end of which it became independent), whilst the first entry for UK is from 1751 (beginning of Industrial Revolution). So the amount of data for each country is different.

We'll be predicting the total number of emissions for the given country and year. Since we need to have only numerical input, we'll substitute the country string using *one-hot encoding* [86]. It creates a series of 1s and 0s where the i-th country will have 1 on the i-th place and the rest will be 0. So for 4 countries, we need 4 new input values in the model. Then e.g. Czech Republic will be replaced with $0, 0, 1, 0$. This is generally useful when we don't want to assume any relationship between the different input values with regards to the output value that we're predicting. The downside might be

[2] Data has been obtained via: `https://datahub.io/core/co2-fossil-by-nation` (Accessed: 22. January 2019)

the number of new variables it creates when transforming a larger amount of distinct values.

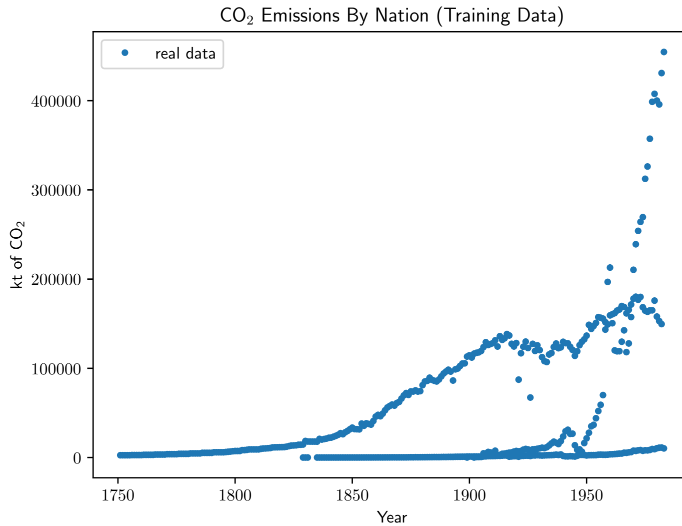


Figure 6.4: Training data for CO_2 Emissions By Nation using United Kingdom, Norway, Czech Republic and Mainland China

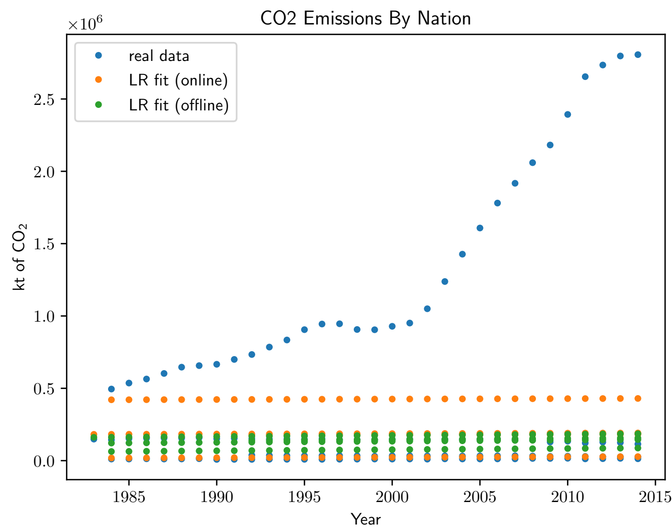


Figure 6.5: Online: $\eta = 100$, MSE $\approx 3.5597 \times 10^{11}$. Offline: $\mu = 10^{-11}$, MSE $\approx 5.1174 \times 10^{11}$.

We can tell from the graph that the data is not very fit for creating a generalized model. First of all, the data isn't linear, which could be fixed by adding more features that make the fit nonlinear. Another problem arises

from the differences between countries. China's emissions have started rising more recently, especially in the years belonging here to the testing dataset. Czech Republic isn't even present in the training dataset, but not because it hasn't produced any emissions (geographically speaking) - here we could've combined it with data for Czechoslovakia, the preceding nation. The large MSE values are also due to a large span of output values (from zero to few million). For some η values here, the online version performs better than the offline (probably by chance).

With this type of data, it seems much better to create a separate model for each country. Below, we present the training and fitted testing datasets for all 4 countries.

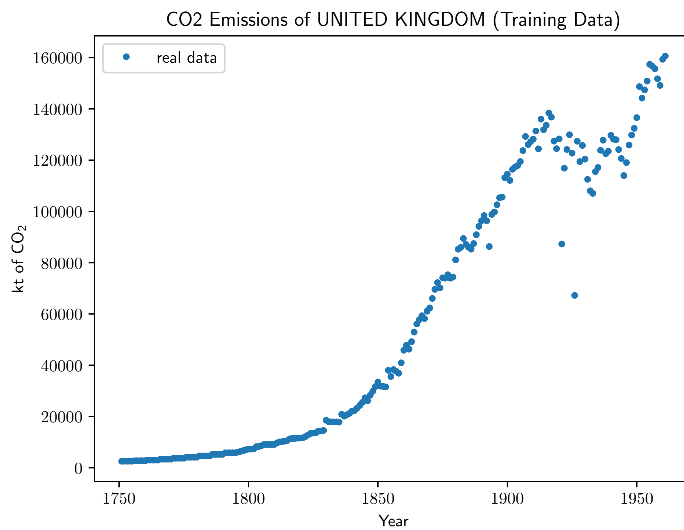


Figure 6.6: CO_2 Emissions of United Kingdom (Training Data)

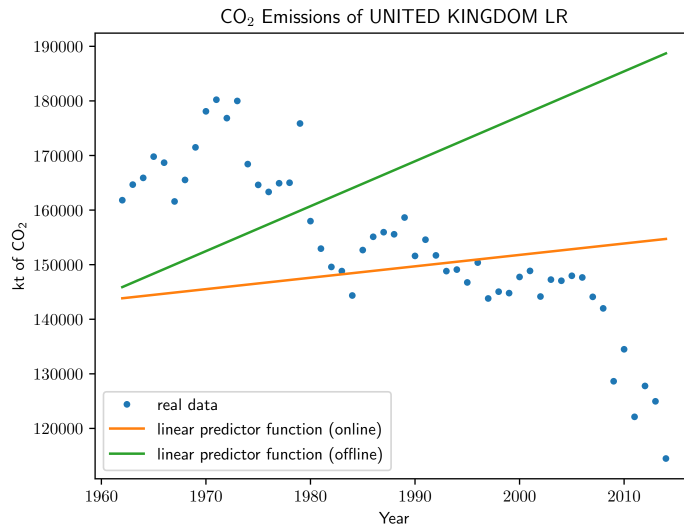


Figure 6.7: Online: $\eta = 10$, MSE $\approx 3.1896 \times 10^8$. Offline: $\mu = 10^{-11}$, MSE $\approx 8.7852 \times 10^8$.

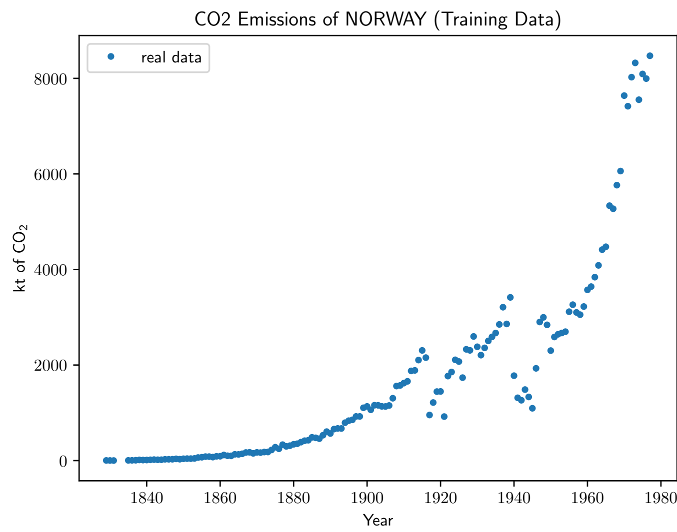


Figure 6.8: CO_2 Emissions of Norway (Training Data)

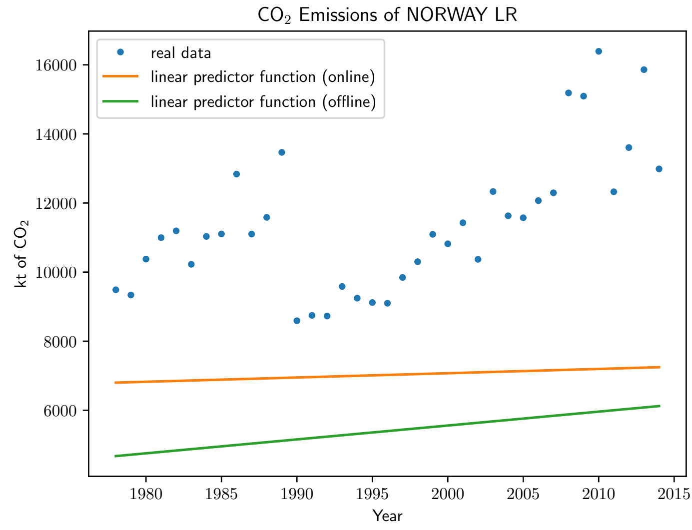


Figure 6.9: Online: $\eta = 150$, MSE ≈ 7866859. Offline: $\mu = 10^{-11}$, MSE $\approx 3.8894 \times 10^7$.

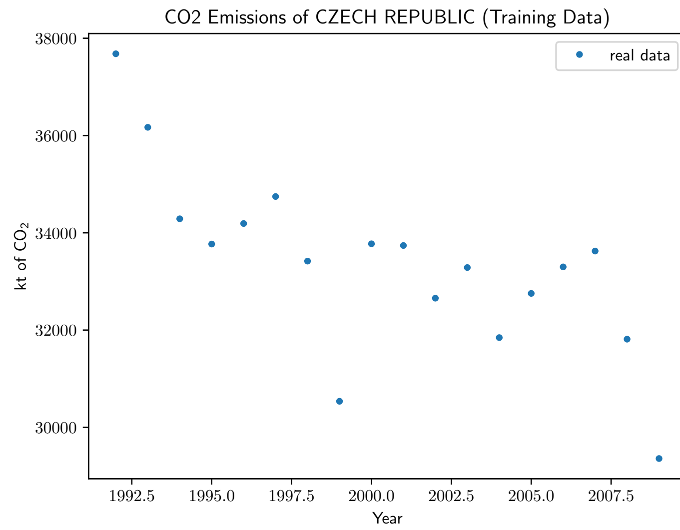


Figure 6.10: CO_2 Emissions of Czech Republic (Training Data)

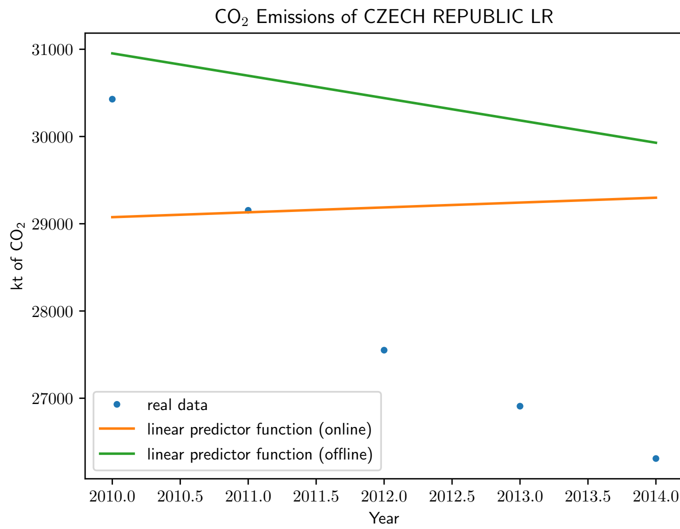


Figure 6.11: Online: $\eta = 1$, MSE ≈ 3860411. Offline: $\mu = 10^{-11}$, MSE ≈ 6966984.

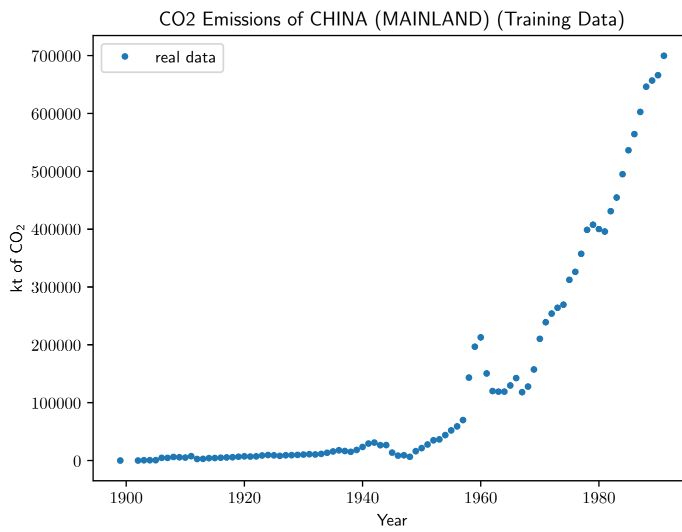


Figure 6.12: CO_2 Emissions of Mainland China (Training Data)

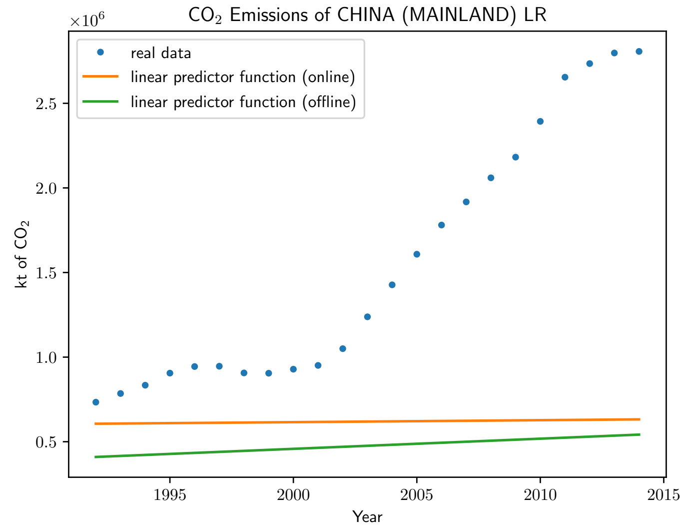


Figure 6.13: Online: $\eta = 100$, MSE $\approx 1.3605 \times 10^{12}$. Offline: $\mu = 10^{-11}$, MSE $\approx 1.6139 \times 10^{12}$.

We can see that most of the nations seem to have had an increase of emissions in the past, while their emissions stagnate or get lower in the more recent years (used in testing). For China, on the other hand, the increase in the past few years is even more rapid. This results in our fits "staying behind". Overall, the used training sets weren't the most suitable due to the tendency of data.

6.1.3 $PM_{2.5}$ Outdoor Air Pollution

The used dataset contains detailed information on the amount of $PM_{2.5}$ air pollutant (particles of solid or liquid matter that are less than $2.5\mu m$ in diameter and are able to pass into lungs and cause health problems [87]) in the region Seattle–Tacoma–Bellevue, WA (core-based statistical area) in 2019 [88]. The daily average concentration of $PM_{2.5}$ is measured in micrograms per cubic meter ($\mu g/m^3$).

We'll want to predict the average of all measurement sites for each given day.

The dataset contains some missing values (NaN), where some measurement sites have almost no valid values. A simple way to deal with them would be to delete the corresponding columns. We could also think about some more complex data imputation. But for the sake of simplicity, since we want to compute the average, we'll consider them taking on the current running average value (that is the average of already processed sites for a given day).

We'll normalize the data by dividing each sample by 50, which almost all the values seem to be lower than.

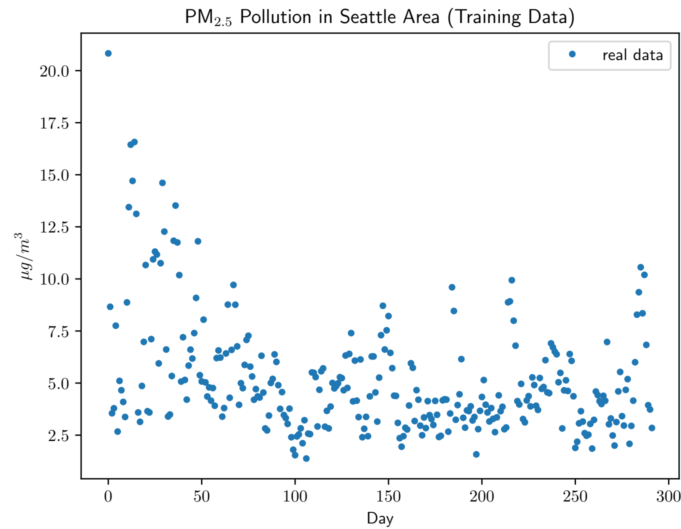


Figure 6.14: $\mathrm{PM}_{2.5}$ Pollution in Seattle Area (Training Data)

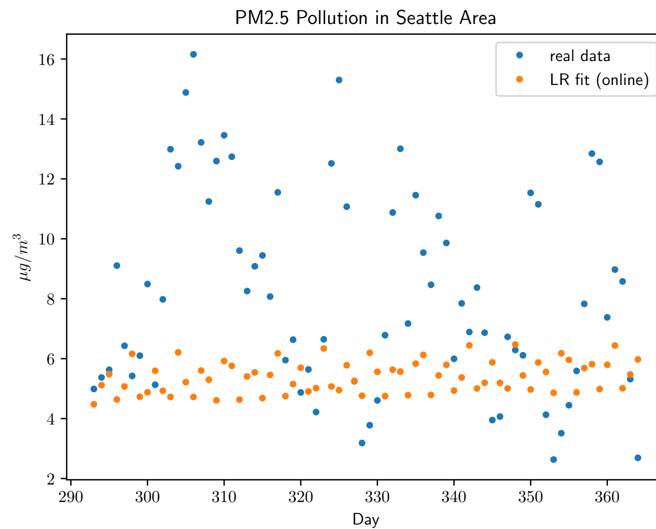


Figure 6.15: Online: $\eta = 0.001$, MSE ≈ 19.5336.

We can see that this data has a nonlinear tendency. Based on the graph, we can guess that there is some periodicity and exponential-like increasing function that decreases once in a while.

Even when adding some nonlinearities to the input vectors, it's more challenging to fit this type of data.

6.1.4 Limitations of Linear Models: Nonlinear Function Approximation

Ensuring the correctness of gradient descent implementation was by far the most challenging part. For that purpose, we additionally tested the online version with a variation of function-generated data.

We created multiple simple functions with increasing complexity, last of which was the function

$$f(x) = 5 + \frac{x * sin(x)}{500} + \left(\frac{x}{500}\right)^2. \tag{6.1}$$

This function introduces a shift of the data as well as nonlinearity/periodicity, so it's not so straightforward to fit using linear regression. Here we've also normalized the input values (divided each by N) to get a better result for the "online" version.

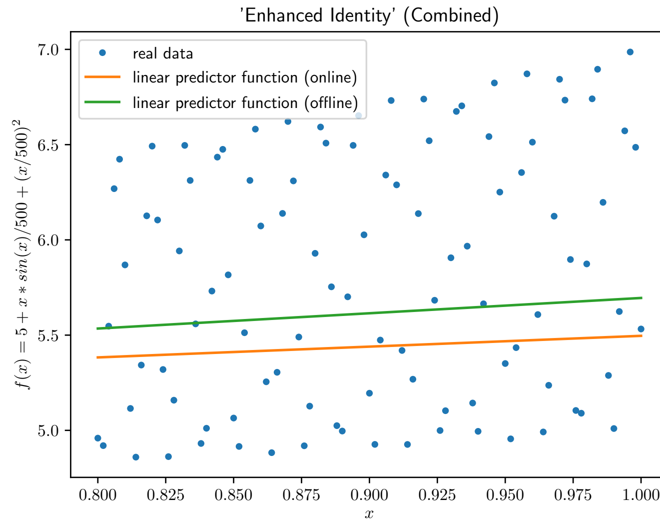


Figure 6.16: Online: $\eta = 8.5$; MSE ≈ 0.5526. Offline: without regularization ($\lambda = 0$); MSE ≈ 0.4504.

The limitations of LR, which could already be seen in the previous examples, are that at its core, it only assumes a linear dependency between input-output pairs and we need to "manually" linearize the original relationship, if it was nonlinear.

6.2 Reservoir Computing

Now that we have the linear readout tested, we shall also demonstrate the complete reservoir computing (ESN) functionality on the same set of examples.

All of the following examples have been conducted as 1-time-step ahead time series predictions, predicting the y-axis values. We've conducted both

online and offline training. In the online case, we've tried tuning the learning rate η solely by doing multiple runs and picking the most optimal one. If one played with the configuration long enough, it is possible that they could've gotten significantly better results for both offline and online case.

6.2.1 Glaciers

For the dataset description, see 6.1.1.

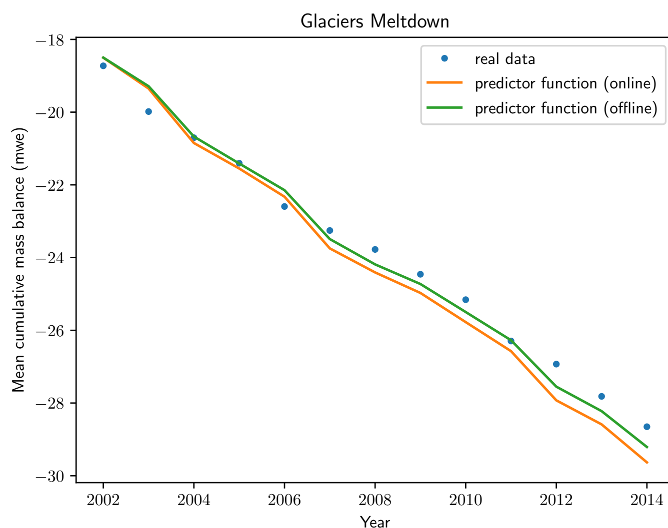


Figure 6.17: Using 80% of data for testing. A randomly good model (other runs with the same configuration performed worse for the online case). Online: $\eta = 50$, MSE ≈ 0.3444. Offline: MSE ≈ 0.1547. (Note: The MSEs were computed before output normalization was used, so they're not comparable with later examples.)

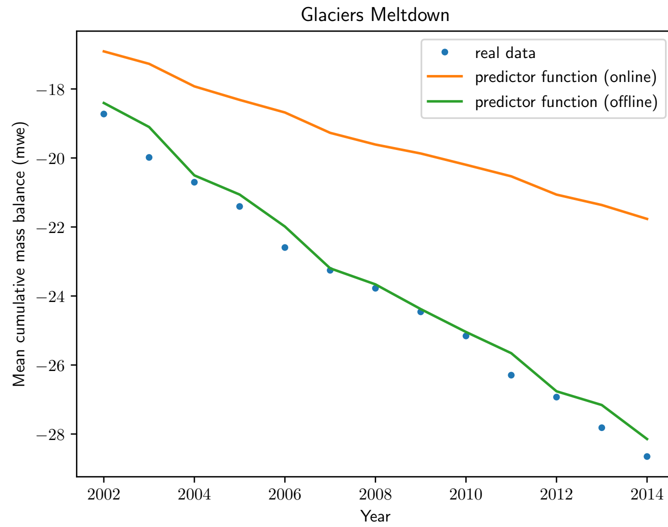


Figure 6.18: Using 80% of data for testing. Online: $\eta = 20$, MSE ≈ 0.2145. Offline: MSE ≈ 0.002.

6.2.2 CO_2 Emissions

For the dataset description, see 6.1.2.

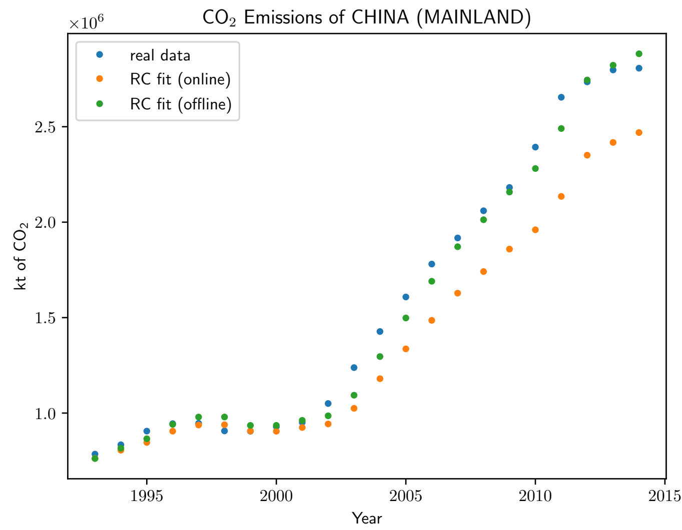


Figure 6.19: Online: $\eta = 10$, MSE ≈ 10.4774. Offline: MSE ≈ 0.897.

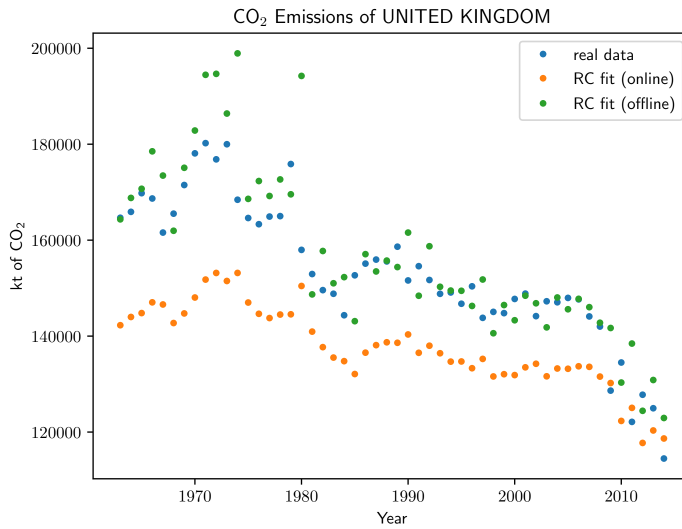


Figure 6.20: Online: $\eta = 10$, MSE ≈ 0.0479. Offline: MSE ≈ 0.0139.

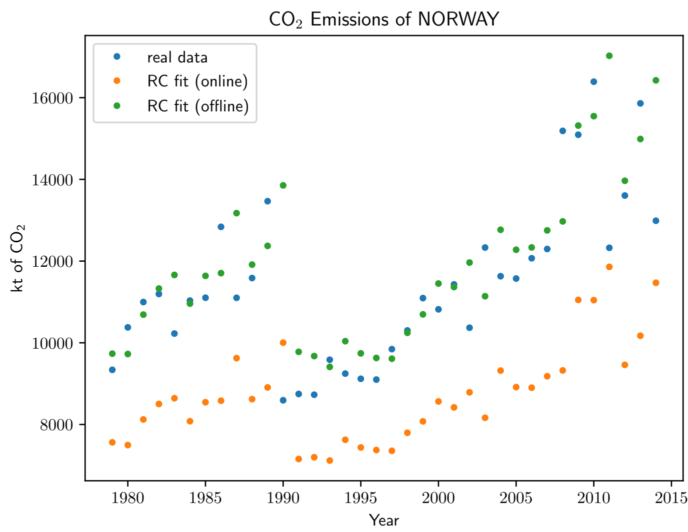


Figure 6.21: Online: $\eta = 10$, MSE ≈ 0.0015. Offline: MSE $\approx 3.91 \times 10^{-4}$.

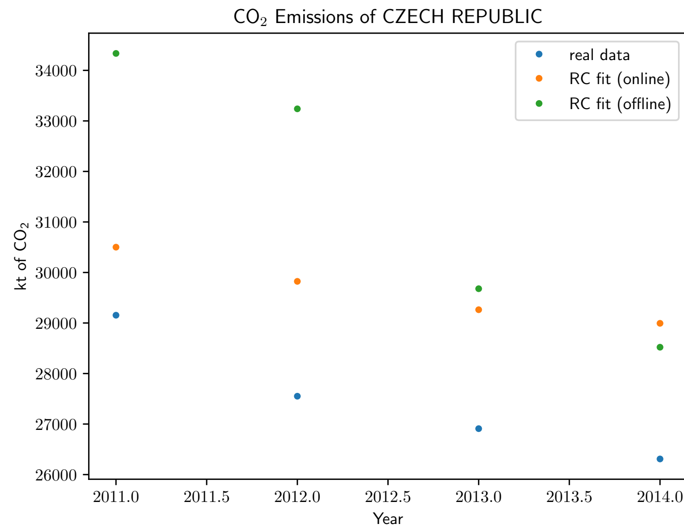


Figure 6.22: Online: $\eta = 10$, MSE $\approx 7.90 \times 10^{-4}$. Offline: MSE ≈ 0.0029.

6.2.3 $PM_{2.5}$ Pollution

For the dataset description, see 6.1.3.

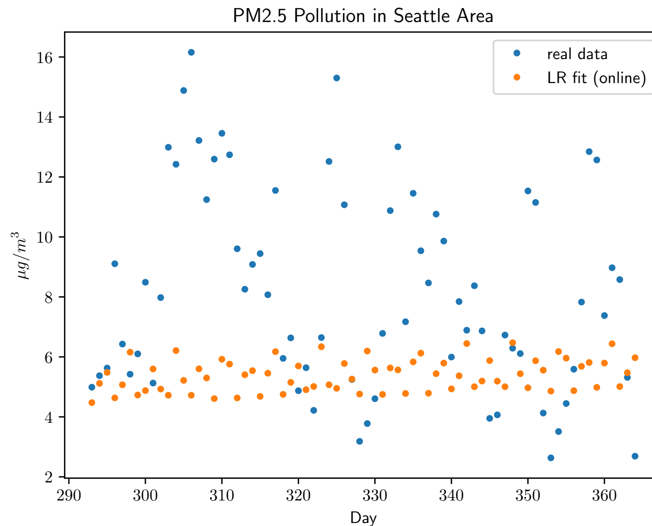


Figure 6.23: Online: $\eta = 200$, MSE ≈ 0.0083. Offline: MSE ≈ 0.0038.

6.3 Sensitivity Analysis of Reservoir Parameters

When manually choosing any of the RC (hyper)parameters, it is crucial to understand their impact. Here we'll analyze each hyperparameter separately

by changing its values with every other hyperparameter being fixed for that particular series of experiments.

We'll compare the performance of the resulting ESNs by plotting the obtained MSEs (lower MSE corresponds to a better model). We've used only offline learning (pseudoinverse), since the online (GD) version generally performs worse, and we usually have to search for optimal η, which is example-dependent. (Choosing a very low regularization factor μ in the offline case is much more stable.)

All example data (input and output) have been normalized to have values approximately from the range $[-1, 1]$.

All examples have been conducted as 1-time-step ahead time series predictions, like in the Reservoir Computing section (6.2).

We've done 10 experimental trials for each evaluated hyperparameter and example, and represented them through different colors in the graph. Each trial consists of one measurement for each value of the evaluated hyperparameter. The amount of trials is limited to ensure low computational time. These experiments therefore do not carry a statistical significance mainly to gain a rough understanding of how modifying each parameter can affect the model performance.

For the real-world examples, we've used an 80/20 training/testing set split ratio. In the Mackey-Glass time series example, we've used 2000 data samples for training, out of 10000 total samples, therefore using 20% of data for training (and 80% for testing).

Besides the aforementioned details, all measurements have been conducted using the default configuration described in Subsection 5.5.6.

6.3.1 Reservoir Size

Each colored line represents one experimental trial.

At first, we've used values of $N_{\mathrm{x}} \in \{50, 75, 100, 125\}$. But for some real-world examples, we can see a very high MSE. We've then tried to use a new set of values where $N_{\mathrm{x}} \in \{4, 10, 20, 50\}$. In the case of all real-world examples this seems more appropriate, giving lower errors. Probably because the sizes of the used datasets are very small (maximum of hundreds of samples in total). We've then adjusted for that by using $N_{\mathrm{x}} = 50$ for Mackey-Glass time series when analyzing other hyperparameters.

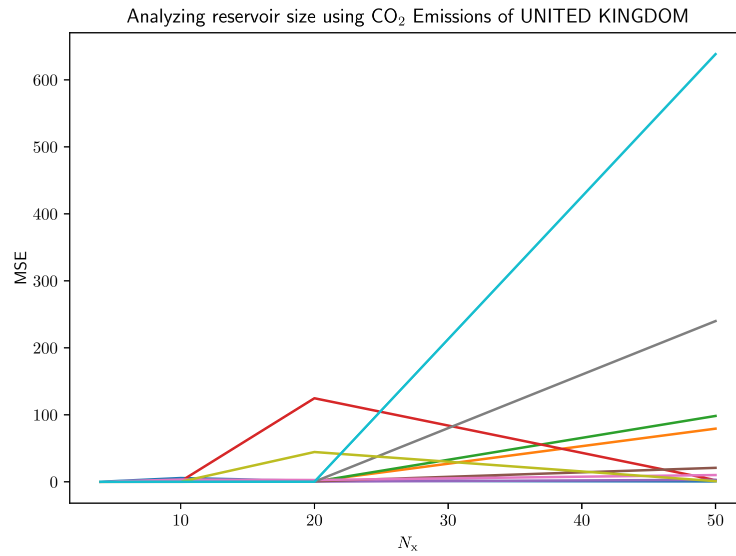


Figure 6.24: Analyzing the impact of reservoir size using CO_2 Emissions of UNITED KINGDOM data.

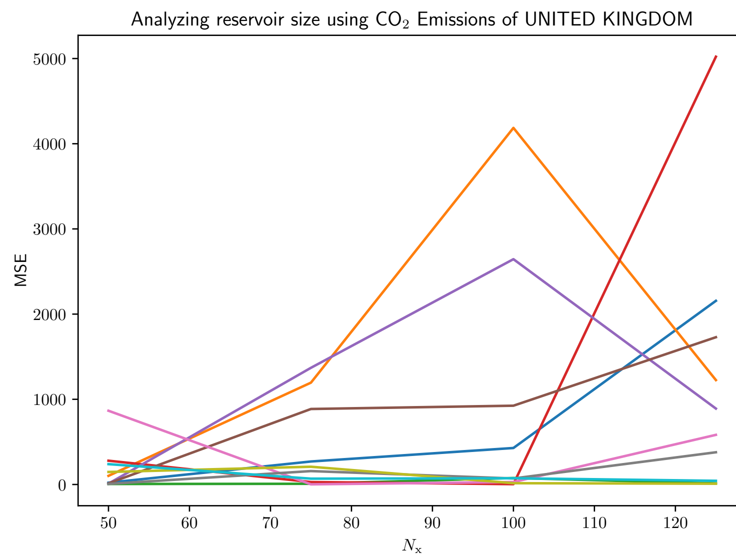


Figure 6.25: Analyzing the impact of (larger) reservoir size using CO_2 Emissions of UNITED KINGDOM data.

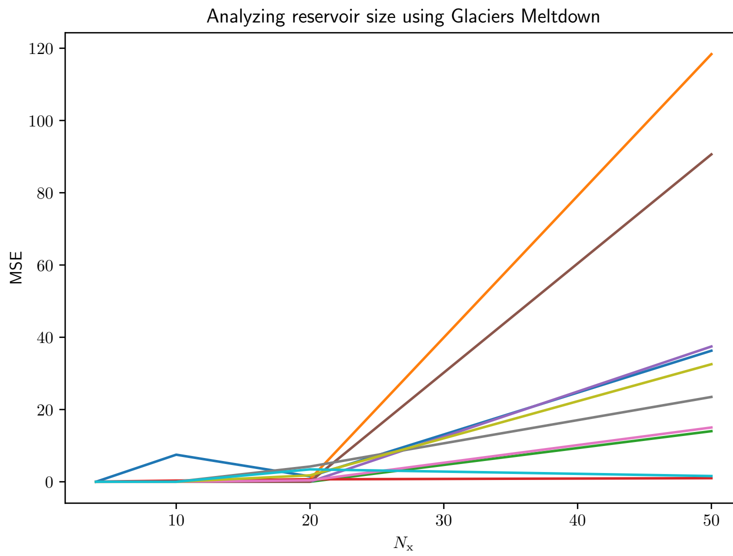


Figure 6.26: Analyzing the impact of reservoir size using Glaciers Meltdown data.

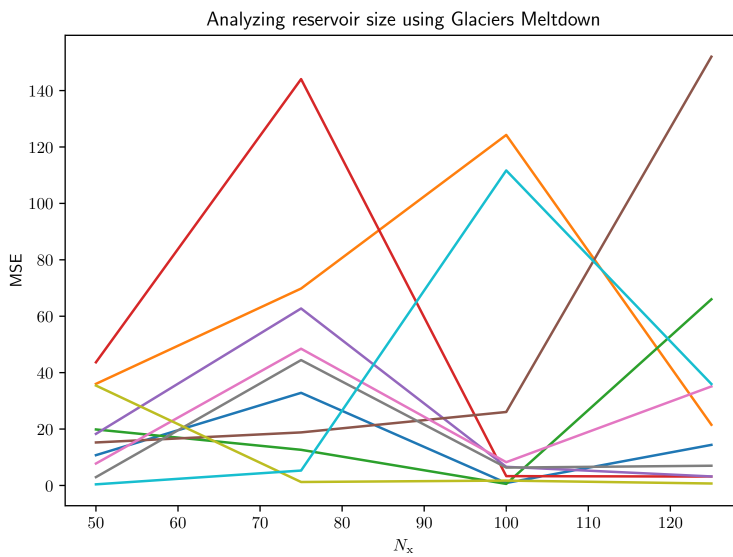


Figure 6.27: Analyzing the impact of (larger) reservoir size using Glaciers Meltdown data.

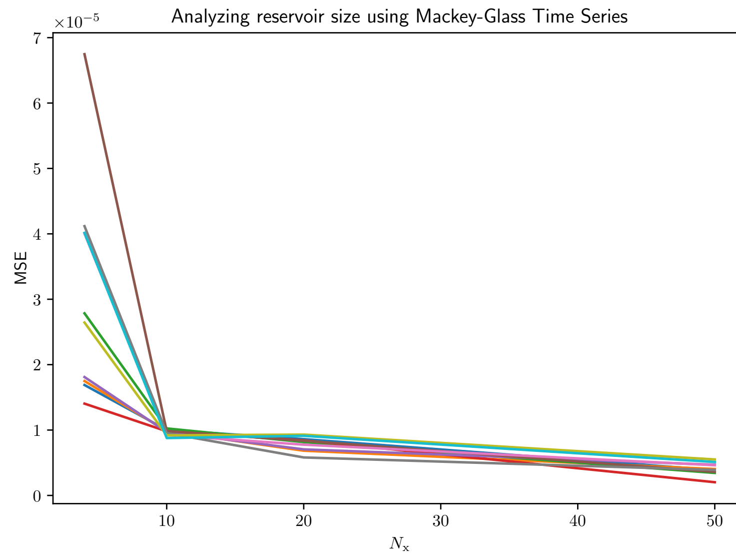


Figure 6.28: Analyzing the impact of reservoir size using Mackey-Glass Time Series data.

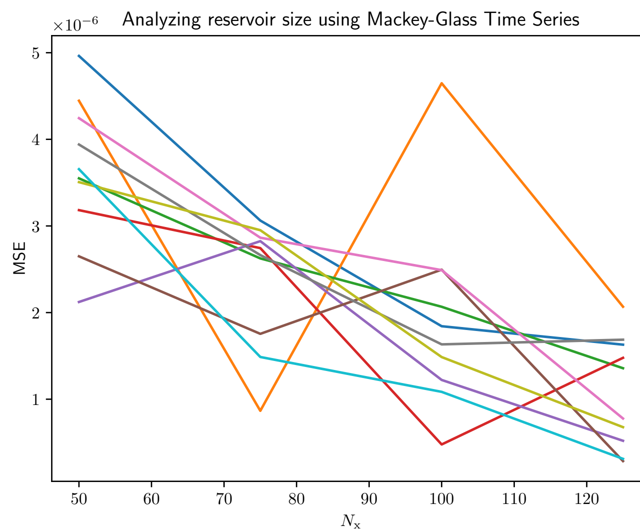


Figure 6.29: Analyzing the impact of (larger) reservoir size using Mackey-Glass Time Series data.

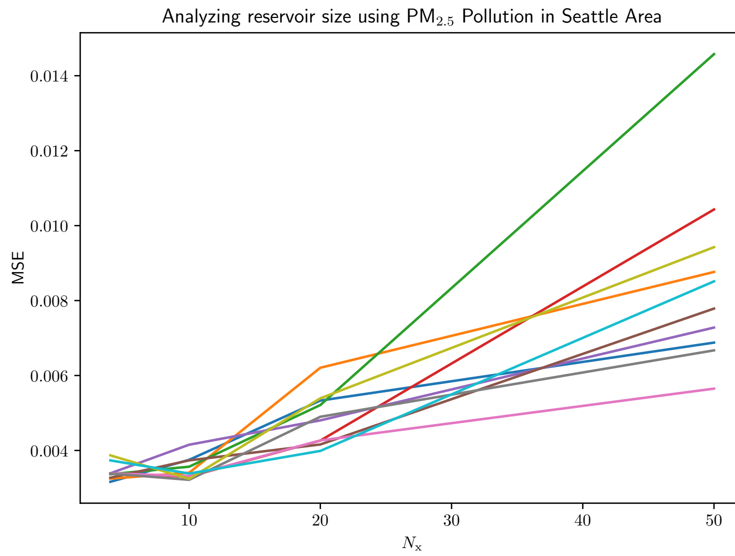


Figure 6.30: Analyzing the impact of reservoir size using $PM_{2.5}$ Pollution in Seattle Area data.

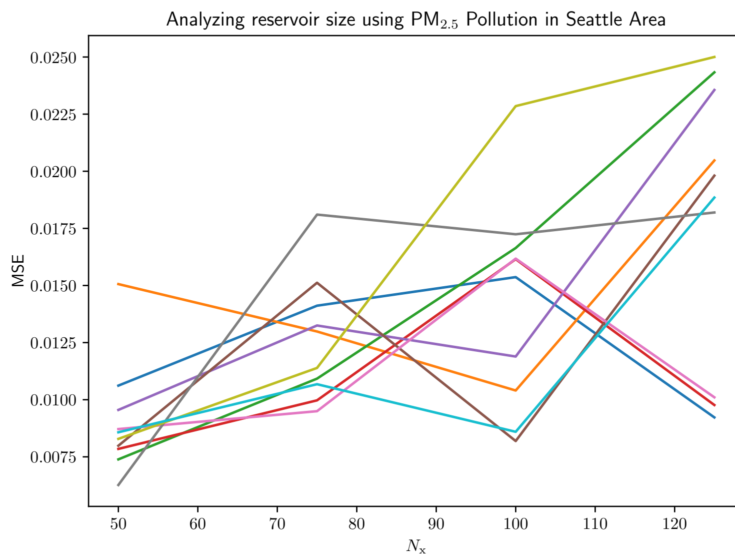


Figure 6.31: Analyzing the impact of (larger) reservoir size using $PM_{2.5}$ Pollution in Seattle Area data.

The pattern here is quite clear: almost all of the experiments give higher MSE values when increasing N_x, whereas the Mackey-Glass time series shows an opposite tendency.

6.3.2 Spectral Radius

Each colored line represents one experimental trial. We've used values of $\alpha \in \{0.1, 0.5, 0.8, 0.9\}$.

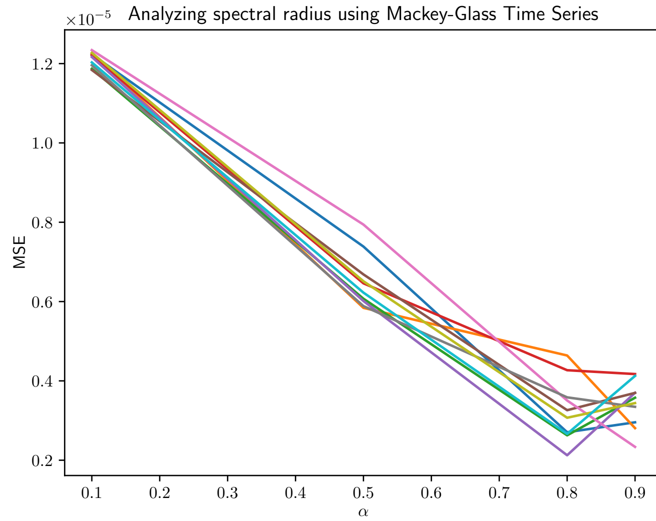


Figure 6.32: Analyzing the impact of spectral radius using Mackey-Glass Time Series data

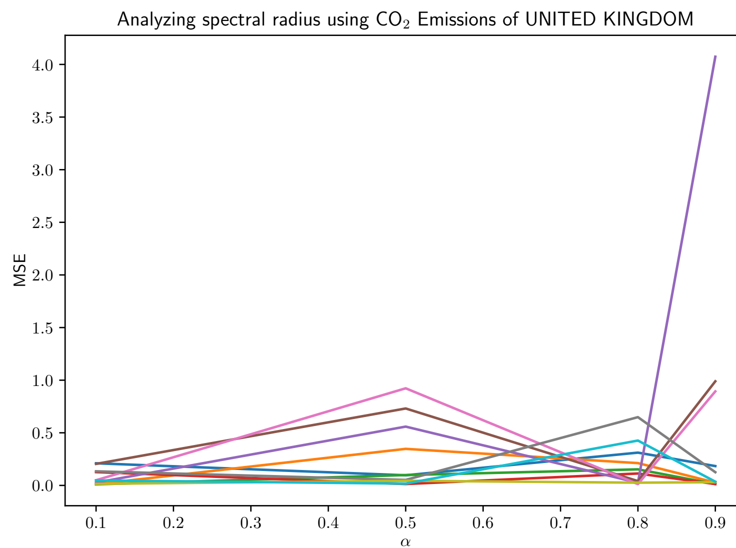


Figure 6.33: Analyzing the impact of spectral radius using CO_2 Emissions of UNITED KINGDOM data.

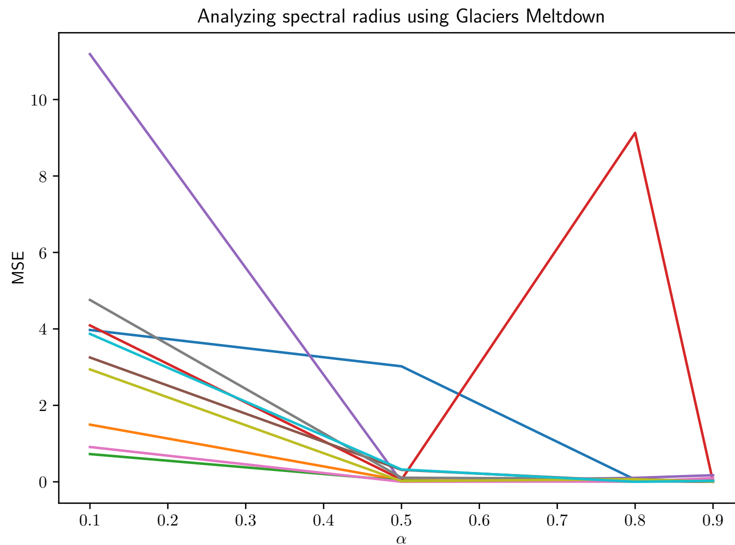


Figure 6.34: Analyzing the impact of spectral radius using Glaciers Meltdown

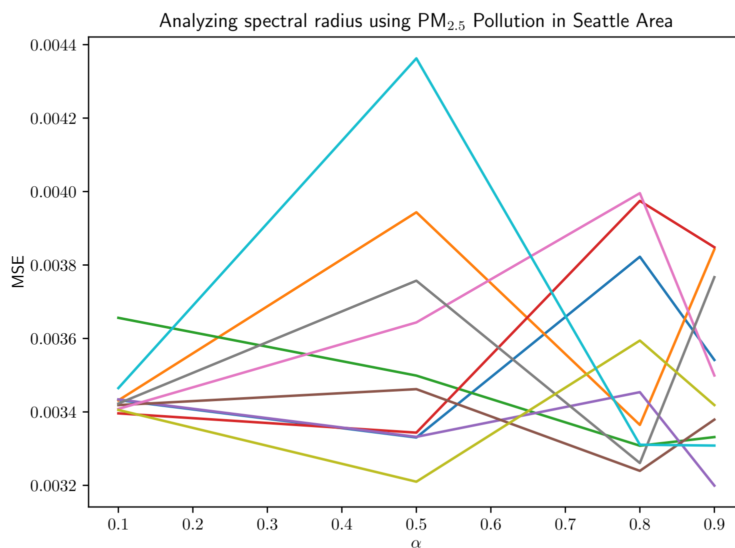


Figure 6.35: Analyzing the impact of spectral radius using $PM_{2.5}$ Pollution in Seattle Area data.

We can see a very clear tendency for Mackey-Glass and even for Glaciers and $PM_{2.5}$, where the optimal $\alpha(\rho)$ seems to lie somewhere between $0.8 - 0.9$. For the CO_2 Emissions of UK, a lower α might be preferred.

6.3.3 Reservoir Topology

Each color of dots represents one experimental trial.

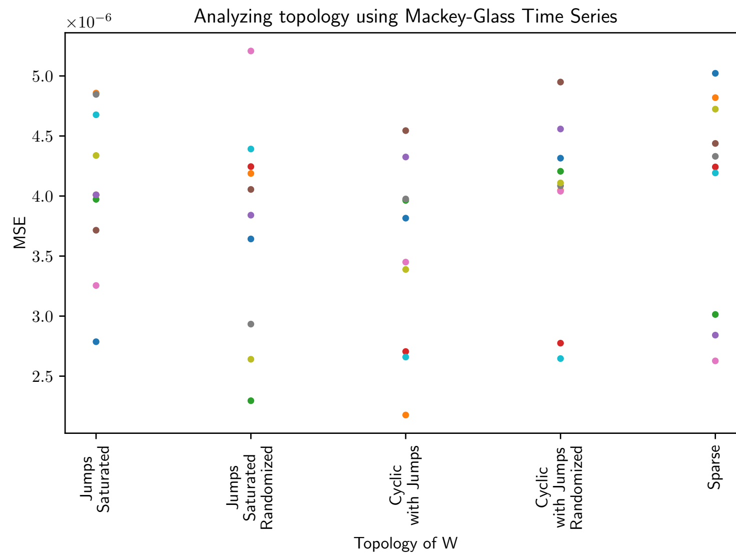


Figure 6.36: Analyzing the impact of changing the pattern of connectivity using Mackey-Glass Time Series data

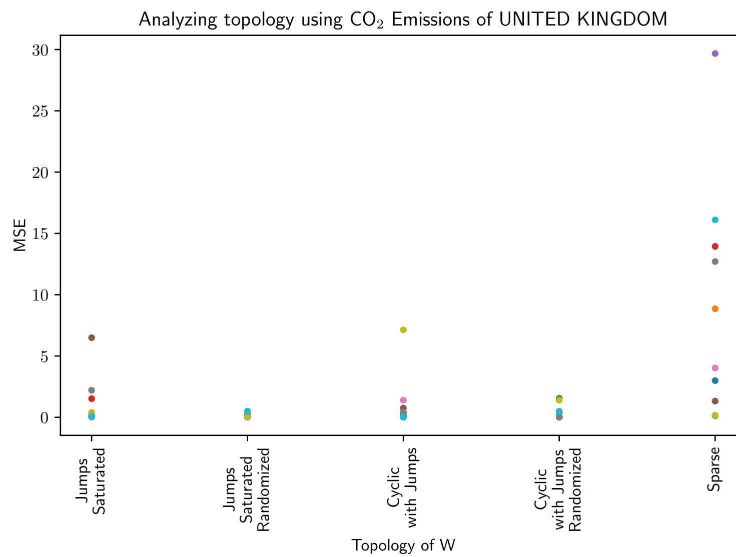


Figure 6.37: Analyzing the impact of changing the pattern of connectivity using CO_2 Emissions of UNITED KINGDOM data.

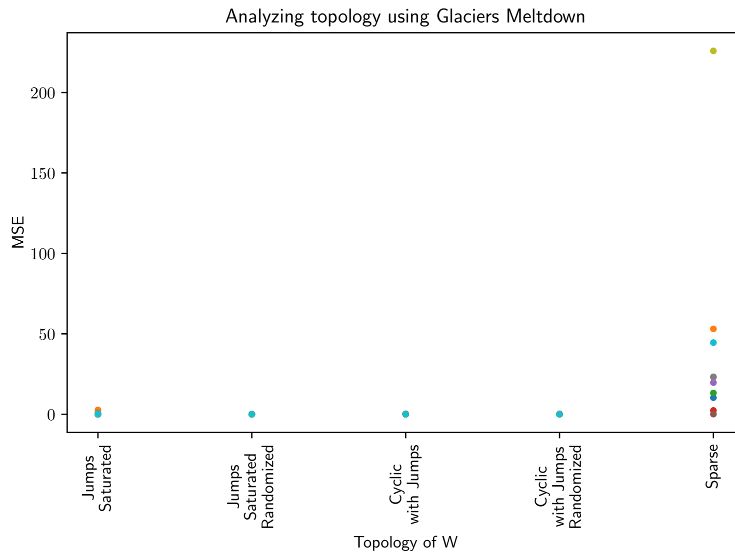


Figure 6.38: Analyzing the impact of changing the pattern of connectivity using Glaciers Meltdown

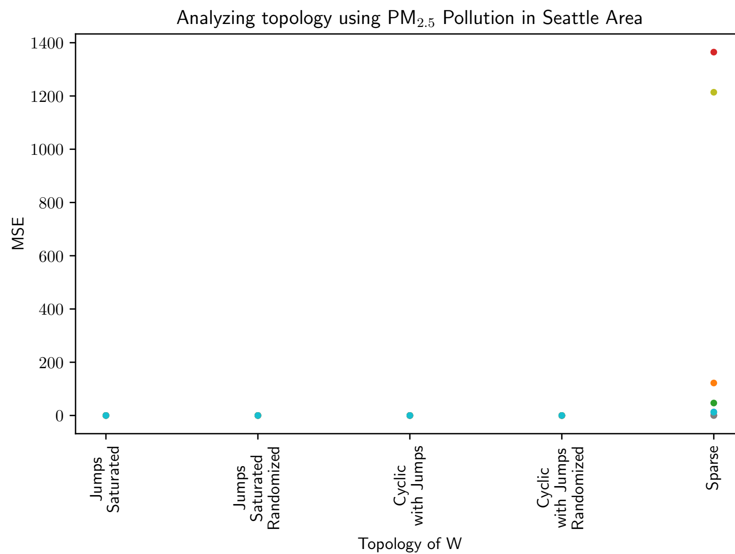


Figure 6.39: Analyzing the impact of changing the pattern of connectivity using $PM_{2.5}$ Pollution in Seattle Area data.

Unfortunately in some figures, we can't really see the details of MSE because the "Sparse" topology's values are too large. This topology (with sparsity set to 80%) apparently has the worst results. Amidst the deterministic topologies, "Cyclic with Jumps" performs reasonably well, but at this level of detail is comparable to the rest. We've therefore conducted the experiments again, while omitting the "Sparse topology".

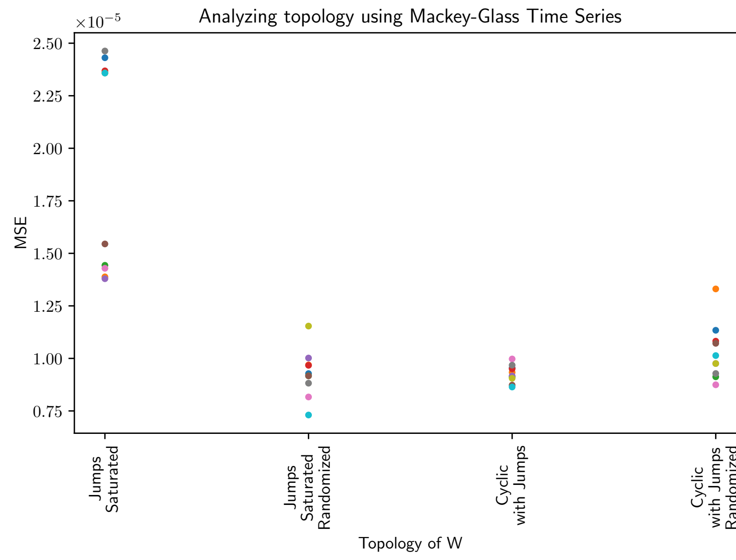


Figure 6.40: Analyzing the impact of changing the pattern of connectivity (excluding “Sparse”) using Mackey-Glass Time Series data

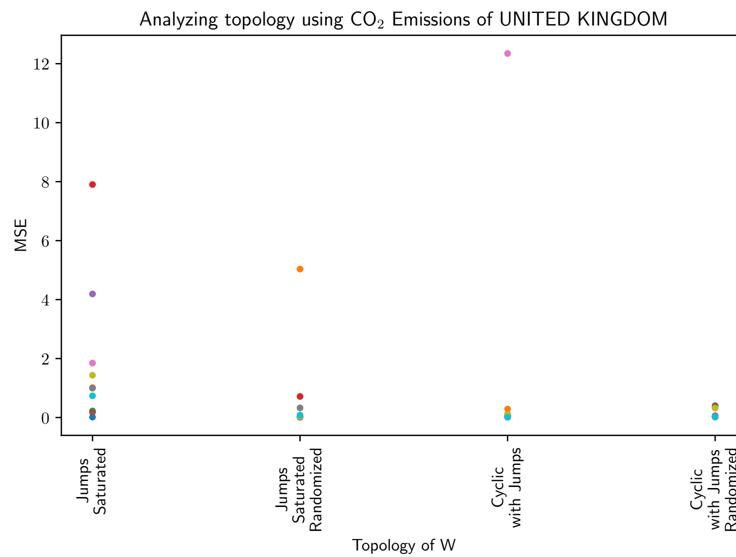


Figure 6.41: Analyzing the impact of changing the pattern of connectivity (excluding “Sparse”) using CO_2 Emissions of UNITED KINGDOM data.

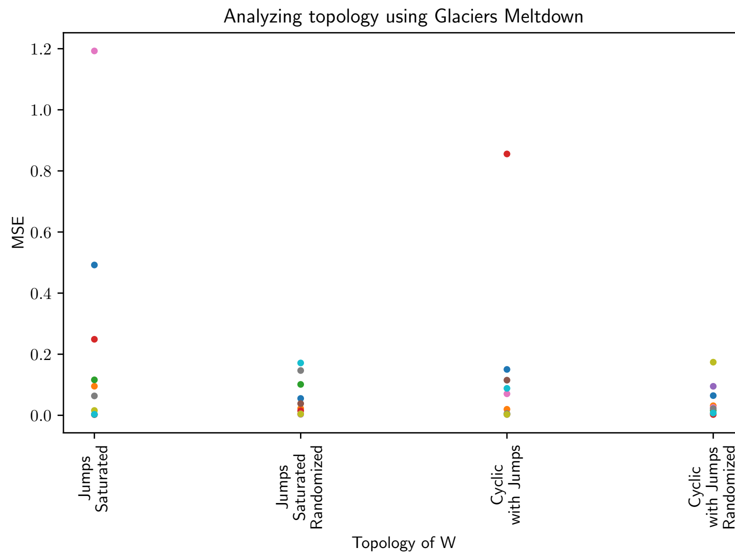


Figure 6.42: Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using Glaciers Meltdown

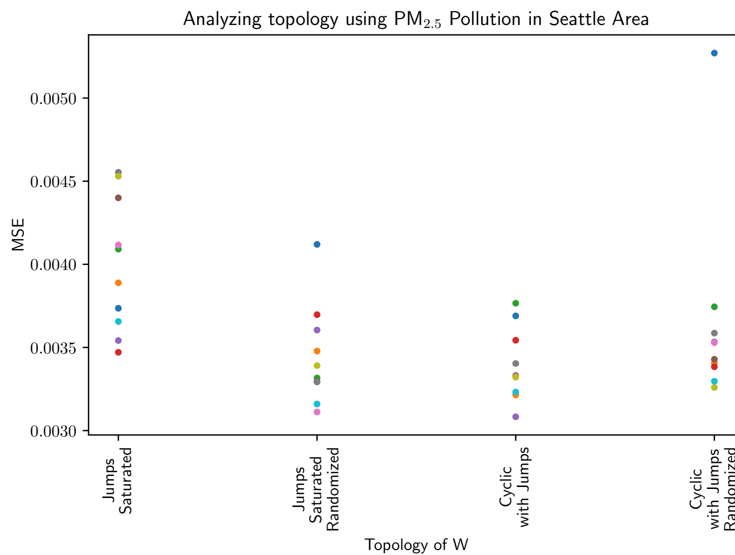


Figure 6.43: Analyzing the impact of changing the pattern of connectivity (excluding "Sparse") using $PM_{2.5}$ Pollution in Seattle Area data.

We can see that, when ignoring the outliers, all deterministic topologies make the model perform similarly well. There seems to be a bigger MSE for "Jumps Saturated" topologies, and "Cyclic with Jumps" still seems as being consistently the most usable.

6.4 Controlling Spectral Radius

To control the spectral radius (4.4) of W, we'll use the scaling hyperparameter α, which is equal to the spectral radius of the scaled W (4.5) [82, p. 28].

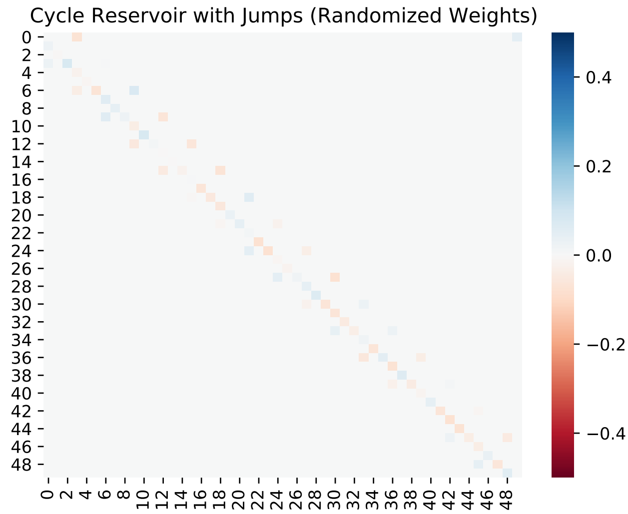


Figure 6.44: $\alpha = 0.1$

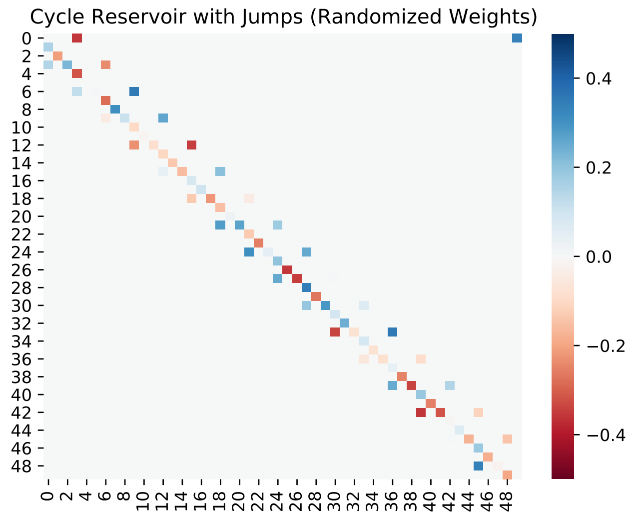


Figure 6.45: $\alpha = 0.4$

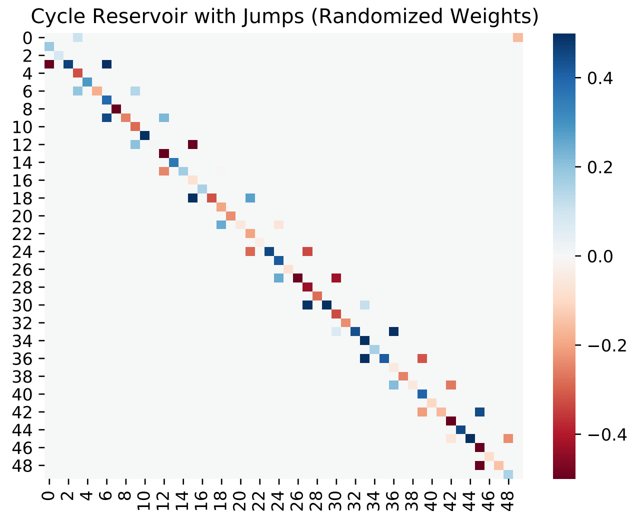


Figure 6.46: $\alpha = 0.6$

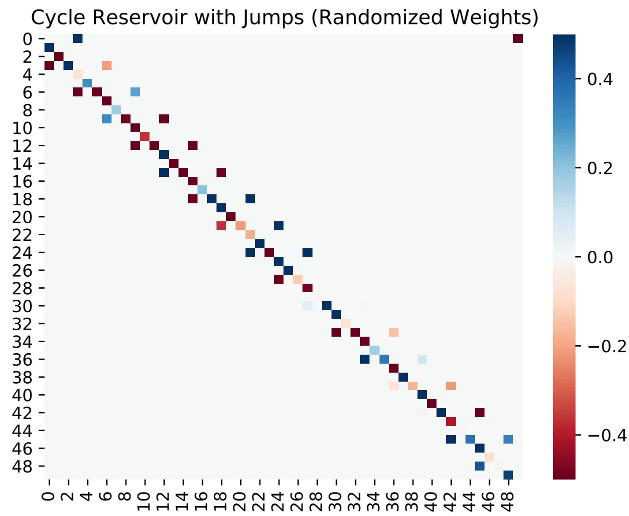


Figure 6.47: $\alpha = 0.9$

Larger values (darker colors) start to appear when setting larger α.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work, we have developed a library in Apache Flink (1.8) for the most popular Reservoir Computing model called Echo State Network. The library supports the use of Flink's DataStream and DataSet API (Core APIs) and it was implemented in Java. We have included several reservoir topologies such as standard sparse (with modifiable sparsity), "Cycle Reservoir with Jumps" and our own slight modifications such as "Jumps Saturated Reservoir". The user has the choice of training the model parameters using offline schema (Ridge Regression) or online (Stochastic Gradient Descent), together with an option to apply a step-based decay on the learning rate.

We've visually demonstrated the basic functionality of both the Linear Regression (readout part of the Reservoir Computing) and the whole (reservoir and readout) Reservoir Computing model on developed examples over real-world and function-generated datasets. We have also performed experimental comparisons of chosen hyperparameters' values on a small scale.

We could see that using just Linear Regression is much more straightforward, as it involves less parameters and no hidden state, but it can produce results with bad accuracy when estimating data with non-linear relationships. Still, a concrete Reservoir Computing model is less complicated to build in comparison to standard Recurrent Neural Networks, and we've tried to make this process easier by including a default configuration (as part of the examples framework).

7.2 Recommendation for Future Extensions

With regards to generalizing concepts, some possibilities have been noted throughout the text. We could think of creating a general neural network library with different predefined types of ANNs. The output of RC could be made such that it belongs to a multidimensional space (now the output space is the real domain). That is, we would predict multiple output features at once.

Regarding Java implementation, we could support a more general data type for the input/output vector elements, like a `Number` or `Comparable`, instead

of just `double`.

The code could always be more user-friendly (convenient) and provide even more customization of calls. But at some point, a library that is concise, yet clear and effective might be preferable. There are several other RC models that haven't been included in this library, such as: hierarchical reservoirs, self-organized reservoirs, cascade reservoirs, and so on. They could be included in the future by extending our developed library. We could also offer alternative types of error measurement apart from MSE, such as cross-entropy and normalized root mean squared error.

Bibliography

[1] Mantas Lukoševičius and Herbert Jaeger. "Reservoir Computing Approaches to Recurrent Neural Network Training". In: *Computer Science Review* 3.3 (Aug. 2009), pp. 127–149. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2009.03.005.

[2] Shaoxuan Wang. *Remove the legacy flink-libraries/flink-ml*. URL: https://issues.apache.org/jira/browse/FLINK-12597.

[3] Ali Rodan and Peter Tiňo. "Simple Deterministically Constructed Cycle Reservoirs with Regular Jumps". In: *Neural Computation* 24.7 (2012). PMID: 22428595, pp. 1822–1852. DOI: 10.1162/NECO_a_00297. URL: https://www.cs.bham.ac.uk/~pxt/PAPERS/esn_jumps.pdf.

[4] Cambridge Dictionary. *Meaning of data in English*. URL: https://dictionary.cambridge.org/dictionary/english/data (visited on 05/11/2020).

[5] E. Udoh. *Handbook of Research on Grid Technologies and Utility Computing: Concepts for Managing Large-Scale Applications*. Information Science Reference, 2009. ISBN: 9781605661858. URL: https://books.google.cz/books?id=sQ6IKhTlzyIC.

[6] Cambridge Dictionary. *Meaning of sequential in English*. URL: https://dictionary.cambridge.org/dictionary/english/sequential (visited on 05/11/2020).

[7] Institute for Telecommunication Sciences. *Definition: data stream*. URL: https://www.its.bldrdoc.gov/fs-1037/dir-010/_1451.htm.

[8] Techopedia. *What is Data Streaming?* URL: https://www.techopedia.com/definition/13604/data-streaming.

[9] Attunity. *What is Data Streaming*. URL: https://www.attunity.com/what-is-data-streaming/.

[10] Ververica. *What is Stream Processing?* URL: https://www.ververica.com/what-is-stream-processing.

[11] Microsoft Docs. *Real time processing*. URL: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/real-time-processing.

[12] Techopedia. *What is Real-Time Data Processing?* URL: https://www.techopedia.com/definition/31742/real-time-data-processing.

[13] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing". In: *ACM Sigmod Record* 34.4 (2005), pp. 42–43.

[14] Chris Chatfield. *Time-series forecasting*. CRC press, 2000.

[15] Sebastián Basterrech. "Pattern Matching in Sequential Data Using Reservoir Projections". In: June 2019, pp. 173–183. ISBN: 978-3-030-22795-1. DOI: 10.1007/978-3-030-22796-8_19.

[16] Pierre Geurts. "Pattern Extraction for Time Series Classification". In: *Principles of Data Mining and Knowledge Discovery*. Ed. by Luc De Raedt and Arno Siebes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 115–127. ISBN: 978-3-540-44794-8.

[17] Haibin Cheng et al. "Multistep-Ahead Time Series Prediction". In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Wee-Keong Ng et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 765–774. ISBN: 978-3-540-33207-7. DOI: 10.1007/11731139_89.

[18] The Apache Software Foundation. *What is Apache Flink? — Architecture*. URL: https://flink.apache.org/flink-architecture.html.

[19] Z. Karakaya, A. Yazici, and M. Alayyoub. "A Comparison of Stream Processing Frameworks". In: *2017 International Conference on Computer and Applications (ICCA)*. 2017, pp. 1–12.

[20] G. van Dongen and D. Van den Poel. "Evaluation of Stream Processing Frameworks". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (Aug. 2020), pp. 1845–1858. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.2978480.

[21] Shivangi Gupta. *Streaming in Spark, Flink, and Kafka*. June 18, 2017. URL: https://dzone.com/articles/streaming-in-spark-flink-and-kafka-1 (visited on 05/20/2020).

[22] Michael G. Noll (https://stackoverflow.com/users/1743580/michael-g-noll). *Answer to: master node in multi-node kafka cluster*. (version: 2016-04-12). URL: https://stackoverflow.com/a/36571070.

[23] The Apache Software Foundation. *Apache Flink Concepts*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.0/concepts/concepts.html (visited on 05/20/2020).

[24] The Apache Software Foundation. *Apache Kafka*. URL: https://kafka.apache.org/intro (visited on 05/20/2020).

[25] Tomáš Werner. *Optimalizace*. (In Czech). Jan. 28, 2019. URL: https://cw.fel.cvut.cz/b181/_media/courses/b33opt/opt.pdf (visited on 05/21/2020).

[26] Xin Yan and Xiaogang Su. *Linear regression analysis: theory and computing*. World Scientific, 2009.

[27] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996. DOI: 10.1017/CBO9780511812651.

[28] Mark Schmidt. *CPSC 540: Machine Learning - Group L1-Regularization, Proximal-Gradient*. University of British Columbia, 2017. URL: https://www.cs.ubc.ca/~schmidtm/Courses/540-W17/L5.pdf (visited on 03/05/2020).

[29] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of machine learning research* 13.Feb (2012), pp. 281–305.

[30] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: https://arxiv.org/pdf/1609.04747.pdf.

[31] angryavian (https://math.stackexchange.com/users/43949/angryavian). *Why divide by* $2m$. Mathematics Stack Exchange. (version: 2014-08-01). URL: https://math.stackexchange.com/q/884903.

[32] R Osuna-Gómez, A Rufián-Lizana, and P Ruiz-Canales. "Invex functions and generalized convexity in multiobjective programming". In: *Journal of optimization theory and applications* 98.3 (1998), pp. 651–661. DOI: https://doi.org/10.1023/A:1022628130448.

[33] Yoshiyasu Takefuji. "A Feedforward neural network is a subset of a recurrent neural network". In: *Science* (Oct. 2018).

[34] Bing Xu, Ruitong Huang, and Mu Li. *Revise Saturated Activation Functions*. 2016. arXiv: 1602.05980 [cs.LG].

[35] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012. URL: https://www.isca-speech.org/archive/interspeech_2012/i12_0194.html.

[36] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[37] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. "An overview of reservoir computing: theory, applications and implementations". In: *Proceedings of the 15th european symposium on artificial neural networks*. 2007, pp. 471–482. URL: https://biblio.ugent.be/publication/416607/file/447949.

[38] Hélene Paugam-Moisy and Sander M Bohte. "Computing with spiking neuron networks." In: *Handbook of natural computing* 1 (2012), pp. 1–47.

[39] Herbert Jaeger. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.34 (Jan. 26, 2010).

[40] A. Rodan and P. Tiňo. "Minimum Complexity Echo State Network". In: *IEEE Transactions on Neural Networks* 22.1 (Jan. 2011), pp. 131–144. ISSN: 1941-0093. DOI: 10.1109/TNN.2010.2089641.

[41] Apache Flink Documentation. *Apache Flink Documentation.* URL: https://ci.apache.org/projects/flink/flink-docs-stable/ (visited on 05/21/2020).

[42] *Apache Flink - Apache Project Information.* URL: https://projects.apache.org/project.html?flink (visited on 05/21/2020).

[43] Juan Soto. *A Historical Account of Apache Flink: Its Origins, Growing Community, and Global Impact.* Technische Universität Berlin, May 31, 2016. URL: https://www.dima.tu-berlin.de/fileadmin/fg131/Informationsmaterial/Apache_Flink_Origins_for_Public_Release.pdf (visited on 05/11/2020).

[44] *Stratosphere.* URL: http://stratosphere.eu/ (visited on 05/21/2020).

[45] Daniel Warneke and Odej Kao. "Nephele: Efficient Parallel Data Processing in the Cloud". In: *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers.* MTAGS '09. Portland, Oregon: ACM, 2009, 8:1–8:10. ISBN: 978-1-60558-714-1. DOI: http://doi.acm.org/10.1145/1646468.1646476.

[46] *Stratosphere accepted as Apache Incubator Project.* Apr. 16, 2014. URL: http://stratosphere.eu/blog/apache/2014/04/16/stratosphere-goes-apache-incubator.html.

[47] *Announcing Apache Flink 1.0.0.* Mar. 8, 2016. URL: https://flink.apache.org/news/2016/03/08/release-1.0.0.html.

[48] Wikipedia contributors. *Apache Flink — Wikipedia, The Free Encyclopedia.* May 4, 2020. URL: https://en.wikipedia.org/w/index.php?title=Apache_Flink&oldid=954804006 (visited on 05/10/2020).

[49] The Apache Software Foundation. *Apache Flink Roadmap.* URL: https://flink.apache.org/roadmap.html (visited on 05/10/2020).

[50] Jincheng Sun. *FLIP-38: Python Table API.* URL: https://cwiki.apache.org/confluence/display/FLINK/FLIP-38%3A+Python+Table+API (visited on 05/09/2020).

[51] Apache Flink 1.8 Documentation. *Dataflow Programming Model.* URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/programming-model.html.

[52] Apache Flink 1.8 Documentation. *SQL.* URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/table/sql.html.

[53] Apache Flink 1.8 Documentation. *Basic API Concepts.* URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/api_concepts.html.

[54] Apache Flink 1.8 Documentation. *Java Lambda Expressions.* URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/java_lambdas.html.

[55] Ververica. *Apache Flink Training - DataSet API Basics Slides*. URL: https://www.slideshare.net/dataArtisans/flink-training-dataset-api-basics.

[56] Apache Flink 1.10 Documentation. *DataStream API Tutorial*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.10/getting-started/tutorials/datastream_api.html.

[57] Apache Flink 1.8 JavaDoc. *org.apache.flink.api.java.DataSet*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/api/java/org/apache/flink/api/java/DataSet.html.

[58] Apache Flink 1.8 Documentation. *Table API*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/table/tableApi.html.

[59] Apache Flink 1.8 JavaDoc. *org.apache.flink.streaming.api.environment.StreamExecutionEnvironment*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/api/java/org/apache/flink/streaming/api/environment/StreamExecutionEnvironment.html.

[60] Apache Flink 1.8 JavaDoc. *org.apache.flink.api.java.ExecutionEnvironment*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/api/java/org/apache/flink/api/java/ExecutionEnvironment.html.

[61] Apache Flink 1.8 Documentation. *Flink DataSet API Programming Guide*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/batch/index.html.

[62] Apache Flink 1.8 Documentation. *Streaming Connectors*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/connectors/index.html.

[63] Apache Flink 1.8 Documentation. *DataStream API Programming Guide*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/datastream_api.html.

[64] Fabian Hueske. *Answer to: Apache Flink Process Stream Multiple Times*. URL: https://stackoverflow.com/a/44647003/4584464.

[65] Stephan Ewen. *FLIP-21 - Improve object Copying/Reuse Mode for Streaming Runtime*. URL: https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=71012982.

[66] Apache Flink 1.8 Documentation. *DataSet Transformations*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/batch/dataset_transformations.html.

[67] Apache Flink 1.8 Documentation. *DataStream API: Operators*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/stream/operators/.

[68] Apache Flink 1.8 Documentation. *Iterations*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/batch/iterations.html (visited on 08/20/2019).

[69] The Apache Software Foundation. *Apache Flink*. Version 1.8. Apr. 9, 2019. URL: https://flink.apache.org/.

[70] Apache Flink 1.8 Documentation. *Event Time*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/event_time.html.

[71] Apache Flink 1.8 Documentation. *Pre-defined Timestamp Extractors / Watermark Emitters*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/event_timestamp_extractors.html.

[72] Data Arisans alpinegizmo. *Apache Flink training video about event time Watermarks*. Youtube. Nov. 25, 2018. URL: https://www.youtube.com/watch?v=zL5JWWgm3xA.

[73] Apache Flink 1.8 Documentation. *Windows*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/operators/windows.html (visited on 05/22/2020).

[74] Apache Flink 1.8 Documentation. *The Broadcast State Pattern*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/stream/state/broadcast_state.html.

[75] Blaise Barney. *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/tutorials/parallel_comp.

[76] Apache Flink 1.8 Documentation. *Parallel Execution*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/parallel.html.

[77] Raphael Yuster and Uri Zwick. "Fast Sparse Matrix Multiplication". In: *ACM Trans. Algorithms* 1.1 (July 2005), p. 5. ISSN: 1549-6325. DOI: 10.1145/1077464.1077466. URL: http://doi.acm.org/10.1145/1077464.1077466.

[78] Peter Abeles. *Java Matrix Benchmark*. URL: https://lessthanoptimal.github.io/Java-Matrix-Benchmark/.

[79] Optimatika. *oj! Algorithms*. URL: https://www.ojalgo.org/.

[80] Prof. William Kahan. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. URL: https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

[81] Apache Flink Documentation. *Data Types & Serialization*. URL: https://ci.apache.org/projects/flink/flink-docs-stable/dev/types_serialization.html (visited on 05/01/2020).

[82] H. Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach*. GMD-Report 159. Fraunhofer Institute for Autonomous Intelligent Systems (AIS), Dec. 2013. URL: http://minds.jacobs-university.de/uploads/papers/ESNTutorialRev.pdf.

[83] World Glacier Monitoring Service. *Global Glacier Change Bulletin No. 1 (2012–2013)*. 2015. URL: https://wgms.ch/downloads/wgms_2013_gmbb12.pdf (visited on 12/10/2019).

[84] United States Environmental Protection Agency. *Climate Change Indicators: Glaciers*. Aug. 2016. URL: https://www.epa.gov/climate-indicators/climate-change-indicators-glaciers (visited on 12/10/2019).

[85] G. Marland T.A. Boden and R.J. Andres. *Global, Regional, and National Fossil-Fuel CO2 Emissions (1751 - 2014)*. Oak Ridge, Tennessee, U.S.A.: Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, U.S. Department of Energy, 2017. DOI: 10.3334/CDIAC/00001_V2017. (Visited on 01/25/2020).

[86] Kedar Potdar, Taher S Pardawala, and Chinmay D Pai. "A comparative study of categorical variable encoding techniques for neural network classifiers". In: *International journal of computer applications* 175.4 (Oct. 2017), pp. 7–9. DOI: 10.5120/ijca2017915495. (Visited on 01/25/2020).

[87] Yu-Fei Xing et al. "The impact of PM2. 5 on the human respiratory system". In: *Journal of Thoracic Disease* 8.1 (Jan. 2016), E69. DOI: 10.3978/j.issn.2072-1439.2016.01.19.

[88] *Air Quality System Data Mart (internet database)*. US Environmental Protection Agency. URL: https://www.epa.gov/outdoor-air-quality-data/download-daily-data (visited on 01/23/2020).

Acronyms

AI Artificial Intelligence 17

ANN Artificial Neural Network 17

CRJ Cycle Reservoir With Jumps 22

ESN Echo State Network 19

GD Gradient Descent 12

JVM Java Virtual Machine 26

LM Linear Model 9

LoAs Levels Of Abstraction 26

LR Linear Regression 9

LSM Liquid State Machine 19

LSTM Long Short-Term Memory 18

ML Machine Learning 17

MSE Mean Squared Error 14

RC Reservoir Computing 18

RNN Recurrent Neural Network 18

SCR Simple Cycle Reservoir 21

SGD Stochastic Gradient Descent 12