# Een uitgebreid theoretisch kader voor Reservoir Computing

# Expanding the Theoretical Framework of Reservoir Computing

Michiel Hermans

Promotor: prof. dr. ir. B. Schrauwen
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012 - 2013

UNIVERSITEIT
GENT

ISBN 978-90-8578-555-2
NUR 919
Wettelijk depot: D/2012/10.500/81

# Dankwoord

De laatste vier jaar zijn een ervaring geweest waar ik als student niet van had durven dromen. De kans om ‘je eigen ding te doen’ is niet iedereen gegunt, en ik ben er dan ook erg dankbaar om. Mijn dank gaat uit naar een hele hoop mensen. In eerste plaats mijn promotor Ben Schrauwen, die altijd een rijke bron van ideëen was en mij altijd onvoorwaardelijke steun gaf. Grote dank gaat ook uit aan de mensen die mijn tekst grondig nagelezen hebben. Joni Dambre en Jan Van Campenhout, bedankt om het taalniveau en wetenschappelijk niveau van mijn thesis significant te verbeteren!

*I would like to thank the jury for reading and commenting on my thesis, giving valuable advice and lifting the text up to its current level.*

Ik denk dat ik het bijzonder goed getroffen heb met mijn naaste collega’s. Het volledige traject is Pieter Buteneers mijn trouwe kantoorgenoot gebleven, van toen we nog in de *konijnepijp* zaten, tot we opgewaardeerd werden tot het hoekkantoor met panoramisch zicht over het water en Tim Waegeman ons vergezelde. Pieter en Tim, merci om jullie ongelooflijke behulpzaamheid, het verdragen van mijn driftbuien als mijn PC weer eens niet wou doen wat hij moest doen, en om er een plezante tijd van te maken.
Verder gaat mijn dank ook in het bijzonder uit naar collega Philémon Brakel. Overgoten met veel Orval (na de symbolische koffie verkeerd) heb ik met hem vele interessante gesprekken gehad over Machine Learning, en mij in’t algemeen bijzonder goed geamuseerd.
De rest van mijn collega’s mogen niet onvermeld blijven natuurljk. Francis, met wie ik samen het vak Ingenieursproject I begeleidde, David, de vaderlijke *ancien* van het lab, Sander Dieleman die me geholpen heeft Theano te installeren op mijn kraaknieuw rekenmachien, Ken, Pieter-Jan, Aäron en Jonas.

*After some years of frequent travels and due to the international character of our lab, I have gotten to know a few foreigners. First of all, Fionntann, my ex-colleague: I've had great fun hanging out with you in Lausanne and other places. John Butcher, a regular visitor of our lab and an overall great bloke, I've had a good time hanging out with you in Barcelona. Mantas Lukosevicius, thanks for the great discussions on science and science fiction.*

Waar het naaste vrienden betreft, t'zou een schande zijn om jullie hier niet te vermelden. Aan Matias, Brahim, Tom, Karim, Michiel Sanne (ge had eerder moeten zeggen dat het geen zin heeft om *voor* uw manneke te blijven), Mouctar, Clara,... merciekes voor het vele plezier dat ik met jullie heb beleeft de laatste jaren! Verder nog een dikke merci aan mijn pa en Ine voor het organiseren van de receptie, mijn ma, Martje, mijn schoonbroer Michael, en mijn piepjonge neefjes.

*Danuta, thank you for always being there for me. You're the best girlfriend I could hope for.*

Michiel Hermans

Dit werk is ondersteund door het Instituut voor de Aanmoediging van Innovatie door Wetenschap en Technologie Vlaanderen (IWT Vlaanderen).

*This work was supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen).*

# Examencommissie

# Samenvatting

## Reservoir Computing

Het laatste decennium wordt gekenmerkt door een sterke toename in rekenvermogen, hetgeen de opkomst van het onderzoeksdomein van *Machinaal Leren* (ML) mogelijk maakte. Het doel is om problemen in dataverwerking op te lossen die zeer moeilijk of zelfs onmogelijk met klassieke programma’s kunnen worden benaderd, maar gewoonlijk door menselijke wezens als eenvoudig of zelfs triviaal worden aanzien. Voorbeelden hiervan zijn onder meer spraakverwerking, gezichtsherkenning, en de motorcontrole vereist om op twee benen te lopen. In plaats van een expliciete reeks instructies te volgen zal ML trachten de te verwerken data te modelleren met een parametriseerbaar wiskundig model, hetgeen dan geoptimaliseerd kan worden om een taak zo goed mogelijk op te lossen.

Eén van de hoofdtypes van ML modellen zijn zogenaamde *Neurale Netwerken* (NN). Ze vormen een sterk geconceptualiseerde representatie van de functie van hersencellen, en bestaan uit een groot aantal neuronen (elementaire verwerkingseenheden) die onderling met elkaar verbonden zijn. Neurale netwerken bestaan al verschillende decennia en hebben bewezen een krachtige oplossing te bieden voor een variëteit aan applicaties.

Een specifieke variant van NNs zijn *recurrente* NNs (RNN). Deze hebben interne terugkoppelingslussen, hetgeen ze dynamisch maakt en dus geschikt om tijdsgebonden informatie te verwerken. In de praktijk blijkt dat RNNs trainen voor realistische toepassingen niet altijd triviaal is, en word gehinderd door een aantal problemen. Hierdoor is er een recente trend ontstaan die het trainproces sterk vereenvoudigt. Het RNN word willekeurig opgebouwd, en blijft onveranderd, afgezien van enkele globale parameters. Enkel een uitleeslaag wordt getraind, hetgeen een veel eenvoudiger probleem is.

Toen eenmaal duidelijk werd dat deze strategie opmerkelijk goed werkt werd het duidelijk dat het RNN vervangen kan worden door eender welk random dynamisch systeem, gegeven aan enkele voorwaarden wordt voldaan. De term *Reservoir Computing* (RC) werd geïntroduceerd, waarin het reservoir het dynamische systeem is dat bewerkingen uitvoert op de invoerdata. Het reservoir concept is toegepast of klassiek RNNs (gekend als *Echo State Networks*), gepulste neurale netwerken (*Liquid State Machines*), en een waaier aan fysieke en abstracte implementatieplatformen.

## Deze thesis

Het feit dat een willekeurig dynamisch systeem een krachtige computationele entiteit kan zijn is intrigerend, en vormde de basis van een nieuw veld van theoretisch onderzoek. Mijn thesis draagt bij tot de kennis in dit domein door het raamwerk van RC uit te breiden naar een ruimere set van situaties. In het bijzonder bestudeerde ik reservoirs in continue tijd, en oneindig grote reservoirs.
Mijn werk kan opgesplitst worden in drie hoofdthemas. De eerste heeft betrekking op hoe reservoirs informatie over hun invoergeschiedenis op kunnen slaan in transiënte dynamische toestanden. Het tweede deel weid uit over hoe reservoirs oneindig groot kunnen worden, en toch nog toepasbaar zijn op applicaties. Het laatste deel kijkt voorbij de gewone RC opstelling, en gaat over hoe we de lessen die we hebben getrokken uit RC kunnen toepassen om klassieke leeralgoritmes te verbeteren, en hoe we architecturen van RNNs kunnen ontwerpen om uitdagende problemen aan te pakken.

## Eigenschappen van geheugen

Een reservoir is een dynamisch systeem. Dit betekent dat zijn huidige toestand afhangt van de geschiedenis van het invoersignaal. Deze eigenschap is wenselijk aangezien tijdsgebonden taken zoals spraakherkenning een integratie van tijdsgebonden informatie vereisen. Precies hoe en hoeveel reservoirs van hun recent invoersignaal onthouden is een goed bestudeerd probleem. de aanname die echter altijd werd gemaakt is dat het te onthouden signaal bestaat uit ééndimensionaal ruis in discrete tijd.
Ik heb dit onderzoek veralgemeend voor twee belangrijke situaties: dat waar de invoer meerdere dimensies heeft, en dat waar het invoersignaal en het reservoir in continue tijd bestaan. Ik toon aan dat voor beide gevallen we het oorspronkelijke raamwerk kunnen herdefiniëren. Ik presenteer een diepgravende studie naar de eigenschappen die nodig zijn om reservoirs adequaat geheugen te geven voor beide gevallen. Ik toon aan dat voor continue tijd

systemen we in staat zijn om bekende wiskundige concepten te gebruiken die de dynamica van discrete tijd reservoirs die wenselijke eigenschappen hebben transformeert naar het continue tijdsdomein.
Ik eindig dit onderzoek met de beperkingen van het huidige raamwerk waarin geheugen is gedefiniëerd te bespreken, en ik bespreek meer algemene, niet-lineaire extensies. Ik stel een manier voor waarop het geheugen van een reservoir kan worden gevisualiseerd op elk moment in de tijd, hetgeen een alternatieve interpretatie biedt voor geheugen in reservoirs.

## Oneindige reservoirs

Het kan bewezen worden dat een oneindig groot reservoir computationeel universeel is: het kan alle mogelijke bewerkingen uitvoeren op zijn invoersignaal. In de realiteit kunnen we reservoirs niet oneindig groot maken, maar ik toon aan dat we iets kunnen doen dat bijna net zo goed is: we kunnen een kernfunctie definiëren die op tijdsreeksen inwerkt en die gedefiniëerd is als het inproduct van de geasociëerde interne toestanden. Dit is het equivalent van zulke oneindig grote systemen, en laat toe om oneindige reservoirs werkelijk toe te passen, met het nadeel dat we beperkt worden in de hoeveelheid data waarop zulk een systeem getraind kan worden. De geassocieerde kernfunctie noem ik *recurrente kernfuncties*.
Ik lijd de recurrente kernfuncties af voor een brede set van RNN types, en ik toon aan hoe deze gebruikt kunnen worden om uitspraken te doen over de stabiliteit ven de onderliggende dynamische systemen. Niet alleen kunnen recurrente kernfuncties gebruikt worden om uitdagende problemen op te lossen, ze kunnen ook dienen als een potentieel middel om grote dynamische systemen te bestuderen.

## Leeralgoritmes en architecturen

In het laatste deel beschouw ik de beperkingen van RC, en presenteer ik inleidend onderzoek naar klassieke leeralgoritmes. Eerst toon ik aan dat met een sterk vereenvoudigde versie van klassieke leeralgoritmes we reeds een sterke toename in prestatie kunnen meten. Ook wanneer we enkel de invoerrepresentatie trainen presteert het model beter. Deze strategie heeft voordelen voor eenvoud en stabiliteit, aangezien het netwerk niet plots onstabiel kan worden tijdens de leerfase, hetgeen kan gebeuren als we een leeralgoritme toepassen op het volledige netwerk.
In het volgende deel bespreek ik de algemene problemen die voorkomen bij het trainen van een RNN op een moeilijke taak. Ik ontwerp een een gelaagde architectuur van RNNs die potentieel efficiënter is dan een klassiek RNN. Ik

train dit model op een veeleisend tekstvoorspelling-probleem en toon aan dat gelaagde modellen niet enkel sneller kunnen getraind worden, maar ook beter presteren.

# Summary

## Reservoir Computing

In the last decade, with the advent of sufficient computing power, a new field of study has become more and more prominent: that of Machine Learning (ML). Its goal is to tackle information processing problems which are difficult or not possible to solve using explicit programming, but are often considered easy or even trivial for human beings. Examples are for instance speech recognition, face recognition, complex movements such as bipedal locomotion etc. Instead of an explicit program, the approach of ML is to attempt to *model* the data using a parametrizable mathematical function that can be optimized to solve a task as well as possible.
One of the main classes of ML models uses neural networks (NN). These are a highly conceptualized rendition of how brain cells function, and they rely on a large number of neurons (elementary processing units) that are interconnected. NNs have been around for several decades and have proven to be powerful ML solutions for many tasks.
One variant of NNs is the *Recurrent Neural Network* (RNN). This is an NN that has internal feedback loops, which makes it a dynamical system that can be applied on temporal data. Typically, training RNNs is hampered by several problems, making them not trivial to apply to real-world applications. For this reason, a recent development has greatly simplified the training algorithm by keeping the RNN essentially random, safe for global parameters, and only training a readout layer, which is much easier.
Once it became clear that this strategy can be remarkably successful, people have found that the RNN can be replaced by *any* random dynamical system, as long as it conforms to a limited set of conditions. The term *Reservoir Computing* (RC) was born, in which the reservoir is the random dynamical system that performs computations. The reservoir concept has been applied

to classic RNNs (known as Echo State Networks), spiking neurons (Liquid State Machines), and a broad variety of physical and abstract implementation platforms.

## This Dissertation

The fact that a random dynamical system can be a powerful computational entity is quite intriguing, and it formed the basis of an extensive line of theoretical research. My thesis adds to this body of knowledge by expanding the framework of RC into a broader family of situations. Most prominently, I study reservoirs in continuous time and infinitely large reservoirs.
My work can be split into three main parts. The first one is concerned with studying how reservoirs retain information of their input signal in their transient states. The second part elaborates on how reservoirs can be conceptualized into infinitely large objects, and still be put to use for real-world applications. The final part looks beyond the common reservoir computing setup. It is concerned with how, using the lessons we have drawn from RC, we can improve classic training algorithms for RNNs and design novel architectures to solve difficult tasks.

## Memory Properties

A reservoir is a dynamical system. This means that its current state depends on the history of its input signal. This property is desirable, as solving temporal ML tasks, such as, e.g., speech recognition, requires to integrate information over time. Exactly how much of its input history a reservoir remembers has been studied in the past. In this research people have always assumed one-dimensional noise to be the input signal.
I have extended this research to two important situations: that where the input consists of more than one dimension, and that where the input signal and reservoir dynamics exist in continuous time rather than discrete time. I show that in both cases we can find extensions of the original framework in which memory was defined. I also provide an in-depth study to what the determining factors are for having good memory in both situations. For continuous time reservoirs I show that we can use well established mathematical concepts to transform discrete time network dynamics with known properties to the continuous time domain.
Finally, I end this part with explaining what the limitations are of the current setup, and I discuss more general and non-linear ways in which memory needs to be defined. I propose a way in which we can visualize reservoir memory directly at each time, which provides an alternative way of looking at the memory of reservoirs.

## Infinite Reservoirs

An infinitely large reservoir can be proven to be computationally universal: it can perform any computation on the input time series. In reality we can not make reservoirs infinitely large, but I show that we can do the next best thing: we can define a kernel function, operating on time series, which is the equivalent of such infinitely large systems. It allows to compute the inner product of two infinite-dimensional hidden states, which allows us to actually employ infinite networks. It has the added downside that we are restricted in the quantity of training data we can employ. The associated kernel functions I have called *recurrent kernels*.
I provide kernel functions for a broad set of RNN types, and I show how we can use the kernel functions to study dynamical stability of the underlying dynamical systems. Not only can recurrent kernels be used to solve difficult ML tasks, they can also potentially serve as abstract models for large-scale dynamical systems, both real world and conceptual.

## Training Algorithms and Architectures

Finally, I consider the limitations of RC, and present preliminary work into classical training algorithms. I first show that, using a strong simplification of typical training algorithms of RNNs, we can already strongly boost task performance. Even when we only train the input weights of the reservoir the performance increases drastically. Such a strategy has benefits in the form of simplicity and stability, as there is no danger of sudden chaotic behavior, an event that can occur when employing a training algorithm on the full network.
In the next part, I discuss what the common problems are when training an RNN on difficult ML tasks, and I use these problems to come up with a layered RNN (LRNN) architecture that is potentially more powerful than the classic one. I train this model on a challenging text prediction task, and I show that using the layered approach is both faster to train and gives better performance.

# List of Abbreviations

| | |
|---|---|
| BPC | Bits Per Character |
| BPTT | Backpropagation through time |
| CER | Character Error Rate |
| DS | Dynamical System |
| ESN | Echo State Network |
| FER | Frame Error Rate |
| FFNN | Feedforward Neural Network |
| LMC | Linear Memory Capacity |
| LSM | Liquid State Machine |
| LS-SVM | Least Squares Support Vector Machine |
| LRNN | Layered Recurrent Neural Network |
| MF | Memory Function |
| MFCC | Mel Frequency Cepstral Coefficient |
| MKL | Multiple Kernel Learning |
| ML | Machine Learning |
| MSE | Mean Square Error |
| NMSE | Normalized Mean Square Error |
| NN | Neural Network |
| NRMSE | Normalized Root Mean Square Error |
| PCA | Principle Component Analysis |
| RBF | Radial Basis Function |
| RC | Reservoir Computing |
| RNN | Recurrent Neural Network |
| SFA | Slow Feature Analysis |
| STUN | Sparse Threshold Unit Network |
| SVM | Support Vector Machine |
| WER | Word Error Rate |

# List of Symbols and Notations

| | |
|---|---|
| $\mathbf{a}(t)$ | Hidden state of a neural network at time $t$ |
| $\mathbf{s}(t)$ | Input signal at time $t$ |
| $\mathbf{y}(t)$ | Desired output at time $t$ |
| $\tilde{\mathbf{y}}(t)$ | Model output at time $t$ |
| $\mathbf{x}(t)$ | Feature vector at time $t$ |
| $\mathbf{W}$ | Recurrent weights of an RNN |
| $\mathbf{V}$ | Input weights of an RNN |
| $\mathbf{U}$ | Output weights of an RNN |
| $\mathbf{Z}$ | Inter-layer weights of an LRNN |
| $\mathbf{X}$ | Design matrix: concatenation of feature vectors |
| $\mathbf{Y}$ | Concatenation of desired output vectors |
| $\mathbf{C}$ | Autocovariance matrix of the feature vectors |
| $\mathbf{C_Y}$ | Covariance matrix of the features with the desired output |
| $\mathbf{C_s}$ | Autocovariance matrix of the input signal |
| $\xi_i$ | $i$-th eigenvalue of $\mathbf{C}$ |
| $\psi_i$ | $i$-th eigenvalue of $\mathbf{C_s}$ |
| $N$ | Number of hidden nodes / support vectors |
| $N_{in}$ | Number of input dimensions |
| $N_{out}$ | Number of output dimensions |
| $\rho$ | Spectral radius |
| $\zeta$ | Input scaling factor |
| $\gamma$ | Leak rate |

| | |
|---|---|
| $\tau$ | Delay |
| $m(\tau)$ | Memory function |
| $M$ | Memory capacity |
| $M_q$ | Memory quality |
| $\tau_R$ | Reservoir time scale |
| $k(\mathbf{s}_1, \mathbf{s}_2)$ | Static kernel function |
| $\kappa_{t_1,t_2}(\mathbf{s}_1, \mathbf{s}_2)$ | Recurrent kernel function |
| $\mathbf{K}$ | Gram-matrix |
| $\boldsymbol{\Upsilon}$ | Output weights for a kernel machine |
| $\boldsymbol{\beta}$ | Output bias for a kernel machine |
| $P(\mathbf{v})$ | Probability distribution of $\mathbf{v}$ |
| $K$ | In-degree |
| $W$ | Window size |
| $\ell$ | Lyapunov exponent |
| $\eta$ | Step size for learning |
| $L$ | Number of layers of an LRNN |
| $\|\|\cdot\|\|_F$ | Frobenius norm |
| $\langle \cdot \rangle_x$ | Average over $x$ |
| $[\mathbf{u}; \mathbf{v}]$ | Vertical concatenation of $\mathbf{u}$ and $\mathbf{v}$ |

# Contents

# 1 Introduction

The work performed in my research encompasses a broad variety of topics, all within the domain of Machine Learning (ML), more particularly Recurrent Neural Networks (RNNs), and even more specifically Reservoir Computing (RC). In this chapter I will sketch out the concepts that underlie the field of ML and introduce the techniques that are studied in this dissertation. Next I elaborate on the specific type of task that this thesis is about: processing time series and sequential data. The end of this chapter explains the structure of the rest of the thesis, will provide a list of my main research contributions and a list of publications.
I have deliberately kept this chapter easy to read, and kept most of the math and technical jargon for later chapters where they are more appropriate. After all, the concepts I explain in this chapter are well known, and a quick search can provide all the details a reader would want. The function of the introduction is to set the scene and provide the context in which this thesis was written.

## 1.1 Outsourcing intelligence

Many of us start our day by quickly browsing through our mail, at a glance separating pamphlets and advertising from bills and important documents. Next we step into our cars, drive to work and navigate through busy urban traffic without conscious effort. We open our e-mail clients in the office and filter out an average of 90% of spam to get to relevant information. Some of us write down what our bosses dictate, type a witness account in a court case, or simply scribble an address read to us over the phone. When we search Google for images using only vague search terms, we scan the picture collage it offers and within seconds select the ones we need. Similarly, when we are looking for the TV remote, we only require a few saccades to cover the couch, the table and a few other parts of the room, crunching through an onslaught of visual information. Yet somehow we instantly home in on the target, regardless of its perceived orientation or background.

All of these feats are highly trivial by human standards; a tiny sample from a wide set of skills that we barely think worthy of that name. Yet, despite the rapid rise of computational power in the last few decades, and the undeniably paramount role computing devices are playing in our present day lives, not a single one of the above day-to-day actions can be executed with the same success rate or ease by a machine.

Nevertheless, technology slowly starts to catch up, and revealing examples of this fact are starting to pop up here and there in our daily lives.

Everyone with a modern digital camera will have noticed how it can automatically find and focus on human faces in the picture. Recently I have seen an example of a camera that could be set to snap a pic when the subject smiles.

Many word processors in smartphones and on PCs will, while you are typing, provide suggestions to complete the word of phrase. Especially Google seems to have an eerie capability of predicting your search terms.

Our mailboxes will automatically tag suspected spam messages. Though not foolproof (the one my email client uses seems to work by randomly sampling one in every ten mails), it can give a good indicator for someone coming back from holidays to six hundred unread mails.

Another application domain that is becoming more prominent are the so-called *recommendation* systems. You have rented a number of movies from Netflix, listened and rated a number of songs on Pandora, bought a number of items from Amazon, and you will start noticing that the respective websites start to recommend products to you. Admittedly, such systems do not work very well but the problem at hand is a very difficult one, as the goal is to model your artistic tastes and preferences on a very small number of examples.

In one of the most impressive achievements up to date, the *Google driverless car* is a project that, as its name suggests, aims to have cars that drive around fully autonomously. Using a combination of video cameras, laser distance sensors, radar and position sensors, the cars have been able to cross a total of 1,609 kilometers fully autonomously, and an additional 225,308 kilometers with occasional human intervention. Of the only two accidents to occur involving these vehicles, the first was caused by someone crashing into the rear of the car at a red light, and another one when the car was driven by a human.

All these examples are either expert systems or clever combinations of expert systems. Systems that are designed and optimized to perform very well at a single task. All of them have one thing in common: they are not explicitly programmed. And for good reason. Try to imagine you would program an algorithm that transcribes handwritten text. You would need to give an explicit set of instructions to take an image of handwriting: a large matrix of grey scale pixels, and transform this into a string of characters. As we all would agree, the number of ways a single character can be written is truly vast. It can be shifted, skewed, deformed in all possible ways imaginable, and often simply be ambiguous even for human readers, where we would resolve to use the surrounding context to decide what is the most likely option. The set of instructions to solve this task would have to be so vast, it is no longer humanly possible to write down the necessary computer code[1]. Instead, modern day algorithms usually have an underlying mechanism that can be split in two parts.

On one hand there is the *model*, which could be considered the end product; the actual machine. The model can usually be described as a mathematical function of the data the model receives. This function is characterized by a large set of *parameters* which will determine the nature of the mapping it performs.

The other part is the *training algorithm*. Its role is to find parameters that yield a model which performs satisfactorily on whatever task you wish to solve. Usually, a single model can be trained with several training algorithms, and vice versa, a single training algorithm can be used on different models. What unifies all training algorithms is that they work by considering examples of the data the model needs to process. For our earlier example of handwriting transcription, a training algorithm would typically need a large set of written text, together with the associated character strings.

When the model and training algorithm are suited for the task at hand, they will *generalize* from the training data. This means that the model will also perform well on data which is similar but not identical to the training data. Special care needs to be taken that the model does not simply learn the training examples by heart, a phenomenon known as *overfitting*, which I will elaborate on later.

[1]The first attempts to create artificial intelligence were of this nature. Mainly performed between the sixties and the eighties, this approach never rose above the level of demonstrating success in very small toy tasks.

## 1.2 Machine learning taxonomy

It would be a bad idea to call the above account a definition of the field of Machine Learning. Still, to me, it bears a rather well-framed description of ML while still giving a clear picture of what is going on under the hood of the great majority of things classified as such. In my opinion, there is no such thing as a complete definition of ML. Just like there are no adequate definitions for *life* or *intelligence*. Most people consider ML to be a subdomain of the equally hazy field of Artificial Intelligence (AI). The difference between AI and ML, as I feel it, is that the end goal of AI is to create an intelligent system (or more popularly: ‘agent’), which is capable of displaying intelligent behaviour. Machine learning on the other hand, emerged from the attempt to solve smaller, more well-defined subproblems, which need to be tackled if we ever wish to actually build such an agent. As ML flourished in the last two decades, it became a whole research domain by itself. This fact, combined with a marketable application domain, caused the initial purpose to get pushed to the background, and these days when we think about speech recognition, face detection, medical signal processing etc., the term Artificial Intelligence springs to mind far less easy than Machine Learning. Despite the difficulties of corralling ML, I will present an overview of learning paradigms and tasks that are included into its realm.

### 1.2.1 Learning paradigms

- **Supervised Learning** This is the most straightforward learning setup: there exists a dataset containing input examples that are annotated with their desired output values, and the learning is geared towards finding a function that optimally performs this mapping.

- **Unsupervised Learning** In this case, there is only input data, and the algorithm will try to find an underlying structure that explains it. A typical example of unsupervised learning would be clustering, where one assumes that the data is a superposition of several different datasets, each with their respective distribution. Unsupervised learning can also be used to find a lower dimensional representation of the input data which doesn’t lose relevant information.

- **Semisupervised Learning** A more common scenario is where only a small part of the data is annotated (as this is rather expensive and slow). It is still possible to put the rest of the data to use to improve performance on the mapping from input to output.

- **Reinforcement Learning** Another well studied case is that when there is no direct target for the model to learn, but only an indirect clue as to whether the model performs poorly or not. Examples can typically be found in the world of robotics, where explicit movements needed to solve a task are not known, but it is usually possible to judge the relative performances of different instances of the action. If the robot improves on the action it is given a 'reward', which typically reinforces its behaviors, and vice versa it can be 'punished' for bad behavior.

### 1.2.2 Tasks

It is useful to provide an overview of the types of tasks that are commonly considered, as each has their own unique difficulties and challenges.

- **Regression** When the desired output data of the model is continuous, the learning algorithm will perform *regression*. Typically this is performed by minimizing the squared error between the actual and desired output of the system. An example of this would be to predict the price of a house from a large set of describing features, such as location, lot size, number of rooms, etc.

- **Classification** When the output of the model are labels, e.g. 'apple', 'pear', or 'banana', we speak of classification. A broad set of methods to obtain classifiers is available.

- **Sequence prediction** A whole field of study is devoted to time series prediction. The task consists of predicting the next instance in a sequence given the history of the previous ones. The underlying principle can be both regression (for instance for predicting stock prices), and classification (predicting the next character in a sequence of text). Especially when one considers recurrent prediction, i.e. predicting many steps into the future, typical challenges appear which are unique for this field.

- **Controllers** The final type of task I mention are control tasks. Closely related to reinforcement learning, here the task is defined as: provide an input to a system which then gives a desired output. For instance: rotate the handlebar of a bicycle in such a way that it stays upright.

In this dissertation I will only focus on regression, classification and sequence prediction.

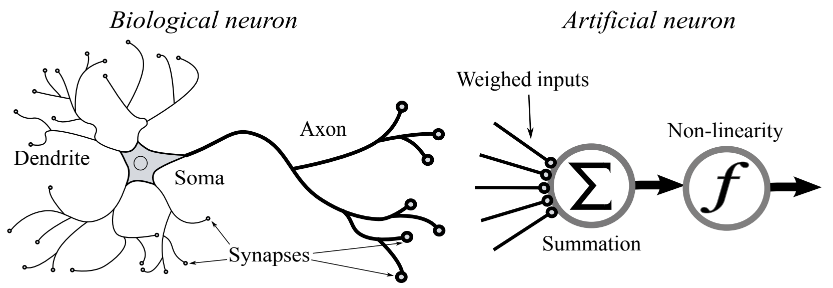

**Figure 1.1:** Schematic renderings of a biological and artificial neuron.

## 1.3 Models

There is a plethora of models in the field of ML, many geared towards rather specific tasks, and an exhaustive overview of all of these would be beyond the scope of this work and my expertise. Therefore I will limit myself to describing only the two techniques which are relevant to the rest of this book: namely neural networks and kernel machines. I will already begin to describe some of the concepts relevant for this thesis, leaving the details for later chapters.

### 1.3.1 Artificial Neural Networks

#### 1.3.1.1 Neurons

A *neuron* can be considered an elementary computational unit. Its operation is based upon the function of biological neurons, which are the elementary processing units in our nervous system. The human brain packs an estimated $10^{11}$ of these in an extremely dense and strongly interconnected network (Herculano-Houzel, 2009). The number of connections between them is a staggering $10^{14}$ (Drachman, 2005). As far as we know, all of our thoughts and feelings, both conscious and unconscious, are patterns of activity within this compact, soft and pink organ. No wonder then, that the biological neuron was the inspiration of one of the largest fields of ML.
In this paragraph I will briefly explain the elementary behavior of biological neurons, and then proceed to explain their artificial counterparts in some detail.

- **Biological neurons** A biological neuron is a somatic cell. It is capable of producing a rapidly rising and falling spike of electric potential over its cell-membrane. These spikes are called *action potentials*, and they are usually very constant in shape and size. It is widely assumed that the information contained in a single spike is only the moment in which it occurs, and not its size or duration. Neurons produce irregular sequences of spikes (called spike trains) and are able to communicate these to the surrounding neurons.
  The general morphology of a brain cell is depicted on the left side of figure 1.1, and can be described as follows. First there are the *dendrites*, large, branching structures of filaments that permeate the surrounding tissue. On these, a large number of *synapses* are placed, through which action potentials from neighboring neurons arrive. When this happens, a synapse will, depending on its type, either stimulate or suppress the activity of its cell. The amount of influence a synapse has (its relative strength, or as we will later call it, *weight*) differs widely and is adaptable. Currently it is believed that what we as humans call learning is nothing more than changes in synaptic strength throughout the brain.
  The dendrite collects the incoming spikes and channels them towards the cell body (the soma). Here, the processing happens. If the cell is stimulated sufficiently, such that its internal membrane potential rises above a certain threshold level, it will produce a spike, which will travel down the axon that connects to other neurons.
  The internal dynamics of biological neurons is rather complicated, and described by the so-called Hodgkin-Huxley model (Hodgkin and Huxley, 1952). In fact, a precise model would need to take into account the exact three-dimensional morphology of the cell, which is the focus of the so-called *Blue Brain project* (Markram, 2006), where extremely realistic biological networks are built and simulated in intricate detail. Although these models are of great interest to brain research and eventually understanding cognitive functions, their computational demands are exceptionally high, and for real-world applications rather impractical. For this reason, the biological neuron model will be simplified a great deal, as I will discuss next

- **Artificial spiking neurons** The first simplifying measure is to still use spikes, but to simplify the underlying dynamics. This lead to a broad range of models, such as the Izhikevich model (Izhikevich, 2004), the Fitzhugh-Nagamo model (FitzHugh, 1955), the *Spike Response Model* (Gerstner, 2001), the *Leaky Integrate and Fire model* (Gerstner and Kistler, 2002), and many others. All of these still consider spike trains to be the basic representation of information. Most of these models still serve to research models of biological neural networks, and even though some applications for artificial spiking neural networks exist, in reality there is no single agreed-upon way in which information can be encoded in spike trains (Masuda and Aihara, 2003; Theunissen and Miller, 1995), hence the next simplification goes one step further.

- **Analog neurons** The simplest way in which information can be encoded into spike trains is to only consider the average spike rate, which is also one of the oldest interpretations (Adrian, 1928). We can define a certain quantity $a$, which is the average spiking frequency of a cell over a sufficiently long time period. We call the quantity $a$ the *activation* of the neuron. Similarly, we can then approximate the total stimulus entering the cell as being the sum of the respective activations of the neurons it receives input from, weighted with the synaptic strengths: given input signals $s_i$ and synaptic weights $w_i$, with $i = \{1 \cdots N\}$, $N$ being the total number of incoming connections, the artificial neuron's state $a$ is then given by

$$a = f\left(\sum_{i=1}^{N} w_i s_i\right).$$

  Here, $f$ will be a response function that represents how fast the neuron will spike for a given amount of stimulus. Biological neurons will start to produce spikes as soon as the stimulus is higher than the previously mentioned threshold value. As the stimulus increases, so will the firing frequency. For physiological reasons, a neuron has a certain fixed period after a spike in which a new spike is impossible, such that there is a maximum firing frequency. This means that for the stimulus going to infinity, the activation function will saturate. I have sketched the biological response function in the upper left panel of Figure 1.2.
  In artificial neural networks, the response function is often chosen to be of a shape roughly similar to the biological function, but with an easy mathematical expression, such as the fermi-function:

$$f(x) = \frac{1}{1 + \exp(-x)}, \tag{1.1}$$

  A broad range of other functions, which divert even further from the biologically realistic model, have been applied, such as the hyperbolic tangent, the linear rectifier function, the linear function, the threshold

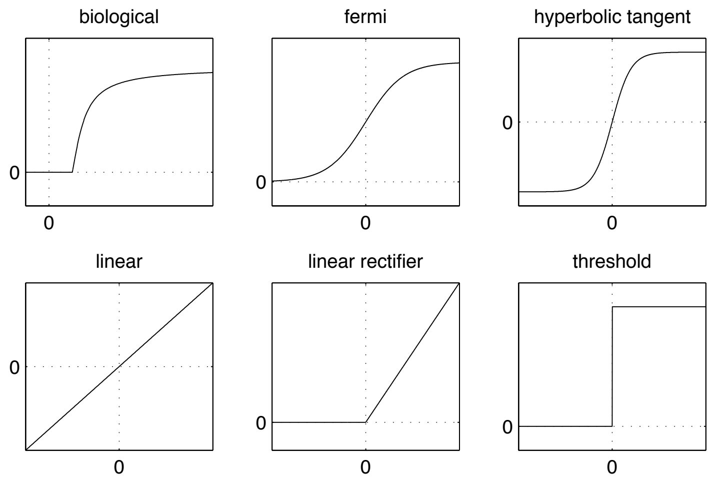

**Figure 1.2:** Examples of commonly used activation functions. The horizontal axis is $x$, the total amount of stimulus a neuron receives, where the vertical axis is $f(x)$, the responding activation of the neuron.

function, etc. I have presented a graphical overview of these so-called *activation functions* in Figure 1.2. Notice that the link with the activations of such neurons and biological neurons is rather strongly abstracted. For one there is no such thing as a negative spiking frequency. All of these activation functions have special purposes, however. Mostly they make theoretical predictions on network behavior easier.

#### 1.3.1.2 Building networks

To put neurons to use they are built into structures known as *neural networks* (NN). A broad variety of network architectures exist, and I will provide an overview of the most common ones here.

- **Single hidden layer** The most basic form a neural network can take is that with one layer of so-called 'hidden neurons', as depicted in Figure 1.3. The input data is projected into a large set of neurons, the combined activations of which form the *hidden state*, which is in turn projected onto one or multiple output nodes (which are usually linear). The network is now characterized by the set of weights from the hidden layer, which we can gather in a matrix $\mathbf{V}$, and the set of weights going

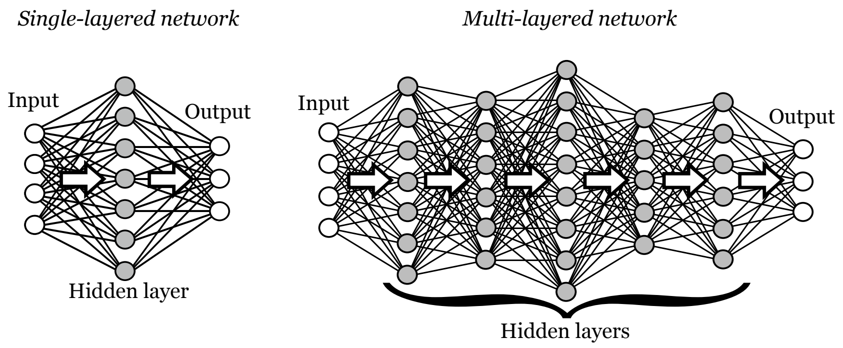

**Figure 1.3:** A single-layered and multi-layered feed-forward neural network. The grey circles are the non-linear nodes and the black lines the connections. The white circles are the (usually linear) in- and output nodes. The white arrows depict the direction in which information flows.

to the output layer **U**, and the output vector **o** can be compactly written as $\mathbf{o} = g(\mathbf{U}f(\mathbf{Vs}))$, where **s** is the input vector and $g$ a (usually linear) output function. The system now needs to be optimized by finding weights **U** and **V** which give the desired output. It can be mathematically proven that any bounded function on the input data can be approximated arbitrarily well as the number of hidden nodes increases (Cybenko, 1989). In this sense a neural network is a *universal approximator*. Important to realize is that the system requires to be non-linear, i.e. the activation function of the hidden nodes should not be simply the linear function, as a linear combination of linear function is always linear itself.

- **Deep architectures** It is also possible to stack layers of hidden neurons on top of each other, as illustrated in Figure 1.3. Such networks can model more complicated functions with more efficiently than single-layered networks (i.e., with a smaller number of neurons), but are usually harder to train. A lot of recent research focuses on ways of pre-training such networks layer by layer in an unsupervised manner, before training the whole architecture for the task they wish to solve. Among others, this research has led to *stacked denoising autoencoders* (Vincent et al., 2008) and *Deep Boltzmann Machines* (Hinton, 2006).
- **Recurrent networks** The network variant I have studied throughout my research is the *recurrent neural network* (RNN). Essentially, when

one considers data that is sequential, such as time series or text, it is desirable to keep a certain level of context into account. For instance, if one wishes to predict the next character in text, it is vital to have information on the preceding sentence and not just the current input character. The RNN is essentially a single-layered network that includes the hidden state of the previous data entry as additional input. Later in this chapter I will go into more detail as to the particular behaviors such networks can exhibit.

#### 1.3.1.3 Training strategies

After defining the network architecture, one needs to try and find weights which work well for the task at hand. Again, a broad set of algorithms exist that tackle this problem, often for particular types of nodes and networks. In the past, people have applied evolutionary algorithms, expectation maximization, simulated annealing, and many others. However, the most common form of optimization in nodes that have a well-defined derivative of the activation function is *gradient descent*. Without going into the details, gradient descent will typically minimize a cost function (for instance the mean square error between the desired and actual output value), and calculate the derivative of the cost function with respect to the parameters. This derivative is then the gradient, i.e. the direction in which the parameters have to change in order to lower the cost. The algorithm will then change the weights a small step in this direction and repeat the process.

Gradient descent is a relatively reliable method for single-layered networks, but can run into trouble when applied to deep networks, and especially RNNs. First, as the learning algorithm essentially takes the path of steepest descent in the cost function landscape, it can get stuck in a local optimum. Secondly, for RNNs, changing the parameters continuously can lead to very sudden and discontinuous changes in the network's behavior, which almost always leads to an increase in the cost from which it may never recover. These so-called *bifurcations* are the bane of people trying to optimize RNNs, their frustration something I've had a few tastes of myself.

A conceptually easier training strategy is to focus only on the weights that lead from the hidden state to the output, and leaving all other parameters untrained. This simple, but surprisingly effective idea has been applied to single-layer networks (called *extreme learning machines* (ELM) (Huang et al., 2006)) and RNNs (called *Reservoir Computing* (RC), see Chapter 2), and solves both mentioned problems associated with gradient descent. As the hidden-to-output mapping is a linear projection, for many cost functions there exists a single, easy-to-find optimum. In the case of mean-squared-error, these optimal values can be found in a single pass via linear regression.

For RNNs, bifurcations only occur when the recurrent weights change, and the dynamics of the network are unaffected by altering the output weights[2]. The downside of RC compared to fully trained RNNs is that the network will be less efficiently tuned to the task, such that in order to have similar performance, the networks will generally have to be larger.
Reservoir Computing is one of the main themes this thesis will deal with.

### 1.3.2 Kernel Machines

The second class of ML algorithms I will discuss here are *Kernel Machines.* Their operational principles are radically different from NNs. It is possible to explain kernel machines from a strictly mathematical point of view, which I will do in Chapter 4. As this is an introduction, I will restrict myself here to a more intuitive explanation.
Suppose you have only a limited amount of data, for instance you have a labelled example set of fifty pictures of apples, and fifty pictures of pears. What you wish to do is to classify new pictures of fruit to any of these two classes. Obviously it is possible to try and train a complicated model that performs such a mapping, however, as the dataset is small, it is also possible to simply take the new picture and try and see how similar it is to the examples. One could for instance assign a score to how similar the new picture is to each of the training pictures, and for each class add the scores. The new picture will then be assigned to whichever class has the highest score in total.
This method can be further refined. Instead of simply taking the total sum, one can weigh each of the scores according to how representative each particular example apple or pear is. Also, instead of comparing with all the data in the set, one can select a smaller subset which is particularly representative for its class. Obviously, the way in which the scores are determined is very important, and may yet require the optimization of some parameters.
Any ML strategy that uses the above working principle can be considered a Kernel Machine. A 'kernel' is nothing more than the function that determines the previously mentioned score, a similarity measure. The kernel function has to fulfill the so-called *Mercer condition*[3] (Mercer, 1909), which a.o. implies that it needs to be symmetrical, i.e., the score of comparing data point A with B has to be equal to the score of comparing B with A. Typically, the choice of kernel function will depend on the sort of data one

[2]With the exception of the case where there is output feedback, i.e., where the current output of the network serves as input to the next hidden state as is common practice in recursive time series prediction.

[3]Suppose $k(\mathbf{x}, \mathbf{y})$ is a kernel function. It fulfills the Mercer condition if there exists a map $\phi(\cdot)$, such that $k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$

tries to process. I will briefly mention some examples of typical kernels.

#### 1.3.2.1 Kernel functions

- **Linear** The simplest similarity measure between two data points $\mathbf{x}$ and $\mathbf{y}$ is their inner product. If we write $k(\mathbf{x}, \mathbf{y})$ as the kernel function, this becomes $k(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$. Linear kernels can work very well for classification problems where the data is linearly separable[4].

- **Polynomial kernels** The linear kernel is obviously unable to solve problems which require a nonlinear model. For this reason the linear kernel can be easily extended to nonlinear versions: $k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^n$ and $k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^n$, respectively known as the homogenous and inhomogenous polynomial kernel of the $n$-th degree.
  All kernels of the form $k(\mathbf{x}, \mathbf{y}) = f(\mathbf{x} \cdot \mathbf{y})$ have an important limitation in their representational power. Indeed, from a geometrical viewpoint it is quickly obvious that such a kernel will assign an equal value to all $\mathbf{y}$ that have the same projection onto $\mathbf{x}$, even though the part of $\mathbf{y}$ orthogonal to $\mathbf{x}$ can take on any size or direction possible. Nevertheless they can work very well for data that is contained within a certain range (such as grayscale values for pixels, which lie between zero and one)

- **Radial Basis Functions** A more intuitive measure is one that is based on the distance between the two data points: $k(\mathbf{x}, \mathbf{y}) = f\left(||\mathbf{x} - \mathbf{y}||\right)$. The most popular kernel of this form is the Gaussian Radial Basis Function (RBF) kernel:

  $$k(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{1}{2\sigma^2}||\mathbf{x} - \mathbf{y}||^2\right),$$

  which immediately introduces the parameter $\sigma$, known as the *kernel width.*

- **Advanced kernels** Using geometrical concepts such as inner products or distances will not work for certain kinds of data. Let us consider strings of text as the data points. Take for instance two sentences:
  ```
  Mary played with the ball.
  Tim played with the ball.
  ```
  which I have placed in monospace font for an easy character-wise comparison. If we do a letter-by-letter comparison of the two sentences, i.e.

[4]This means that there exists a hyperplane which will have all data from the first class on one side, and all data from the second class on the other.

consider each vertical couple of characters as you would do with the coordinates of two data entries, these two almost identical strings of text have very little in common. The fact that the bottom string is shifted one character to the left makes concepts as Euclidean distance meaningless as a similarity metric. Therefore, specialized kernels (called string kernels) have been developed that operate on strings. They are based on comparing all possible sets of substrings. As such the strings `supercomputer` and `computational` will have a non-zero degree of similarity, but `spoonful` and `migrated` are completely dissimilar.
Another, similar metric used to build kernels that operate on structured data is based on the so-called *edit-distance*, which is the minimal number of operations required to change one structure in another. For strings, these operations are insertion, deletion and replacement of a character. A typical example of data processing that benefit from edit-distance is DNA-matching. Edit-distance kernels can also be used for comparing graphs, which have wide application domains, a.o. in bioinformatics (Leslie et al., 2004) and chemoinformatics (Micheli et al., 2007).

#### 1.3.2.2 Practical setups

Many types of models and training algorithms apply kernels. Here I will discuss the three most important types and briefly explain their underlying principles.

- **Support Vector Machines** Originally designed for classification problems, Support Vector Machines (SVM) use the so called *maximal-margin* principle, where the classifier tries to find the hyperplane that separates the two classes with the widest possible margin (Cortes and Vapnik, 1995). Without going into the details I will add here that SVMs have one important advantage: namely that they automatically select a smaller subset of the training data to serve in the final classifier. This subset consists of the *support vectors*, as they 'support' the separating hyperplane. The SVM algorithm has a single global optimum which can be found using quadratic programming techniques.

- **Gaussian Processes** A whole different approach one encounters with Gaussian Processes (GP) (Rasmussen and Williams, 2006), which are primarily used in regression tasks. Here, first of one assumes that a given set of data can be fitted by a broad variety of models, each corresponding to a different set of parameters. A GP then assumes that the set of possible parameters is distributed according to a Gaussian distribution, and it is possible to take the integral over all possible parameter sets. Kernels are then an integral part of the end solution. In its most basic form, GPs lead to the same end solution as linear regression. However, if one assumes you use some sort of feature map $\phi(\mathbf{x})$ instead of the data itself, the end solution leads to expressions of the form $\phi(\mathbf{x}) \cdot \phi(\mathbf{y})$, which is exactly a kernel function. The downside of GPs is that typically they use all the available data in the train set in the end solution.

- **Least-Squares Support Vector Machines** The last kernel machine I explain is the *Least-Square Support Vector Machine* (LS-SVM) (Suykens and Vandewalle, 1999; Suykens et al., 2002). The underlying principle of LS-SVMs is to minimize the mean square error of a linear combination of the kernel functions applied on the data. The end solution it offers is similar to that of GPs. Due to their underlying theoretical principles, LS-SVMs are able to be readily extended for a wide range of applications.

## 1.4 Machine Learning Challenges

Each ML strategy has its pros and cons. For instance, when training an NN with gradient descent, one needs to set a number of metaparameters, such as learning speed, the initial weight distribution, the size of the data chunks the algorithm sees before updating the weights, etc. All of these parameters are important, and they need to be chosen well. Unfortunately there is no real guideline as to which parameters will work and which won’t. The only way to find out is to try out different combinations and let it run, which in itself may take considerable time.
Kernel machines, on the other hand, have fewer parameters to set which may be optimized faster, but as mentioned earlier they incorporate the training data in the end solution. This means that, for very large datasets, with millions of data points (which is certainly not unusual), kernel methods are particularly impractical. If we extend our scope to other strategies, we find that these kinds of setbacks are common, and choosing a model and training algorithm for a particular problem is a matter of experience and luck. No single technique will work well on all tasks[5]. However, two particular issues arise in all ML domains and are worth mentioning.

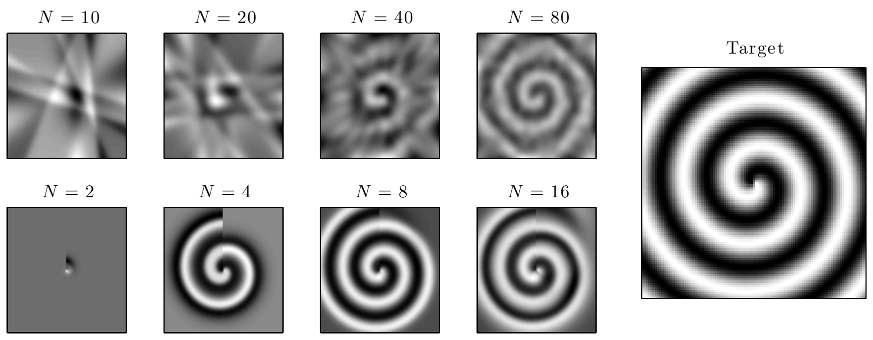

**Figure 1.4:** Illustration of the required representational power of simple single-layer neural networks. The top row are results for the functions learned by networks operating on $x$ and $y$ coordinates, the bottom row is for networks that operate on $r$ and $\theta$. On top of each box the number of neurons of the underlying network is written.

## 1.4.1 Combating complexity

One of the most challenging and interesting applications for ML is Machine Translation. There exist several thousands of languages on the planet, and even though we might think English gets us a long way, we would be unpleasantly surprised if we travel to China, South America, the Middle East, or large parts of Africa. Even South-European countries such as Italy and Spain score particularly low on the English-speaking scale.
It is almost certainly impossible to design that ever so useful science fiction plot device that is a Universal Translator: a machine that picks up the language of a newly discovered alien race and instantly transmogrifies it into American English. Less ambitious and far more feasible would be a system that specializes in translating between two specific languages. Here, at least we can generate unlimited amounts of training data, and can incorporate expert knowledge from people who happen to be fluent in both languages.
Let us take a look at the largest freely available machine interpreter: Google Translate. Consider a Dutch sentence, selected from a Belgian news website: (`www.destandaard.be`) which we are interested in and wish to see translated in English. Google Translate offers us:

> There are more and more pilots in Belgium on the ground with a blinding laser irradiated.

[5]Known within the ML community as the '*No free lunch*'-hypotheses.

After working through the initial confusion this sentence causes, it might for instance signify pilots are being shined upon with a laser when they are not flying (on the ground). Or maybe the pilots hold blinding lasers as they are being irradiated. In fact the article in question deals with pilots that are being blinded by lasers *from* the ground (by pranksters) as they take off or land, and this content, though rather clear in the Dutch version, is completely lost in translation. On top of that the sentence has a highly unnatural feel (though grammatically sound), and no matter how we interpret it, we can easily think of a dozen ways of stating a more plausible equivalent.
Google Translate and similar systems operate at their basic level by applying a combination of both rules and statistics on the given sentence. The model that underlies Google Translate is already tremendously complicated and intricate, and a vast number of exceptions, expressions, strange grammatical quirks specific to a single language, and even slang and swearwords have been seamlessly incorporated within its structure. And still the model is far too simple to capture the full complexity of human language. Is it possible to ever reach human level machine translation? Maybe, maybe not, but it is undoubtedly possible to improve current models.
Machine translation is one example of a task for which we can generate as much data as we want, but simply lack sufficiently powerful models. Other examples that come to mind are weather forecasting, climate modeling, the prediction of the stock-market, and many others. Given more accurate and complex models, we might be able to get better results in all of these. We simply lack the computational power and the ways to deal with such high-dimensional, complicated data.
This phenomenon is known in Machine Learning as *underfitting*. A model needs to have sufficient expressive power to capture the complexity of the data it is processing. Importantly, this is not just a question of giving your model enough parameters (e.g., giving an NN a hidden layer of many thousands of neurons). It is also a matter of choosing the right model and representing your data in the most useful manner. If you assume your data can be expressed as a mixture of Gaussian distributions, while its actual distribution is a combination of croissant-shaped blobs, it would be far better to take a model that assumes just this, and the data would be explained with far fewer parameters.
I have produced an example using single-layered neural networks to illustrate this fact. The task they have to solve is to fit the function $f(\theta, r) = \sin(\theta + 3r)$, an Archimedean spiral where $\theta$ and $r$ are respectively the angle relative to the $x$-axis and the distance to the origin, which I chose for ease of visualization. I try to solve this task by respectively taking Cartesian and polar coordinates as input for the network. Consider Figure 1.4.

Depicted are the resulting functions learned by the networks in both cases. If the network operates on Cartesian coordinates, each neuron will add a sigmoid shape that is drawn out in straight lines across the plane. In fact, the individual contributions of neurons are more or less evident in the top left panel. Fitting a shape like the spiral using such linear shapes is quite unnatural and obviously even for large numbers of nodes (80) the fit to the target function is only mildly convincing. If we work on polar coordinates however, the sigmoids that come out of a linear combination of radius and angle will already be essentially shaped as Archimedean spirals, hence with as few as 8 nodes we already have a nearly perfect fit.
Real life applications often involve high-dimensional data in which a useful representation such as in the example (mapping input to angle and radius) may be incredibly hard to find. There might exist very efficient ways to represent the underlying structure of your data, however, if we lack any additional inside information on the data, the only strategy we have is to try out different models and see how they perform, which is quickly expensive in terms of computing time. In this sense, underfitting is a problem inherent to ML.

### 1.4.2 Overfitting

A completely different situation arises when we have only limited data to work with, yet have no limitations in model power. We as humans are virtually limitless in our capability of storing examples of what we have seen or heard in our lifetimes, but are not always able to generalize beyond what we know, or completely overgeneralize. Not recognizing a penguin as a bird because it doesn't fly would be an example of the first, while claiming that bats and butterflies are birds because they can fly would be the latter. More typically, people learn data by heart, without understanding the underlying principle. This would for instance be the case if you have taught someone to count to forty five, and then isn't able to predict the next number as he never heard it before.
Something similar happens in ML when a model is inherently quite powerful, but is only given a small amount of training data. The cost function on the training data can be brought down almost limitlessly if the model is powerful enough, but the output for unseen data can take on spurious values. In the same vein as the previous paragraph, I will illustrate this fact with an example. I define a one-dimensional mapping $y = f(x)$ (in this case a piecewise cubic Hermite interpolation between 7 randomly drawn couples of $x$ and $y$ values) and draw 20 training examples from this function. Next I train two neural networks on these 20 points, one with 20 nodes, one with

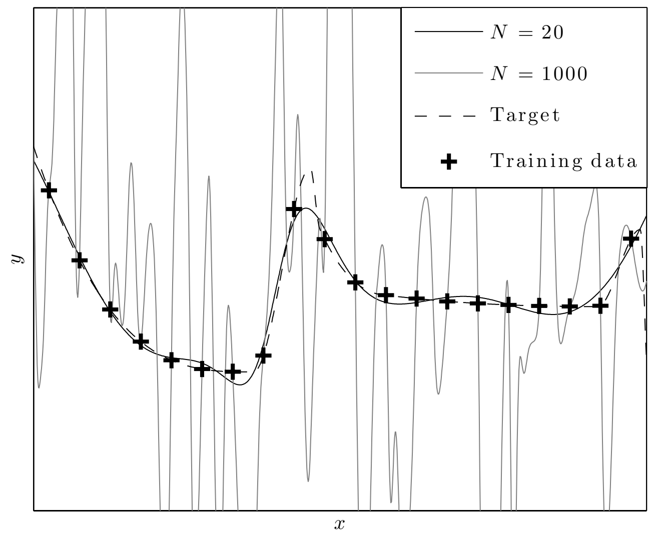

**Figure 1.5:** Illustration of overfitting training data. The number $N$ is the number of nodes in the underlying models. The grey line connects the training data (plus signs) virtually perfectly.

1000. Results are drawn in Figure 1.5. As is apparent, the function learned by the network with 1000 nodes fluctuates wildly in between the training points. Nevertheless, the fit to the training data is almost perfect. As such, when we observe the error on the training data, we will get the impression that the network performs very well. The network with 20 nodes does not connect the training data perfectly, yet the solution it offers is far closer to the underlying target function, with some minor deviations at the sharpest rises and falls.

Overfitting is a very general problem, and in fact can occur with datasets that might seem relatively large. For example: the TIMIT dataset (Garofolo et al., 1993) is a collection of human speech, consisting of 5040 spoken sentences, spoken by 630 different people. The resulting training dataset consists of over 1 million data points, each a 39-dimensional feature vector representing preprocessed sound. The speech is labelled with phonemes, and the goal is to retrieve the sequence of phonemes from the sound.

Although the training set for TIMIT might *seem* large, it is in fact quite easy to overfit; getting training errors that become arbitrarily low, but test

errors that are quite high. The reason for this is that the data is very high-dimensional and diverse. Essentially, the model needs to map the distributions of 61 phonemes, so the number of examples for each of them will already be quite small. Next, this distribution exists in a 39 dimensional space. Getting an idea of what this shape looks like requires a number of examples that rises exponentially with the dimensionality, simply because the volume of this space is so massive. If we want an idea of what a function operating on a one dimensional line looks like, we can take, e.g., ten equally spaced points to get an idea (similar to the training data in Figure 1.5). For a function operating on two dimensions we would need to make a grid of one hundred points, on three dimensions one thousand, and so on. For 39 dimension this number would be $10^{39}$, i.e., astronomical. The reason we are able to extract any information at all from the TIMIT dataset, is that most of the 39 dimensions are unimportant, and the relevant information can be found in a much lower-dimensional subspace.
Various schemes to counter overfitting exist, and I will explain some of these in detail in the later chapters. One of the most common methods is to divide the data into two subsets: a true training set and a validation set. One trains the model on the training set and judges the performance on the validation set, which will give a far more informative error value than the error on the training set. Next, if the model allows it, several techniques exist that can avoid overfitting. For example, with neural networks, one can add an additional term to the cost function that penalizes large weight values, (the smaller these are, the smoother the underlying function will be). Using the validation set it is then possible to optimize the influence of the penalty term, which can be used to train on all the available data.

## 1.5 Sequential data and time series

The type of data I am mostly concerned with in my research is sequential and temporal data. Temporal data is any type of information where the content is spread out in time. We can think of several examples coming from our senses: audio, video and tactile information is highly temporal.
Consider audio: most of the information we extract from the sound that we hear is very strictly temporal in nature. Be it interpreting speech, detecting the swelling rev of an approaching car or a suspicious sound in the middle of the night, gauging the depth of a well by dropping in a pebble and waiting for the plunge, timing is an integral part of all these skills.
The same line of reasoning obviously applies to vision. Cycling through a

crowded street, playing basketball, or reading a book, our brains receive streams of moving images that are processed on the fly into meaningful concepts. Less obvious, but also relevant are our tactile sensations. When we touch an object in the dark to find out what it is, we move across it in order to sense the changes in pressure and feel of texture. Similarly, blind people can read Braille by quickly gliding over the dots with their fingertips. The human brain is so strongly geared towards temporal data that some of our species' most widespread forms of art are fully temporal in nature: music and dance.

A similar type of data is sequential information. Though not temporal in the strict sense, it shares the property that there is a dimension in which the data is ordered, and relevant information is spread out over this dimension. The first example that comes to mind is obviously text. You, as reader, are absorbing this sequence of words and your brain seamlessly transforms this into meaning. Precise timing is not important here, only the order of the sequence. Another example is DNA. The order of base pair triplets in a gene will determine a corresponding order of amino-acids that ultimately decides the structure of a functional protein.

For ease of notation I will introduce some useful concepts. When we assume that a sequence exists in discrete instances arranged in a specific order, we will call the input data at a specific instance a *frame*. The sequences we work with are thus sequences of frames with a fixed dimensionality, which we denote as the *input dimension*. Temporal data normally exists in continuous time, like for instance the world we perceive. However, we will usually assume that we can discretize time in such small partitions that changes between two adjacent frames are negligible, and we can still use the above terms. We will look at actual continuous time systems later in this dissertation.

For what comes next we will use 'time' in the broad sense, as being the direction in which our sequence progresses, even if there is no direct connection to physical time. Similarly I shall no longer discern between 'temporal' and 'sequential' as the difference is always clear from context. The tasks we shall consider in this dissertation are temporal tasks that have a well defined output at each time step. Examples are the aforementioned phoneme recognition task, industrial control tasks where for instance a controller needs to keep the temperature in a tank at a constant value, and for instance predicting the values of stocks one week ahead. Tasks which would consider a full sequence of data and require a single output (such as for instance detecting whether e-mail is spam or not) are of a somewhat different nature and I won't go into these.

### 1.5.1 Variable context

When trying to process sequential or temporal data, one runs into a unique problem. Consider the previous sentence. If we would read all words save the last two, we would be capable of predicting that the next words are quite likely to be 'specific problem', or 'particular issue', or any of a very limited set of similar phrases. We would be quite surprised to find the words 'chestnut tree' next. However, if we change the sentence to read 'In complete darkness, lost in the forest, Billy runs into a ...' this option would become much more acceptable. Obviously the phrase 'to run into a' can have several meanings, according to context. As we read, our brain somehow drags along the relevant context which greatly aids us to interpret and predict. It is capable of doing so almost indefinitely long. Often, sentences will only convey meaningful information within this context, which on its own contains information from a potentially very long stretch of foregoing text.
At the moment we lack the means to drag along such a context within ML. As such, applications that try to extract meaning from texts are extremely limited. One technique quite commonly used to determine the nature of a text (document classification, spam filtering, sentiment analysis[6]) is the so-called *bag of words*-model (Sivic and Zisserman, 2003), which only counts the relative frequencies of words and uses this as a feature vector as input into a model. For more advanced tasks, which operate more locally in a text, this technique is insufficient. We need to take into account some information of the past text, but there is no specific guideline on which information this would need to be, and where we can retrieve it. If we wish to predict the next word in a text, we might need to use context that was given many pages earlier, and often it is highly impractical to operate on the full text, as this can be of considerable length.
Nevertheless, we can make certain assumptions and try and see how far we get. The next two paragraphs will deal with the two main strategies applied on temporal and sequential information.

### 1.5.2 Time window

The first technique to process sequential data is to use time windows. Let's assume all the information that is relevant at this moment in time is contained in a stretch of data with a history no longer than a certain maximal value $W$. This finite sequence, starting from $W$ frames ago up to the present moment we can call a *window* of data. As time progresses, this window slides

[6]For instance to determine whether a movie review is good or bad.

over the sequence, and at each time step we process the chunk of data it sees[7]. The advantage of this strategy is that the size of the data we consider is constant. That way, we can apply any technique we would normally apply to static data, such as multi-layered NNs and kernel machines, without having to worry about what context we need to take into account.
For many tasks the underlying assumption that all relevant information is contained within a short part of the sequence works very well. For speech for instance, when transcribing spoken sounds to labels, the relevant information doesn't need to be longer than about the length of a word. Looking further ahead or back in time may help, but isn't absolutely necessary. Other tasks, like the previous hypothetical example of predicting the next words in a sentence may quickly reach an upper bound in performance that can never reach human level due to the lack of context. And even if the task can be solved with a time window, the window length $W$ still needs to be optimized by hand as it is rarely obvious what the required context length will be.

### 1.5.3 Recurrent techniques

#### 1.5.3.1 The downside of time windows

A time window does not actually take time into account. If we take a window of a sequence and randomly redistribute its frames over time according to a fixed permutation, this will make no difference to the model that processes the data, as we made no specific assumptions to the role of time (other than that all relevant information is within the window). Obviously, sequential data will normally have a strong causal structure, and is often correlated locally in time.
Time windows do not take advantage of this fact. One of the results is that the dimensionality of the input data presented to the model can become very high. Indeed, if we have a time window of length $W$ and a signal of dimensionality $N_{in}$, the number of input dimensions is $N_{in} \times W$. Let us again consider the text prediction task. Working at the level of words quickly becomes problematic: there are hundreds of thousands of words in English, and a lot of texts will include names of places and persons, so let us operate on the level of characters. If we assign an input dimension to each of a set of characters that includes small and capital letters, numbers, spaces, punctuation, apostrophes, brackets and quotation marks, we easily get somewhere around 70 dimensions. Say that we consider a context slightly longer than

[7]The input would be formed by concatenating the individual frames: if we have data $\mathbf{s}(t)$ for $t = 1...W$, we can construct the vector $\mathbf{s_W} = [\mathbf{s}(1); \mathbf{s}(2); \cdots ; \mathbf{s}(N)]$.

an average English sentence: about 100 characters, we already have 7000 input dimensions, for many models an impractically large number. There must exist a more compact representation of the context, that explicitly takes time into account

#### 1.5.3.2 Dynamical systems

The information contained within a sequence can be considered to be causal to some degree. The next frame in a sequence is often quite predictable, and can be said to be at least partly "caused" by the previous one. This is especially apparent in tasks that model the dynamics of complicated entities such as car engines, weather systems, chemical plants, etc, which are governed by physical equations that are inherently causal. To a lesser degree, the same holds for human languages: when we tell a story, we tell it from beginning to end and not vice versa, the next sentence being the logical follow-up from the previous one. It is only natural to assume that a model that operates on such data would benefit from being inherently causal itself.
A set of causal systems which are also parametrizable mathematical entities, and therefore are potential candidates for models in ML, are *dynamical systems* (DS). A DS is defined by an internal state, which evolves according to either a differential equation (in continuous time) or an update equation (in discrete time). A real-life example of a DS is for instance a swinging pendulum which has as its internal state angle and angular velocity. A DS like this can be excited by external input, in the case of the pendulum for example by setting it into motion. Let us formalize the DS by stating it has an internal state $\mathbf{a}(t)$, and receives an external input signal $\mathbf{s}(t)$. The evolution of the DS is then given by

$$\mathbf{a}(t+1) = f(\mathbf{a}(t), \mathbf{s}(t+1)) \quad \text{(Discrete time)} \tag{1.2}$$

$$\frac{d\mathbf{a}(t)}{dt} = f(\mathbf{a}(t), \mathbf{s}(t)) \quad \text{(Continuous time).} \tag{1.3}$$

The function $f$ will define the dynamical behavior of the system and contain all parameters. Remember I mentioned RNNs in the section on neural networks. RNNs count as discrete time DSs, as their current hidden state depends on the current input and the previous hidden state.
The idea of Reservoir Computing, which I have briefly described earlier, generalizes the use of DSs for ML to an unlimited range of possible implementation platforms. Computing can be performed by buckets of water (Fernando and Sojakka, 2003), mass-spring-damper networks, the gene regulation network of bacteria (Jones et al., 2007), photonic chips (Vandoorne et al., 2008), chemical reactions, animal morphology, etc.

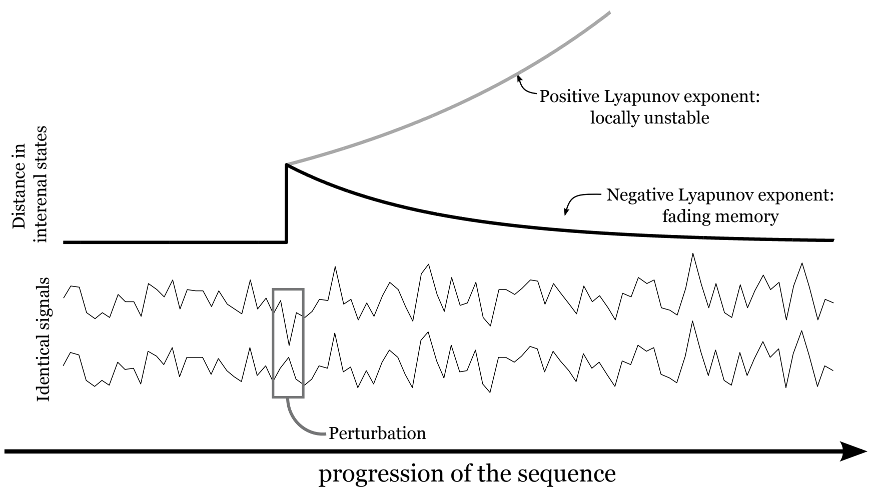
Positive Lyapunov exponent: locally unstable
Negative Lyapunov exponent: fading memory
Distance in interenal states
Identical signals
Perturbation
progression of the sequence

**Figure 1.6:** Illustration of a way to measure the Lyapunov exponent of a DS.

### 1.5.3.3 Fading memory

The internal state of a DS depends on the current input and the previous internal state, which depends on the previous input and the internal state before that, and so forth. This recursion causes the internal state to depend on the entire history of the input up to the current time. For most tasks we expect that the relevant data will be in the recent history of the signal, and we do not want the internal state to depend on the whole sequence. The quality we seek for in the dynamics of our model is called *fading memory* (Boyd and Chua, 1985): we wish that the internal state depends the strongest on the most recent history of the input sequence, and progressively less on older data. There exists a very intuitive mathematical entity that provides a useful frame in which to describe this fading memory property quantitatively, called the Lyapunov exponent (Lyapunov, 1992). Later in this dissertation I will give the formal mathematical description, but here I will only give a illustrative explanation.

In order to check for how long the internal state depends on the former history, we can take two identical copies of a DS, fed with the same input sequence, but at a single point in time we slightly change the input frame of one of the two copies, such that their internal states are no longer identical. If the DS displays fading memory, we expect that the distance between the two internal states will gradually fade as the model "forgets" the influence of the perturbed frame. If it does not fade, but rather stays present or even

starts to grow, we know that the internal state will depend on the whole history of the sequence. I have illustrated this in Figure 1.6.
Classically, such an analysis is made for autonomous systems, i.e., systems that do not receive external input, but rather operate under their own dynamics. Instead of a perturbation in the input signal, a perturbation in the internal state is considered, and the corresponding evolution of the internal state distance. The rate of growth of this distance is known as the Lyapunov exponent. The type of dynamical system I study is driven by external input, however, and I consider this input signal to be a part of the whole system, and define the Lyapunov exponent as such.
Note that, if the disturbance grows with time, this means that the dependence of the internal state on the history increases as it goes further into the past; in some sense the memory of the beginning of the sequence is stronger than that of the current input, the opposite of the fading memory property. This type of dynamics is associated with what is known as *chaos* (though it is not the only necessary condition). We almost never wish for a chaotic system as a model for our data, hence we make some effort to keep it in the fading memory regime. One of the problems with the previously mentioned gradient descent technique for RNNs is that a continuous change in the parameters can lead to an abrupt change from a fading memory to a chaotic regime (the bifurcations, which I mentioned before) which is one of the reasons why gradient descent is quite non-trivial with RNNs.
In reality the picture is much more complicated. If a positive Lyapunov exponent appears, it doesn't inevitably mean the system will descend into chaos. Quite often it will start to exhibit oscillatory behavior (which is not the same as chaos), or settle into another part of its internal state space where the local dynamics are stable. The situation also strongly depends on the input. It is perfectly possible to have a DS that, when it doesn't receive input, behaves chaotically, but when driven with a sufficiently strong input signal still has a fading memory. Conversely, it is possible that the dynamics of the system are only locally stable, and given sufficiently strong input can reach an unstable regime. Think for instance about a ball that rolls around in a crater on top of a volcano. As long as the ball doesn't get kicked hard enough it will always roll back and settle in the center of the crater, but as soon as it flies over the rim it will roll down the mountain, i.e. reach an unstable regime.

## 1.6 This dissertation

Now that I have covered the basics of the themes that will be of importance in my thesis, I shall here provide an overview of the chapters to follow.

- **Reservoir Computing**
  In the next chapter I will elaborate on the Reservoir Computing concept in its many forms. I will explain in mathematical detail the important parameters that have been identified in past research that determine fading memory and non-linearity, and provide an overview of common reservoir implementations.

- **Memory Properties**
  Chapter 3 will provide an in-depth study of fading memory in reservoirs. Specifically, I provide a mathematical analysis for some important cases which can identify the accuracy with which past information is stored in the current internal state. In particular I study two cases. First I will consider the case of multidimensional input, where the role of spatial statistics within the input signal is scrutinized. Next I will consider the case of continuous time dynamical systems and provide ways to analyze their fading memory using mathematical tools that are originally developed for discrete-time systems.

- **Infinite Reservoirs: Recurrent Kernels**
  This chapter will explain how it is possible to unite the concept of Reservoir Computing with Kernel Machines. I will formally derive kernel functions that are associated with infinite neural networks, and next show a generic way it is possible to make these networks recurrent. I will offer an analysis on the properties of recurrent kernels which shows that many of the properties known to be of importance in Reservoir Computing have a direct counterpart in the dynamics of recurrent kernels.
  I will provide a broad overview of both network types that can be made infinite by providing an equivalent kernel, and kernel functions that can be made recurrent. In some cases we can even find network types that are finite approximations of recurrent kernel functions.

- **Training Recurrent Networks**
  Although gradient based strategies for recurrent networks can be difficult, I did some research in this domain and explored two strategies to overcome or reduce the problem of bifurcations. In the first I tried out a highly simplified version of a classic learning algorithm, which

has benefits in terms of speed and stability. The second one considers layered architectures of recurrent networks.

- **Conclusions and Future Perspectives**
  Finally, in the last chapter I will recapitulate the results and conclusions that can be drawn from the work I did in the last four years, and speculate on possible paths for future research, and the current challenges facing Reservoir Computing, and Machine Learning in general.

## 1.7 Research contributions

Here I will list the main research contributions of this dissertation
The first part of my research is focused on finding generalizations of what we consider to be memory in dynamical systems. First of all I explain the concept of linear memory capacity: a measure of how well a DS can remember its past input. Classically, this measure is applied in a rather one-sided setup: the signal that needs to be recovered is defined as one-dimensional, discrete time noise sampled independently each time step. Yet, in realistic applications, the data is neither. Typical examples such as speech or financial time series are usually high-dimensional, and they have strong temporal correlations. Quite often the data is originally of a continuous-time nature and is only discretized in order to allow it to be processed by a computer. I introduce novel definitions for memory capacity for both these important situations: high dimensional input and continuous time signals.
I found that many of the well-known conclusions from the classic research on memory capacity remain valid in the high-dimensional case. First of all I show empirically that the total memory capacity, i.e., the sum of the memory capacities of the principal components of the input signal, never exceeds the number of degrees of freedom of the DS (in the case of recurrent networks this is the number of nodes). Secondly I find that random networks assign a disproportionally large part of their memory capacity to principal components with low power, a scenario that we usually wish to avoid. I show that orthogonal networks, i.e., networks with orthogonal recurrent connection matrices, do not suffer from this problem. This reiterates the fact that orthogonal networks exhibit superior properties in terms of linear memory capacity.
In the case of continuous-time DSs I show that random networks are especially poor at storing recent input history. I propose two ways of constructing networks, which are directly based on the inverse $z$-transform, a way to transform continuous DSs to discrete time. The first is based on discrete-time random networks and is much more robust against noise. The second

one is based on orthogonal networks, and yields networks that have tunable memory. Such networks can be readily analyzed mathematically and have very good noise robustness.

The second line of research this thesis is concerned with, is the introduction and analysis of *recurrent kernels*. I show that it is possible to find kernel functions that operate on time series, and which are associated with infinite sized recurrent networks. Essentially such kernels operate on two time series, and allow to take the inner product of the hidden states these time series would have produced in an infinite RNN.

I offer a broad selection of examples of recurrent kernels. Not only can we take several typical recurrent network models and devise their recurrent kernel equivalent, we can also take existing kernels, typically used for static data, and make them recurrent.

I show that a recurrent kernel can reveal how the underlying network parameters influence dynamical stability. I find a way to relate the concepts of spectral radius and Lyapunov exponent with properties that can be derived from the recurrent kernel functions.

Just like RNNs are in certain cases better suited to model time series, so are recurrent kernels better candidates for ML solutions operating on temporal data. I apply recurrent kernels on a difficult speech recognition problem, and show that they can attain results close to the state-of-the-art.

The final part of my research is concerned with the explicit training of machine learning models using gradient descent. First of all I show that it is possible to significantly boost ESN performance by only training input weights, which has the inherent safety that the network cannot suddenly become chaotic. Finally, I present preliminary results that suggest that there is some advantage of creating deep architectures with RNNs. Such structures may have the benefit of being able to offer a more efficient nonlinear transformation of the input data, and being relatively fast to train. I show that layered networks outperform common RNNs with the same number of trainable parameters.

## 1.8 List of publications

### Journal publications

1. Michiel Hermans and Benjamin Schrauwen. Recurrent kernel machines : computing with infinite echo state networks. *Neural Computation*, Vol. 24(1), pp. 104-133 (2012)

2. Francis wyffels, Michiel Hermans and Benjamin Schrauwen. Building robots as a tool to motivate students into an engineering education. *AT & P Journal Plus*, Vol. 2, pp. 113-116 (2010)

3. Michiel Hermans and Benjamin Schrauwen. Memory in linear recurrent neural networks in continuous time. *Neural Networks*, Vol. 23(3), pp. 341-355 (2010)

## Conference publications

1. Michiel Hermans and Benjamin Schrauwen. Infinite sparse threshold unit networks. *Proceedings of the International Conference on Artificial Neural Networks*, (2012)

2. Michiel Hermans and Benjamin Schrauwen. One step Backpropagation Through Time for learning input mapping in reservoir computing applied to speech recognition. *Proceedings of the IEEE International symposium on Circuits and Systems*, pp. 521-524 (2010)

3. Francis wyffels, Michiel Hermans and Benjamin Schrauwen. Building robots as a tool to motivate students into an engineering education. *Proceedings of the 1st International conference on Robotics in Education*, pp. 49-52 (2010)

4. Michiel Hermans and Benjamin Schrauwen. Memory in reservoirs for high dimensional input. *Proceedings of the International Joint Conference on Neural Networks* pp. 1-7 (2010)

5. Michiel Hermans, Benjamin Schrauwen, Michiel D‘Haene and Dirk Stroobandt. Biologically inspired features in spiking neural networks. *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing*, pp. 328-334 (2008)

# 2 Reservoir Computing

In this chapter, the basic principles of Reservoir Computing (RC) will be explained. Commonly, when people provide this explanation, they start from a historical perspective, i.e. they cover the different angles from which the idea of RC emerged, and then build up to a more formal description. I will violate this tradition by starting the story with one of the most common implementations of RC, namely *Echo State Networks*, as they provide good intuitive means to introduce the relevant concepts in Reservoir Computing. From this, I will work my way up to more general and exotic implementations of reservoirs. At the end of this chapter I will briefly describe some of RC's most notable successes, and conclude by speculating on the reason of their impressive performance. For a more exhaustive overview of the world of Reservoir Computing I refer to Verstraeten et al. (2007) and the excellent review paper of Lukosevicius and Jäger (2009).

## 2.1 Echo State Networks

### 2.1.1 Basic network setup

An *Echo State Network* (ESN) is a recurrent neural network. If we consider a network with $N$ neurons[1], the $i$-th neuron is characterized by an internal state $a_i(t)$, and collectively, we denote the internal state of the network by the column vector $\mathbf{a}(t)$. In these equations, $t$ can be considered as discrete and two-way infinite, i.e. $t \in \mathbb{Z}$, but in practical situations time will be bounded by the time span of the input data. Whether $t$ is associated with time, or more general: progression through a sequence, depends completely

[1]Know that the term 'neuron' can be freely interchanged by 'node' or 'unit'.

on the context of the data.
Next, we consider the input signal $\mathbf{s}(t)$ of dimensionality $N_{in}$. We assume that $\mathbf{s}(t)$ is defined for $t = 1, \cdots, T$, i.e., we only have a finite sequence of data. This will generally be the case for real-life data, such as speech corpora or financial time series. If we use artificial data, we can in principle generate limitless amounts, but if we wish to train the neural network in a finite amount of time we will have to work with a finite sequence of data. We further assume that there exists an input bias, which can be simply considered to be an additional input dimension with a signal that is constant and equal to one. This signal can be used to break the inherent symmetry present in the internal state of an ESN, as I will explain later on.
The evolution of the internal state of the ESN is as follows:

$$\mathbf{a}(t) = \tanh\left(\mathbf{W}\mathbf{a}(t-1) + \mathbf{V}\mathbf{s}(t) + \mathbf{V}_b\right). \tag{2.1}$$

Here, $\mathbf{W}$ is an $N \times N$ matrix, $\mathbf{V}$ is an $N \times N_{in}$ matrix, and $\mathbf{V}_b$ is an $N \times 1$ matrix, i.e. a column vector. These matrices contain the recurrent weights, input weights, and bias weights of the network respectively. Notice that the last can be considered as being an additional input matrix, where the corresponding input is always equal to one, hence the previous explanation of input bias. The non-linearity, a hyperbolic tangent, operates element-wise, i.e. $\tanh(\mathbf{x}) = [\tanh(x_1), \tanh(x_2), \cdots, \tanh(x_N)]$.

### 2.1.2 Choosing the weights

The feature of ESNs that is quite unique is that generally all the weights in the network are chosen randomly from a certain distribution. The only parameters that are optimized are global scaling parameters for each of the weight matrices, as they will determine the dynamical behavior of the internal state. In what follows I will describe some of the heuristics used to predetermine these global scaling factors.

#### 2.1.2.1 Recurrent weights

- **Initializing the weights** We generally assume that the recurrent weights are drawn i.i.d. from a Gaussian distribution. Usually it will not matter much whether they are drawn from another distribution, be it uniform, Laplacian, or even randomly drawn from $\{-1, 1\}$. The so-called *connection fraction* deserves some attention. It determines the average number of neurons each neuron in the ESN connects to, and later in this paragraph I will discuss its influence.

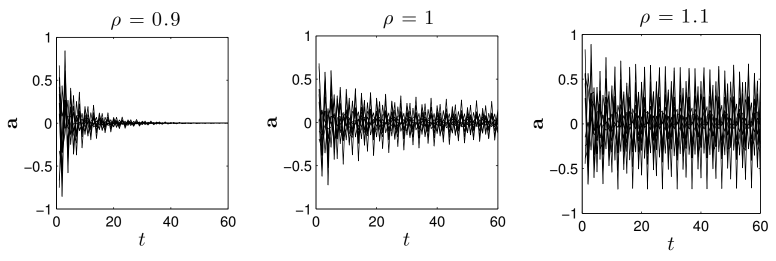

**Figure 2.1:** Three examples of autonomous ESN dynamics for different spectral radii. The network is initialized with hidden states uniformly distributed between $-1$ and $1$ and then left to evolve without external input.

- **Spectral radius: linear approximation** The recurrent weights will primarily determine in which dynamical regime the network will operate. In reality, the network dynamics are rather difficult to study due to the non-linearity incorporated by the hyperbolic tangent. Therefore we can simplify the equations as follows: we assume that the bias weights are zero, and the signal and state are close to zero. This allows us to make a linear approximation of the hyperbolic tangent in equation 2.1:
$$\mathbf{a}(t) \approx \mathbf{W}\mathbf{a}(t-1) + \mathbf{V}\mathbf{s}(t),$$
as $\tanh(x) \approx x$ for $x$ close to zero. If we are only interested in the inherent dynamics of the system we can further simplify this by assuming the input signal is zero, leading to
$$\mathbf{a}(t) \approx \mathbf{W}\mathbf{a}(t-1). \tag{2.2}$$
The above equation has been well-studied in linear dynamical system theory and the evolution of the internal state $\mathbf{a}(t)$ can be solved exactly. What matters for us is that the internal state will only be stable if all the eigenvalues of $\mathbf{W}$ have a norm smaller than or equal to one. If there exists one with a norm larger than one, the magnitude of the internal state will start to grow exponentially.
The eigenvalue with the largest norm is known as the *spectral radius*, which I shall denote as $\rho$. The rule of thumb in ESNs is to rescale the initial recurrent weight matrix such that its spectral radius is fixed to a certain value equal to or slightly less than one. In linear dynamics, $\rho$ will also determine the time scale of the network dynamics in the sense that it puts a lower bound to the rate at which the internal state

will converge to zero at the absence of input:

$$||\mathbf{a}(t+1)|| \leq \rho||\mathbf{a}(t)||.$$

- **Effects of the non-linearity** For truly linear ESNs, the spectral radius would place a strict bound on the regime in which the network will have fading memory. Due to the saturation of the hyperbolic tangent, however, this is not necessarily the case for non-linear networks. First of all, the neuron states are bounded between $-1$ and 1 due to the hyperbolic tangent (see Figure 1.2), such that unbounded growth of $\mathbf{a}$ is impossible. Furthermore, if the network states are not close to zero (as would be the case in normal operation), we no longer can use the approximation for small $\mathbf{a}$, and the linear approximation breaks down. Generally, what happens when $\rho$ is slightly larger than one, is that the hidden state will spontaneously begin to oscillate or move to another stable fixed point. If it becomes significantly larger, the network may become chaotic.
  A thorough analysis of the conditions for fading memory in ESNs has been attempted in several occasions (Ozturk et al., 2006; Verstraeten and Schrauwen, 2009). We can state at least two important findings from these studies:
  - If we still wish to use a linear approximation, we need to do so not around the origin, but around the actual operating point of the internal state. If we take $\mathbf{a}_0$ as the point around which we want to linearise the dynamics, equation 2.2 changes into:

    $$\mathbf{a}(t) \approx \mathbf{DWa}(t-1),$$

    where

    $$\mathbf{D} = \mathrm{diag}(1 - \mathbf{a_0}^2),$$

    i.e. a diagonal matrix with elements equal to the local derivative of the activation function. Generally, as all the diagonal elements of $\mathbf{D}$ are smaller than or equal to one, the spectral radius of the matrix $\mathbf{DW}$ will be smaller than that of $\mathbf{W}$. This means that the more the ESN is driven into the nonlinear regime, the smaller the local derivatives are, and the smaller the effective spectral radius of the system becomes.
  - For realistic problems, the previous analysis is not possible as the internal state does not stay in the vicinity of an operation point $\mathbf{a}_0$, but can have a large variance, making any locally linear approximation useless. Nevertheless we can still calculate the

average of the previously determined local spectral radius, which will give a fairly good indication of the global fading memory of the system (Verstraeten and Schrauwen, 2009).
As a heuristic rule, we can state that it is generally possible to still have fading memory (and hence potentially good task performance) for networks with spectral radius larger than one, if the input weights are sufficiently high, i.e. if the input forces the network into the non-linear regime which will subsequently dampen the activity. Alternatively, one can scale up the input bias of the network such that the states are statically pushed into the saturating part of their non-linearity.

As a rule of thumb, we find that for many tasks the good performance can be found when $\rho$ is close to one[2]. For researchers that wish to gauge ESNs' performance on their task, this is usually a good initial choice.
In Figure 2.1 I have drawn examples of the autonomous dynamics of small reservoirs for different values of $\rho$. When it is smaller than one the states will fade to the origin rapidly. When it is equal to one the hidden state will only be dampened by the hyperbolic tangent, and the fading is very slow. A spectral radius larger than one leads to autonomous activity that never stops.

- **Connection fraction** Finally, concerning the recurrent weight matrix I shall briefly discuss connection fraction. We define the connection fraction $c$ as the fraction of other neurons each neuron connects to. Equivalently: $c$ is the fraction of non-zero elements in $\mathbf{W}$. Connection fraction has been studied for networks with binary nodes, i.e., nodes that have a threshold function as activation function (see Figure 1.2). It has been shown that for such networks, $c$ is a determining factor in the global dynamical stability (Kauffman, 1969; Drossel, 2008): the higher $c$, the less stable the network becomes.
  A threshold function is an extremely non-linear function, however, and if our network still operates in a quasi linear regime, the connection fraction itself has very little influence. Indeed, it has been shown

[2]Here I should include an important remark. There exists a strict mathematical definition of what can be considered a guarantee for fading memory in ESNs, known as the *Echo State Property* (Jaeger, 2001a; Buehner and Young, 2006). From this it follows that a spectral radius smaller than one is not always a sufficient condition for the Echo State Property, and more correctly, it is required that the largest singular value of $\mathbf{W}$ is smaller than one. In reality however, this assumption is far too restrictive, and spectral radius is still a very good indicator for global dynamical behavior

quite often that for most tasks, $c$ has no measurable influence on task performance. In principle, if we would drive a network into an extremely non-linear regime, such that all the states are essentially $-1$ or 1, the activation function starts to resemble a threshold function and the connection fraction will become an important parameter. Realistically however, tasks that require such an extreme non-linearity are rare, and as a rule of thumb, optimizing $c$ is a step that can be omitted. The influence of connection fraction on non-linear networks has been studied thoroughly in Büsing et al. (2010).
Nevertheless, there is an obvious advantage of choosing a low connection fraction: when running an ESN, the computational bottleneck consists of the matrix-vector multiplication $\mathbf{Wa}$. A sparse weight matrix can significantly speed up this calculation and reduce memory requirements of your computer. Sparse matrices have allowed researchers to use ESNs of up to 20,000 nodes (Triefenbach et al., 2010).

#### 2.1.2.2 Input and bias weights

The initial input weights are generally chosen from a Gaussian distribution. Other variants are possible, such as sparse connections, weights chosen from $\{-1, 1\}$, etc. Again, all this matters little. If they are sparse the simulation can be sped up, but sparsity normally has little effect on task performance, and again, the only truly important parameter here is global scaling. The stronger the input weights are, the stronger the network states will be pushed into the saturating parts of the network and the more non-linear its dynamics will be.
Quantifying the degree of non-linearity for an ESN is quite challenging, as it will depend on several factors, including the spectral radius of the network, the spatial and temporal structure of the input signal, its variance, and obviously the input weight scaling. However, to have at least a rough indicator, I here define the *input scaling factor* $\zeta$. Unless stated otherwise, input weights are drawn from a Gaussian distribution with mean zero and standard deviation $\zeta$. Later, in Chapter 3 I shall refine this definition and as the degree of non-linearity I estimate the standard deviation of the ESN.
Bias weights can be separately scaled, but it introduces an additional parameter to be optimized, and often it will change little to final performance. For some tasks, however, bias is essential, in particular if a non-antisymmetric non-linearity is required. Consider a network without bias, driven by input $\mathbf{s}(t)$ and $-\mathbf{s}(t)$. Due to the antisymmetric nature of the hyperbolic tangent, the only difference this makes to the network states is their respective signs. Suppose that somewhere within the underlying function of the task, lies a

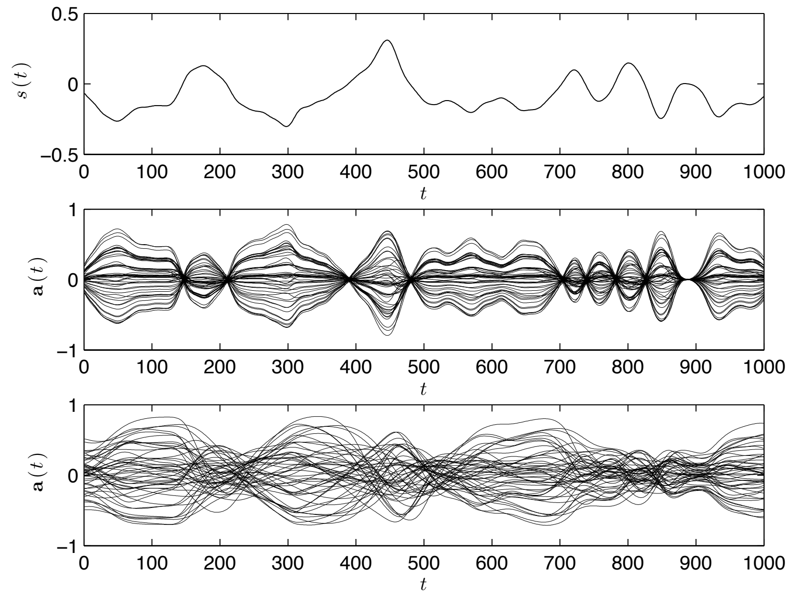

**Figure 2.2:** Illustration of the hidden state for a slowly varying signal. The top panel shows the input signal, the middle one shows the hidden state for an ESN without leaky integration and the bottom one the hidden state of an ESN with leaky integration.

non-antisymmetric function, then we will not be able to solve it without input bias. However, if we use a bias term, we essentially push the network away from the origin, and the activation functions will no longer be antisymmetric.

### 2.1.3 Continuous time problems: leaky integration

In this section I explain an important extension to the previously defined model. Real world time series exist in continuous time. Ideally, we would like to use a continuous-time dynamical system to model it. We are not able to, as computer inherently operate on discrete time, but nevertheless, we can still mathematically define such a system. A continuous time dynamical system is defined by a differential equation. Not wanting to change too much to the discrete time version of an ESN, we write the following definition:

$$\dot{\mathbf{a}}(t) = \frac{1}{\tau_R}\left(-\mathbf{a}(t) + \tanh\left(\mathbf{W}\mathbf{a}(t) + \mathbf{V}\mathbf{s}(t)\right)\right), \tag{2.3}$$

where $\tau_R$ will define the typical time scale of the dynamics of this network. The additional term $-\mathbf{a}(t)$ will ensure that the hidden state remains bounded in the domain $\{-1 \cdots 1\}$. We assume that our time series $\mathbf{s}(t)$ exists in continuous time, but we are able to discretize it by sampling it at a fixed interval $\Delta t$. We can then approximate the differential equation using Euler integration:

$$\mathbf{a}(t + \Delta t) = (1 - \gamma)\,\mathbf{a}(t) + \gamma \tanh\left(\mathbf{W}\mathbf{a}(t) + \mathbf{V}\mathbf{s}(t + \Delta t)\right), \tag{2.4}$$

where $\gamma = \frac{\Delta t}{\tau_R}$, known as the *leak rate*, which should be a number between 0 and 1. The above equation was first introduced in Jaeger et al. (2007) and is now considered one of the most important variations of the classic ESN paradigm. In engineering terms, equation 2.4 describes the situation where each neuron has an internal first order low-pass filter. This concept has hence been extended to band-pass filters (wyffels et al., 2008b)
Adding leak rate to an ESN essentially slows down the dynamics of the network with a factor roughly equal to $\gamma^{-1}$. Notice that for $\gamma = 1$ we end up with the original ESN. If $\gamma = 0.1$, the new internal state will differ only little from the old one. In the extreme case that $\gamma = 0$, the network is infinitely slow, and the internal state does not change at all.
As it slows down the dynamics, leak rate extends the fading memory of the ESN. In Figure 2.2 I show how a network responds to a slowly changing signal, with and without leaky integration. Without, the hidden state basically follows the input signal and shows little inherent dynamics. The memory of the network is shorter than the relevant changes in the input signal, and as such the network will operate in a quasi-static regime; as if the input signal is constant. The network with leaky integration is able to respond to a slow input signal with transient dynamics, such that the hidden state is diverse, and not only a function of the immediate input. As such, leak rate is an ideal means to match the time scale of the ESN to that of the input signal. In chapter 3 I will perform an extensive analysis of the memory properties of continuous time systems, and I will provide insights in what the precise link is between the time scale of the input signal versus the leak rate of the network.

## 2.2 Output weights

The only weights that are explicitly trained in the ESN framework, are the output weights. Here I explain the strategies of obtaining them for two types of problem: regression and classification.

### 2.2.1 Collecting the data

After having chosen a set of parameters and generated all the weights, we now have to run the ESN. We take all the data we wish to train on, and simulate the reservoir according to equation 2.1. We collect the internal states of the network and assemble them in the so-called *design-matrix*, which we denote with $\mathbf{X}$. The size of the design matrix is $N \times T_{\text{end}}$, with $T_{\text{end}}$ the length of your training data. The most basic form the design matrix can take is that where its elements are the concatenated internal states: $\mathbf{X}_{it} = a_i(t)$.
The next commonly performed step is to expand the design matrix with a constant bias equal to one. This is necessary to compensate for any constant offset your desired output may have. If we denote the $t$-th column of the design matrix as $\mathbf{X}_{:,t}$, we can write

$$\mathbf{X}_{:,t} = \begin{bmatrix} \mathbf{a}(t) \\ 1 \end{bmatrix}.$$

Often, the the design matrix is further extended by concatenating the current input signal with the column entries:

$$\mathbf{X}_{:,t} = \begin{bmatrix} \mathbf{s}(t) \\ \mathbf{a}(t) \\ 1 \end{bmatrix}.$$

In principle you can extend it further with whatever additional feature you desire: e.g. the squares of the internal states, the derivatives of the input, the third power of the sine of the products of the input signal with the internal states... anything that you suspect might carry supplemental information can be added to the design matrix. One should be conservative though, as extending it comes with rapidly increasing computational costs, and often does not lead to a noticeable increase in performance.
The vertical entries in the design matrix are the prototypes of data that are fed into the final output weight matrix. If, after training you run your network on new data and you have used the last given definition of $\mathbf{X}$ to obtain your output weights, then, each time step you must form a vector

consisting of the current input frame, the current internal state, and a one in the end. If you use supplementary features to construct your design matrix, these should also be included in the vector that is multiplied with the output weights.
Throughout the rest of this section, I will no longer be concerned with the precise definition of the design matrix. The space spanned by the columns of $\mathbf{X}$ we will refer to as *feature space*, and correspondingly the columns themselves are *feature vectors*, which we shall denote with $\mathbf{X}_{:,t} = \mathbf{x}(t)$. The other matrix we need is the design matrix of the desired outputs. Assuming that each time step we have a desired output $\mathbf{y}(t)$, we construct the matrix $\mathbf{Y}$ such that $\mathbf{Y}_{it} = y_i(t)$.

### 2.2.2 Linear regression

As mentioned before, we now need to find output weights $\mathbf{U}$ that optimally project the design matrix onto the desired output, i.e. if $\tilde{\mathbf{Y}} = \mathbf{U}^\mathsf{T}\mathbf{X}$, or $\tilde{\mathbf{y}}(t) = \mathbf{U}^\mathsf{T}\mathbf{x}(t)$. We wish that $\tilde{\mathbf{Y}}$ resembles the desired output $\mathbf{Y}$ as closely as possible. One way to do this is to minimize the *Mean Square Error* (MSE):

$$\text{MSE} = \frac{1}{T}\sum_{t=1}^{T} ||\tilde{\mathbf{y}}(t) - \mathbf{y}(t)||^2,$$

or in terms of the design matrices:

$$\text{MSE} = \frac{1}{T}||\mathbf{U}^\mathsf{T}\mathbf{X} - \mathbf{Y}||_F,$$

where $||.||_F$ is the Frobenius norm, i.e. the sum of the squares of all elements in the matrix. Minimizing this equation is performed by calculating its derivative w.r.t $\mathbf{U}$, and equating this to zero. The resulting equation for the output weights is then obtained via linear regression, i.e., we need to solve

$$(\mathbf{X}\mathbf{X}^\mathsf{T})\mathbf{U} = (\mathbf{X}\mathbf{Y}^\mathsf{T}) \tag{2.5}$$

for $\mathbf{U}$, which can be performed quickly and efficiently with any linear algebra solver (e.g. in Matlab© this would be the backslash operator).
Most literature on ESNs prescribes that in order to find output weights one needs to calculate the inverse of the matrix $(\mathbf{X}\mathbf{X}^\mathsf{T})$. The output weights are given by the following entity: $\mathbf{U} = (\mathbf{X}\mathbf{X}^\mathsf{T})^{-1}(\mathbf{X}\mathbf{Y}^\mathsf{T})$. Though this is indeed the algebraic solution to equation 2.5 and is quite useful for theoretical research, it does in fact not give the best solution in numerical terms or performance. When inverting a matrix $\mathbf{A}$, a computer will typically try to find a matrix $\mathbf{B}$ such that $\mathbf{AB}$ is as close to the unity matrix as possible.

This step is computationally demanding, as it requires the optimization of a set of $N$ systems of linear equations, $N \times N$ being the size of $\mathbf{A}$. Furthermore, it does *not* try to optimally fulfill the equality of equation 2.5, which a linear equation solver actually will do, and this in a fraction of the time required to invert a matrix. Let it be known henceforth that we never need to calculate explicit inverse matrices for calculating output weights of ESNs!
I will define the estimated covariance matrices $\mathbf{C} = \frac{1}{T}(\mathbf{X}\mathbf{X})^{\mathsf{T}}$ and $\mathbf{C}_{\mathbf{Y}} = \frac{1}{T}(\mathbf{X}\mathbf{Y})^{\mathsf{T}}$, such that equation 2.5 becomes $\mathbf{C}\mathbf{U} = \mathbf{C}_{\mathbf{Y}}$. The division by $T$ is to ensure that the linear system does not depend on the size of the dataset, which is useful for the interpretation of the ridge regression parameter I introduce in the following section.

### 2.2.3 Avoiding overfitting: ridge regression and cross validation

Often the amount of available training data is small, and solving equation 2.5 will offer weights that score very well on the training data, but will poorly generalize to new data, i.e., overfitting occurs.
How can we fight this effect? One solution which is fairly intuitive, is to make multiple copies of the design matrix $\mathbf{X}$, and to each we add a little bit of numerical noise. We can then treat this as a single, large dataset and make output weights accordingly. Adding noise forces the solver to be tolerant to small variations in the internal state, and will not lead to solutions that fluctuate wildly in between actual training points, which in turn leads to better generalization.
We do not actually need to add noise to the design matrix explicitly, rather we can calculate what happens if we average out over all possible noise instantiations. Suppose we define a noise matrix $\mathbf{N}$ of the same size as the design matrix $\mathbf{X}$, and where each element is drawn i.i.d from a distribution with variance $\lambda$ and zero mean. If we construct the covariance matrix with noise $\mathbf{C}_{\mathbf{N}}$ we get

$$\begin{aligned}\mathbf{C}_{\mathbf{N}} &= \frac{1}{T}(\mathbf{X}+\mathbf{N})(\mathbf{X}+\mathbf{N})^{\mathsf{T}} \\ &= \frac{1}{T}\mathbf{X}\mathbf{X}^{\mathsf{T}} + \frac{1}{T}\mathbf{N}\mathbf{X}^{\mathsf{T}} + \frac{1}{T}\mathbf{X}\mathbf{N}^{\mathsf{T}} + \frac{1}{T}\mathbf{N}\mathbf{N}^{\mathsf{T}}\end{aligned}$$

If we assume truly i.i.d noise, we can easily average out this equation over $\mathbf{N}$. We find that $\left\langle \mathbf{N}\mathbf{X}^{\mathsf{T}} \right\rangle = \left\langle \mathbf{X}\mathbf{N}^{\mathsf{T}} \right\rangle = \mathbf{0}$, due to the zero mean, and $\frac{1}{T}\left\langle \mathbf{N}\mathbf{N}^{\mathsf{T}} \right\rangle = \lambda\mathbf{I}$, with $\mathbf{I}$ the unity matrix. For $\mathbf{C}_{\mathbf{Y}}$ we have to consider the average covariance matrix $\left\langle \mathbf{N}\mathbf{Y}^{\mathsf{T}} \right\rangle$ which is equal to zero so that $\mathbf{C}_{\mathbf{Y}}$ doesn't change. This means that we can rewrite equation 2.5 as:

$$(\mathbf{C} + \lambda\mathbf{I})\mathbf{U} = \mathbf{C}_{\mathbf{Y}}. \tag{2.6}$$

This strategy is known as *ridge regression* (or more correctly Thikonov regularization (Tikhonov and Arsenin, 1977)), and is the time-honored method to train reservoirs. Interestingly, it is possible to derive the exact same condition from an entirely different approach: adding a penalty term in the cost function meant to keep the output weights small. Explicitly we write for the cost function:

$$\text{cost} = ||\mathbf{U}^{\mathsf{T}}\mathbf{X} - \mathbf{Y}||_F + \lambda||\mathbf{U}||_F.$$

Yet another approach arrives at this solution from the perspective of putting a prior assumption on the probability distribution the weights **U** (Rasmussen and Williams, 2006). Explicitly, if we assume that **U** is drawn from a normal distribution with variance $\lambda$, we obtain equation 2.6 as the set of weights with the maximum a posteriori likelihood, i.e., the set of weights that is most likely to explain the data given the weight prior.
Ridge regression is one of multiple possible forms of *regularization.* Roughly stated, in machine learning, regularization is any method to keep the complexity of your model within certain bounds. Keeping the output weights small is one of the ways to do this, but a variety of other criteria exist, leading to other forms of regularization. Before ridge regression was widely used, the only easy method available was to keep the ESN small, which required you to search for the number of neurons from where the model would start overfitting. Thanks to ridge regression we can now simply apply the rule that 'bigger is better'.
One variant of the above equation deserves some mention. Essentially, the Frobenius norm is the squared Euclidean 2-norm: the sum of squares. The 1-norm, also known as the Manhattan metric, is the sum of the absolute values of the elements of a matrix. Minimizing this cost (as, e.g., implemented in the LASSO algorithm (Tibshirani, 1994))has the interesting side effect that many of the weights will tend to go to zero. In the end this will lead to a sparse solution, which can be in a sense quite informative, especially in the case where you add many additional features to the design matrix, since it will give you an idea of which features are of importance, and which are not. The 1-norm is also related to the loss function used in SVMs for regression (Smola and Vapnik, 1997) While this cost function is interesting in terms of end product computational cost and perhaps better performance, it also suffers from an important drawback: the 1-norm is not continuously differentiable, and minimizing the cost becomes computationally demanding.

Obviously, ridge regression leads us to a new problem: which value should we choose for $\lambda$. If it is too small the model might still overfit, if we choose it too large we are basically trying to extract our desired outputs from random noise, which will not lead to usable readout weights. The solution is to use cross-validation: split up the training dataset into $L$ chunks, then pick out one of the chunks to serve as a validation set, train on the remaining $L - 1$ chunks, and compute the cost function with respect to $\lambda$ on the validation set. Repeat the process until all chunks of data have been used and then pick the value for $\lambda$ associated with the lowest average cost and use it to train on the complete data set.
This is the method of choice for finding $\lambda$. If you train on very large datasets, or are able to generate as much data as you want, it might be faster and almost as good to simply choose (or generate) a single training and validation set beforehand. When dealing with a large dataset, one should keep in mind though that the size of the training dataset should be more or less similar to the total set, and at the same time the validation set needs to be large enough to be representative for the whole set.

### 2.2.4 Classification

In the previous two sections I described the optimal strategy for solving regression problems. Minimizing MSE is indeed quite intuitive if you wish for two sets of numerical values to be as close to each other as possible. A wholly different problem is the one where you want to classify data. You do not wish to map an output to a continuous value but rather to a discrete variable which indicates a certain class. A good example of such a task is phoneme recognition, which is an integral part of speech recognition, where each frame in the speech signal is classified as belonging to one of a discrete set of phonemes (or silence).
Formally, we wish to find weights $\mathbf{U}$ such that for all data points $\mathbf{x}$ belonging to the first class $\mathbf{U}^\mathsf{T}\mathbf{x} > 0$, and $\mathbf{U}^\mathsf{T}\mathbf{x} < 0$ if $\mathbf{x}$ belongs to the second class. In this case, $\mathbf{U}$ is a vector that defines a hyperplane within feature space that acts as the separator between the two classes.

#### 2.2.4.1 MSE is poor for classification

If we consider the simple case of two classes, how do we go about to train a readout layer that outputs a class instead of a variable? Several solutions to this problem exist. Let us first consider a solution that simply applies the previously explained linear regression scheme. As target values for our output we simply take 1 for time frames belonging to the first class, and $-1$ for those belonging to the second. We can then perform classification by

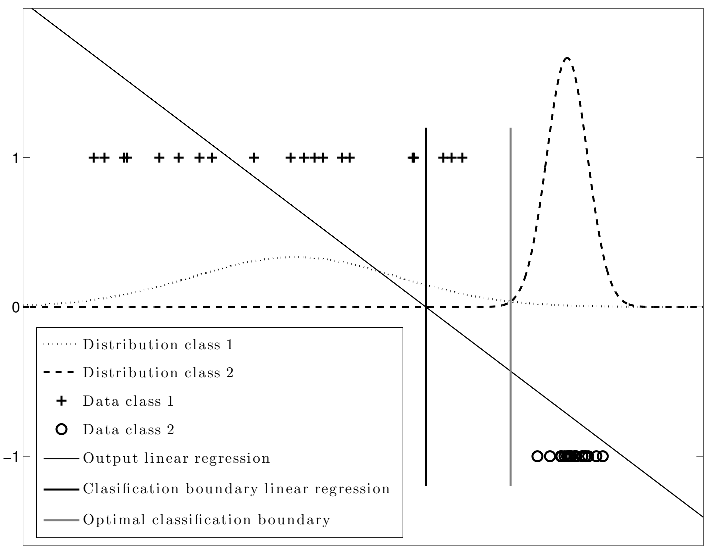

**Figure 2.3:** Illustration of classification by linear regression. Two classes of one-dimensional data are shown, together with their respective distributions. I also show the data fit of linear regression and its corresponding classification boundary, and the optimal classification boundary.

simply looking at the sign of the output: if it is positive it belongs to the first class, and if it is negative to the second. This is in fact the most commonly used scheme when training ESNs for classification problems. It has the advantage that it often works quite well, and it is conceptually simple.

There is, however, a considerable downside to this method. Let us explore a very simple example. For the moment we no longer consider temporal data, but simply one-dimensional data points that belong to one of two classes. Each class has an underlying probability distribution with a respective mean and variance, as I have depicted in Figure 2.3. For each class we have 20 data points in this example. As it happens, the variance of the first distribution is larger than the other. Figure 2.3 shows us the solution that linear regression offers us. The border for the resulting classifier is shown as a black vertical line. We instantly notice that this is not a very good solution. It is too far in the direction of the left distribution, and some points are misclassified. Yet, it is easy to draw a boundary that separates the data perfectly (as, e.g.,

the grey vertical line).
The reason why this happens, is that linear regression tries to map to the *target* values, and is not truly concerned with the classification boundary. In the example, it tries to place the line such that it passes as well as possible through the data points. If a point from the first dataset would lie very far to the left in the picture, it would be classified correctly, but its MSE would be very large. Herein lies the problem with using MSE for classification problems. We are not interested in MSE, we are interested in the number of points that is classified correctly: the classification error.
The same issue arises when the number of data points in the two classes is very different. The classification boundary will inevitably be pushed away from the class with the most data, leading to poor classification on the whole (in the worst case it will learn to always classify as being from the class with the most data).

#### 2.2.4.2 Solutions

A number of solutions for this predicament exist, and I will discuss them here. First of all, there is a range of training strategies that are specifically geared towards classification problems, and these can be readily applied for training output weights. For completeness I will mention them here before moving on to Support Vector machines, which are of more importance for this thesis.
First of all, the problem of unbalanced datasets can be countered by reweighing the misclassification error for each class. This is known as *Fisher relabeling* (Duda et al., 2001), and is a very straightforward extension of simple ridge regression. Next, *Linear Discriminant Analysis* (LDA) (McLachlan and Wiley, 1992) works by assuming that data from both classes has a Gaussian probability distribution with the same variance but a different mean. It works by drawing the line that optimally separates these two distributions (the discriminator). This solution is also highly similar to ridge regression. Other techniques that start from a more probabilistic approach are *Naive Bayes* (Domingos and Pazzani, 1997; Hand and Yu, 2001) and *Logistic regression* (Hosmer and Lemeshow, 2000). Naive Bayes essentially maps a probability distribution on each class and builds a classifier by simply looking at the probabilities for a new data point belonging to each class. The advantage of this technique is that there are no prior assumptions on the probability distributions. The 'naive' part of naive Bayes is that it is assumed that each dimension of the data points is statistically independent of the others. Logistic regression tackles the classification problem by using output nodes with a fermi-nonlinearity (equation 1.1), also known as the logistic function. When the labels of the classifier are then 0 and 1 (instead

of $-1$ and 1), the output can be interpreted as a probability. The downside here is that single-shot learning is no longer possible and we will need a gradient approach (which can be solved very efficiently with second order methods (Fletcher, 1987)).
The classification technique most important to us is that of Support Vector Machines (SVMs) (Boser et al., 1992; Vapnik, 1995). We start out from a specific cost function. Consider the output value of the $i$-th feature vector $\tilde{y}_i$. The target value $y_i$ is either 1 or $-1$. We define the following cost function. If $\tilde{y}_i y_i > 1$, the data point in question is definitely classified correctly, and we give it a cost of zero. If $\tilde{y}_i y_i < 1$, the cost is given by cost $= 1 - \tilde{y}_i y_i$.
This cost function is known as the *hinge loss* and it is particularly well suited for classification. What will happen is that the cost of most data points will be brought to zero, and only a small number of points that lie close to the separator will still count in the cost function. These data points are known as the so-called *support vectors*, in the sense that they 'support' the separating hyperplane. SVMs will define this hyperplane such that it separates the support vectors of each class with the widest possible margin.
SVMs have several important advantages over other classification techniques. Primarily, they are not influenced by the precise shape of the distributions of the classes: rather they only look at the data in the region where the two classes are closest to each other, i.e. the data that is the easiest to misclassify. The optimization problem is convex, such that it is not possible to get stuck in a local optimum, and the global optimum can be found efficiently using quadratic programming.
In Chapter 4, we will use a loss function similar to the hinge loss, but with a quadratic instead of a linear part, which allows us to do support vector selection using linear regression.

#### 2.2.4.3 Multiple classes

So far I have only considered the case with two classes. Some interesting tasks have multiple output labels (e.g. predicting the next letter of a text). A number of strategies exist to solve this problem, and I will provide a very short list here.

- The most straightforward technique is to train a separate classifier for each class which treats the data of the other classes as one single class. This technique is known as the *one-versus-all* strategy
- Another technique, which is a little more involved, trains a large set of classifiers that each are trained into discriminating only two of the classes of the whole set. If there are $C$ classes, the number of classifiers that need to be trained is $C(C-1)/2$. Finally, classification is usually

performed by a voting mechanism over all one-versus-one classifiers. Though more convoluted, this strategy seems to work better than one-vs.-all strategies for some problems (Hsu and Lin, 2002).

- Special mention I will give to the so-called *softmax* output function. Suppose there are $C$ classes and each output node has weights $\mathbf{U}_i$ associated with them. For feature vector $\mathbf{x}$ the output of the $i$-th node is then given by

$$y_i = \frac{\exp(\mathbf{U}_i \cdot \mathbf{x})}{\sum_{j=1}^{C} \exp(\mathbf{U}_j \cdot \mathbf{x})}. \tag{2.7}$$

  This function provides a probability for the datapoint to belong to each class, as it is normalized and nonnegative. The softmax function is the multi-class version of the logistic function. Training it can be performed quite efficiently using gradient ascend on the log likelihood of the output. In Chapter 5 I will apply this strategy on a text prediction task.

## 2.3 Reservoir Computing

### 2.3.1 Generalizing beyond ESNs

When considering 2.1 and 2.2 we can notice a clear distinction. We attain the output weights by applying some well known, preferably simple, algorithm on the feature set provided by the ESN. Nowhere do we rely on the connection matrix of the network, on the fact that the nodes have a hyperbolic tangent nonlinearity, or any other specific quality of the network. This opens the door to an interesting possibility: if ESNs are essentially random dynamical systems that provide useful features, why couldn't we just use *any* dynamical system for computing, provided that its global dynamical properties are within the right regime.

Indeed, here we finally arrive at the paradigm that provides the title of this chapter: Reservoir Computing. A *reservoir* in this context is a high-dimensional non-linear dynamical system of a random nature. The name derives from the fact that it acts as a 'reservoir' of non-linear dynamics that provides an interesting set of features of the history of an input signal, which can - under the right circumstances - be highly useful for solving ML tasks. What exactly are the properties that would define a 'good' reservoir? This question is the subject of some debate still (Legenstein and Maass, 2005), but there is strong agreement on the following points:

- The reservoir is essentially random, safe for some global parameters. We do in no way train the detailed parameter set that describes the system. Normally, the larger the reservoir becomes, the less the performance on a task will depend on the precise instantiation of the system, and the more it will depend only on the global parameters.
- The reservoir should have fading memory during operation, i.e., when driven with the input it is meant to process. Ideally there is some global, tunable parameter that serves to set the 'forgetting speed', conform the spectral radius in quasi linear ESNs, or more generally conform the Lyapunov exponent as explained in the introduction.
- The reservoir's internal state should be high dimensional (in ESNs this is the number of nodes), as a rule it should have significantly more dimensions than the input signal. Furthermore, the internal states should be uncorrelated, in the sense that each dimension of the feature vector carries some information that is not present in the rest of the internal state. Formally, we wish that the covariance matrix of the internal state is of full rank.
- The internal dynamics should be non-linear to some degree. If this is not the case, the reservoir essentially only acts as a linear filter on the input data, and we would be able to compute a FIR-filter that fully replaces the reservoir and output weights. Ideally, the 'amount' of non-linearity of the reservoir is tunable, conform the input scaling factor in ESNs.

### 2.3.2 Originators of RC

The idea of RC arose more or less concurrently from various different fields of study. Here I will list what could be called the architects of Reservoir Computing: the researchers that independently launched the idea of applying large, random dynamical system to perform computations on time series. Next I will list some interesting and more exotic variations of RC.

#### 2.3.2.1 Liquid State Machines

The concept of RC in the context of neuroscience was first coined in Maass et al. (2002), under the name *Liquid State Machines* (LSM), wherein the *liquid* is a large set of randomly connected spiking neurons. In this paper, the idea is coined that the microcircuits in the neocortex of the brain (small groups of particularly densely connected neurons (Mountcastle, 1997)) remain essentially random and all learning happens in the outgoing connections of these circuits. The circuits themselves then only serve as spatio-temporal filter banks from which other neurons can select useful features.

#### 2.3.2.2 Backpropagation Decorrelation

A wholly different approach is found in the framework of *Backpropagation-Decorrelation* (Steil, 2004). Starting from a combination of a special error gradient method (Atiya and Parlos, 2000) and information theoretical viewpoints, Steil arrives at a learning algorithm that is remarkably similar to the normal RC setup, especially in that it uses a large, random and unchanging recurrent network at its core.

#### 2.3.2.3 Cognitive function modeling

Years before all other approaches, cognitive scientists used large random networks to model large scale structures in the brain to model cognitive functions, using reinforcement learning rather than linear regression to find output weights. Their findings have been published in Dominey (1995)

### 2.3.3 Exotic forms

Often cited as the real-life example of Reservoir Computing, in Fernando and Sojakka (2003), the concept of 'liquid' computing has been taken quite literally, as the researchers have used a basin of liquid water to serve as a reservoir. The researchers used a transparent reservoir filled with water, and eight prodding devices to excite the surface. The ripples on the surface were then projected onto a screen, recorded with a webcam, and the resulting video stream was used as feature vector in the RC network. The researchers showed that they were able to solve the temporal XOR task and a basic speech recognition task.
Other physical implementation platforms that have been studied in past research are based on photonic chips (Vandoorne et al., 2008), on optoelectronics (Paquot et al., 2012).
Expanding into the realm of biology, several researchers have found links between computation in organisms and the RC concept. It has been suggested that the bacterium Escherichia Coli uses an LSM to react to changing environmental conditions (Jones et al., 2007). In the field of robotics, the concept of *morphological computation* (Pfeifer et al., 2007), essentially the study on how the anatomy of an animal can perform useful computations, has been linked to RC. Among others people have studied abstract models of biological anatomy (mass-spring-damper systems), as reservoir manifestations (Caluwaerts and Schrauwen, 2011; Hauser et al., 2012).

Surely the range of potential reservoir systems is still much broader, and finding good potential candidates is an exciting field of study.

## 2.4 Achievements

As the scope of this work is largely theoretical, I will not linger long on how well RC performs compared to other techniques. Nevertheless I reserve this short section for an non-exhaustive list of noticeable achievements for RC.

- The very often quoted seminal paper on reservoir computing is Jaeger and Haas (2004), which deals with time series prediction and signal correction. In this paper, reservoir computing beats the state-of-the-art performance on chaotic system prediction, the Mackey-Glass attractor (Mackey and Glass, 1977) and the Lorenz attractor (Lorenz, 1963) by two orders of magnitude in precision. Other works on time series prediction include wyffels et al. (2008a) and wyffels and Schrauwen (2010).
- One of the most prominent engineering applications of machine learning is speech recognition, which has seen some notable successes (Skowronski and Harris, 2007; Verstraeten et al., 2006). The first serious attempts to build large-scale speech recognition systems based on RC have been performed by the Speech Lab in Ghent university, with performance that rivals state of the art. Their work has been published in Triefenbach et al. (2010) and Jalalvand et al. (2011).
- Important work in the field of autonomous robots has also benefited from the reservoir computing approach. A strategy for robot localization has been worked out in Antonelo and Schrauwen (2012), which combines the concept of RC with *Slow Feature Analysis* (SFA) (Wiskott and Sejnowski, 2002).
- In medical signal processing, RC rivals state of the art in epileptic seizure detection (Buteneers et al., 2011, 2012).

## 2.5 Why they work

Finally, we reach the most important question of this chapter: what makes reservoirs actually work? In other words, how comes that a random dynamical system is able to perform useful computations of an input signal? Let us first imagine what exactly a single neuron in a reservoir computes:

a function of the recent history of the input signal. Each neuron activation will depend on the history of the input signal. Due to fading memory, the dependency of the activation on the input signal history will drop off as it is more distant in the past. The exact shape of this function is essentially random, which reflects the random nature of the reservoir. However, there are still global characteristics that are tunable, and which depend on the reservoir parameters: most importantly the rate at which the fading memory drops off, and the degree of non-linearity.

Essentially, reservoir computing thus combines a large number of non-linear filters operating on the input signal to perform useful computations of the input signal. It can be proven mathematically that, if the number of non-linear filters tends to infinity, and they are sufficiently diverse (such that the covariance matrix has full rank) , a linear combination of these filters can approximate any other filter arbitrarily well (Maass et al., 2002, 2007; Schäfer and Zimmerman, 2006). From this point of view, reservoirs are nothing more than random non-linear filter banks. Optimizing the meta-parameters only ensures that the underlying filter that is defined by the task can be efficiently approximated, i.e., with a small number of neurons. This fact is demonstrated in a rather interesting manner when studying very large reservoirs. For small reservoirs, meta-parameters need to be optimized rather precisely to get good performance on a task, but this dependency becomes weaker for larger neuron numbers.

What is interesting about the relative success of RC within machine learning is the fact that apparently, random dynamical systems *are* good models for time series processing. Many sequence processing tasks seem to benefit from the assumption that the underlying function that needs to be approximated has some the dynamical properties of a reservoir. This fact undoubtedly reflects the causal nature of time series as opposed to static data.

In my research I have come to adopt a somewhat comparable view. From the perspective that is introduced in Chapter 4, we can interpret reservoirs as being approximations of infinite-sized DSs. A reservoir is then nothing more than a finite sample from this infinite-dimensional construct. I show that these infinite DSs can be applied explicitly on tasks via the kernel-trick. This allows a researcher to truly work with an infinite-dimensional DS as a model for his or her data.

Considering reservoirs as finite samples from an infinite DS firmly establishes the notion that the nature of a reservoir forms a prior assumption on the dynamical process that generated the data. A reservoir is essentially random, save for the nature of the distributions of parameters, which are the scaling parameters, and activation function and potentially assumptions on connectivity. This limited set of parameters will form the prior. If we then define a reservoir with an infinite number of nodes using this prior, we will incorporate all possible reservoirs that can be drawn from this distribution. In this sense, we can consider an infinite reservoir as a distribution over finite reservoirs.

The infinite reservoir will *de facto* contain the process that generated the data of the task in some subspace of its hidden state (if this process actually exists). How well the prior matches the nature of the task will then be reflected by how efficiently we can approximate this subspace by randomly sampling reservoirs from it.

# 3 Memory Properties

Dynamical systems absorb information of their recent input history. As time progresses, old information dissipates and gets overwritten. The degree to which information of past input is present in the current state of the DS is determined by its memory. In this chapter, I will explicitly analyse memory for Reservoir Computing.
Memory in reservoirs has been the subject of research for some time, and originally the focus of this research was on linear memory capacity, which describes how much information of the input signal can be linearly extracted from the current hidden state. In my work I investigated two important cases, first for multiple input dimensions, and finally, how we can describe and study memory in continuous time systems.
More recently, the focus has shifted from linear memory capacity to non-linear extensions. After all, in practical applications we do not wish to linearly retrieve the input signal, instead, we wish to produce a rich set of non-linear features of the input history and we wish to understand how much of the history of the input is included in this feature set. In the last section of this chapter I will discuss broader definitions of what memory in reservoirs entails. I briefly present the work from Dambre et al. (2012) and review some potential paths for further research, including a method to present an instant visualization of reservoir memory.

## 3.1 Linear memory capacity

### 3.1.1 Definitions

One way to quantify memory is to assign a score to how well past input can linearly be recovered from the current hidden state of the network. In Jaeger

(2001b), a method that measures this score is proposed. Suppose $\mathbf{U}_\tau$ are readout weights that are trained to optimally reproduce a one-dimensional input signal with delay $\tau$: $s(t-\tau)$. If we denote the recovered signal as $\tilde{s}_\tau(t) = \mathbf{U}_\tau^\mathsf{T}\mathbf{a}(t)$, we can define the *memory function* (MF) as

$$m(\tau) = \frac{\mathrm{cov}(s(t-\tau), \tilde{s}_\tau(t))^2}{\mathrm{var}(s(t))\mathrm{var}(\tilde{s}_\tau(t))}. \tag{3.1}$$

The MF is a number between zero and one, and is the squared correlation coefficient between the reproduced and the actual delayed input signal. Perfect reproduction corresponds to $m(\tau) = 1$, and complete inability of reproduction corresponds to $m(\tau) = 0$.
We wish to consider the memory function not so much as a machine learning task, but rather as something that is inherent to the reservoir we consider. Therefore, we shall not make use of a train and test set, or apply techniques such as ridge regression to obtain the output weights $\mathbf{U}_\tau$. Rather, we assume that we have access to an infinite amount of data, such that the statistics in equation 3.1 are exact. Furthermore, we can find the algebraic solution of $\mathbf{U}_\tau$ as

$$\mathbf{U}_\tau = \left\langle \mathbf{a}(t)\mathbf{a}^\mathsf{T}(t) \right\rangle_t^{-1} \left\langle \mathbf{a}(t)s(t-\tau) \right\rangle_t, \tag{3.2}$$

where $\langle \cdot \rangle_t$ denotes mean over time.
Note that I reach back to equation 2.5, but here we use the algebraic solution, and as feature vectors we only use the hidden state. We also omit the bias term for the output. This is justified if the input signal has zero mean, such that we don't need to compensate for an offset. Using the above definition, we can transform equation 3.1 by inserting $\tilde{s}(t-\tau) = \mathbf{U}_\tau^\mathsf{T}\mathbf{a}(t)$, and we find:

$$m(\tau) = \frac{\left\langle \mathbf{a}^\mathsf{T}(t)s(t-\tau) \right\rangle_t \left\langle \mathbf{a}(t)\mathbf{a}^\mathsf{T}(t) \right\rangle_t^{-1} \left\langle \mathbf{a}(t)s(t-\tau) \right\rangle_t}{\mathrm{var}(s(t))}, \tag{3.3}$$

which leaves a function that solely depends on the statistics of the hidden state and the input signal.
The above definition of the MF depends heavily on the temporal distribution of $\mathbf{s}(t)$. Suppose for instance that the input signal is periodic with period $T$. In this case, the output weights for all $\tau + nT$, $n \in \mathbb{N}$ will be identical, and the MF will as a consequence, also be periodical. This would give the false impression that our network has an infinitely long memory.
In order to get an honest view of the memory of a reservoir, we want our input to be as hard to remember as possible, i.e., as random as possible. For this reason, we shall assume that the frames of $s(t)$ will be i.i.d. from a normal distribution.

I now define the *linear memory capacity*[1] (LMC), which we denote by $M$, as

$$M = \sum_{\tau=0}^{\infty} m(\tau). \tag{3.4}$$

It can be said to quantify the total 'amount' of memory present in a reservoir.

### 3.1.2 Facts and examples

Linear memory capacity and the memory function have been studied extensively in past research (Jaeger, 2001a; White et al., 2004; Ganguli et al., 2008). Here I will give a short list of important facts and findings.

- The single most important property of LMC in ESNs is that it is at most equal to the number of nodes, and generally smaller. This can be proven mathematically (Jaeger, 2001a), and intuitively it reflects the fact that, if you have $N$ numbers to encode something in, the maximum amount of data that can be exactly encoded in this is $N$ other numbers. If you wish to encode more data, you will necessarily lose precision.

- Non-linearity deteriorates the LMC. A soon as a non-linear transformation is applied to the input history, a linear projection will not be able to perform a perfectly reconstructing mapping. Therefore, the more non-linear the reservoir, the lower the LMC becomes.

- Generally, we are not only interested in LMC, but also in how well it will hold up against noisy conditions. Imagine that the readout layer reads the hidden state plus a certain amount of observation noise, how badly will this degrade the LMC? This question has been studied in White et al. (2004), and they reached the conclusion that optimal noise robustness is obtained when using random orthogonal weight matrices[2] for the recurrent weights, scaled with a spectral radius slightly less than one. For linear dynamical systems, this means that all the eigenmodes of the system have the same exponential decay rate but different oscillatory frequencies.
  Noise robustness – importantly – also addresses numerical noise, caused by the finite precision of computer processors. Even with double precision numbers, and working in the linear regime, for random networks

[1]Often simply known as memory capacity. Here I wish to distinguish it from the more general non-linear memory capacity I will describe later on.

[2]An orthogonal matrix $\mathbf{A}$ has the property that $\mathbf{A}\mathbf{A}^{\mathsf{T}} = \mathbf{A}^{\mathsf{T}}\mathbf{A} = mathbfI$

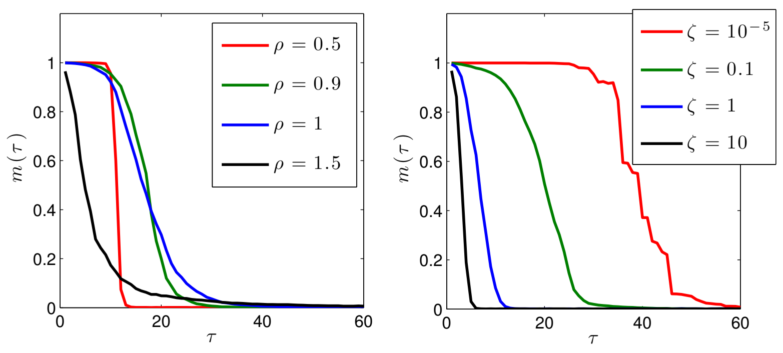

**Figure 3.1:** Examples of MFs for different ESNs with 50 nodes. The left panel depicts the influence of spectral radius on a mildly non-linear ESN ($\zeta = .1$). The right panel depicts the influence of non-linearity for ESNs with spectral radius $\rho = 0.95$. All experiments used 50,000 input frames, drawn from a normal distribution.

(non-orthogonal) it is very difficult to reach $M \approx N$, whereas for orthogonal networks this is relatively easy. The reason as to why will become apparent in the next subsection.

- For the LMC to come close to its theoretical maximum requires a spectral radius close to one. If it is significantly smaller, the memory will decay too quickly and reconstructing input with a long delay becomes numerically infeasible. For linear orthogonal networks, the spectral radius very strongly determines the shape of the MF, in particular, as it approaches 1, it will become lower and reach out further into the past.

In Figure 3.1, I show examples of MFs for 50-node ESNs under different conditions. As is quite clear, both non-linearity and spectral radius heavily influence memory depth of the ESNs. Interestingly, even non-stable ESNs, (the line shown for $\rho = 1.5$) seem to be able to retain some information on past input. Notice that, in the right panel, the line corresponding to $\zeta = 10^{-5}$ corresponds to a very linear ESN, as the elements of the hidden state will be very small and hence remain in the linear part of the hyperbolic tangent.

### 3.1.3 Memory encoded in eigenmodes

It is a quite interesting observation that orthogonal networks have superior noise robustness (and in practical setups, simply a higher LMC). Let us simplify the problem, and for now consider only the linear dynamics of the ESNs. A linear dynamical system given by

$$\mathbf{a}(t+1) = \mathbf{W}\mathbf{a}(t) + \mathbf{V}s(t+1)$$

can be solved analytically. I will not go into the details here, but it suffices to say that the network's hidden state consists of a linear combination of filters, convolved with the input signal. Each of these filters will be characterized by one of the eigenvalues of $\mathbf{W}$. The shape of the filter impulse response is an exponential function, multiplied with a sine wave. The growth rate of the exponential is determined by the modulus of the corresponding eigenvalue, and the frequency of the sine wave by its complex phase. Each of these filters drags along some content of the input signal's history. From this perspective, creating output weights that optimally reconstruct $s(t-\tau)$ is nothing else but an attempt to construct a linear combination of these filters, which is zero for all delays and equal to one for $\tau$.

For a typical randomly created network, what does this set of filters look like? In the top row of Figure 3.2, I have provided an example. On the left I have shown the shapes of all the filters present in the network. On the right we can see the eigenspectrum of the connection matrix. What is immediately apparent is the fact that only a few filter responses actually extend far into the past. When looking at the eigenspectrum, it becomes clear that, even though the *largest* absolute value of the eigenvalues is set to the desired spectral radius, most eigenvalues are more or less uniformly spread over the unit circle, and as such most of them have a much higher rate of decay (as they are closer to the centre).

Still, if we wish to reconstruct $s(t-\tau)$, we will need to combine the impulse responses of the existing filters to approximate a Kronecker delta-peak at $\tau$, and in order to do this, we need very high numerical precision, simply because the output value of most filters will change extremely little for input at large $\tau$. In reality, this kind of numerical precision is infeasible, and as a consequence we will measure an LMC much lower than the predicted maximum.

The bottom row of Figure 3.2 also depicts an eigenspectrum and its corresponding set of filter impulse responses, but here the connection matrix $\mathbf{W}$ is a random orthogonal matrix[3]. For orthogonal matrices, all eigenvalues

[3]There are several ways in which to generate random orthogonal matrices. One would be to generate a symmetrical matrix from a random matrix $\mathbf{A}$ by $\mathbf{S} = \mathbf{A}\mathbf{A}^\mathsf{T}$,

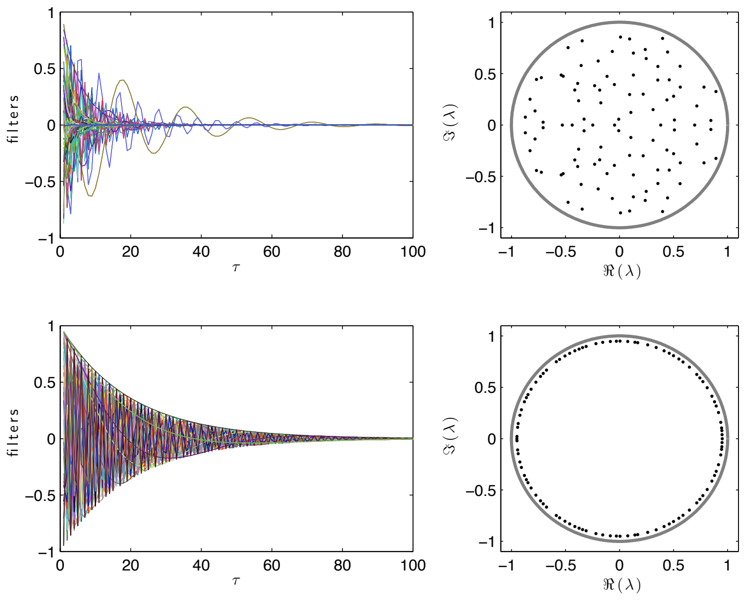

**Figure 3.2:** Examples of linear filters and eigenspectra of linear ESNs. Shown for ESNs with $N = 100$ and $\rho = 0.95$. The top row is for a randomly generated network, the bottom row is for a random orthogonal network. The left panels show the shape of the filters, the right panels the eigenspectra. The grey circles in the right panels are the unit circle.

have the same modulus, hence, all associated filters have the same decay rate. This means that all filters have an equally broad view over the past, and reconstructing $s(t - \tau)$ becomes significantly easier in numerical terms. As we will see in the next section, orthogonal networks also have interesting properties for storing high-dimensional data. Furthermore, it is possible to find a continuous-time equivalent for orthogonal networks that will provide excellent noise-robustness in continuous-time linear dynamical systems.

and next use any computer algebra pack to do an eigenvalue decomposition $\mathbf{S} = \mathbf{ODO}^\mathsf{T}$. The eigenvectors $\mathbf{O}$ now form an orthogonal matrix. In matlab there exists an easier way. It has a built-in function that transforms any matrix to the nearest orthogonal matrix. This function is called orth.

## 3.2 High-dimensional input

True ML applications often have high-dimensional input. In this section I will extend the notion of LMC and the MF to multiple input signals. The results of this research have been presented in Hermans and Schrauwen (2010b). First I will describe the kind of signal I use, next I will write down new definitions for the MF and memory capacity. I then investigate the role of non-linearity, spectral radius, and the number of input dimensions and the statistical structure of the input.

### 3.2.1 Input signal and memory function for multiple signals

I will consider input signals $\mathbf{s}(t)$ of $N_{in}$ dimensions. For simplicity we shall always assume that there are no temporal correlations. Furthermore, we shall denote the covariance matrix of the input as $\mathbf{C_s}$, i.e.:

$$\langle \mathbf{s}(t)\mathbf{s}(t-t')\rangle_t = \delta_{t,t'}\mathbf{C_s}.$$

Obviously, if cross-correlations between the different input dimensions exist, there would be little point defining their individual memory capacities. Indeed, if two input channels are highly correlated, we would violate our previous convention that the input signal should be as hard as possible to remember; if you can reconstruct one input signal you can partially reconstruct the other.
The same is true for temporal correlations. Suppose two input signals are uncorrelated, but one is identical to the other, shifted a few frames in time. In this case most of the information of one signal is already present in the other, and if we would measure the memory capacity of both signals, the total memory capacity would be deceptively high.
For this reason I will make sure that there exist no temporal correlations in the input signal, and I will only consider the principal components[4] of $\mathbf{s}(t)$. In order to do this we perform an eigendecomposition of the signal covariance matrix:

$$\mathbf{C_s} = \mathbf{Q}\mathbf{\Psi}\mathbf{Q}^\mathsf{T}.$$

Here, $\mathbf{\Psi}$ is a diagonal matrix with, entries $\psi_i$ on the diagonal corresponding to the variances of the principal components of $\mathbf{s}(t)$, which we will also denote as being the *power* of the principal component. The principal com-

[4]Principal Component Analysis (PCA) (Pearson, 1901; Jolliffe, 2002)

ponents themselves can be found by $\hat{\mathbf{s}}(t) = \mathbf{Q}^\mathsf{T}\mathbf{s}(t)$. Each element of $\hat{\mathbf{s}}(t)$ is uncorrelated with the others, and now it makes sense to define a memory function and memory capacity for each of them. Suppose $\hat{s}_i(t)$ is the $i$-th principal component of the input signal, we define its corresponding MF as

$$m_i(\tau) = \frac{\mathbf{g}_i^\mathsf{T}(\tau)\mathbf{C}^{-1}\mathbf{g}_i(\tau)}{\psi_i}, \tag{3.5}$$

where $\mathbf{g}_i(\tau) = \langle \mathbf{a}(t)\hat{s}_i(t-\tau)\rangle_t$ and $\mathbf{C} = \left\langle \mathbf{a}(t)\mathbf{a}^\mathsf{T}(t)\right\rangle_t$. This definition is completely analogous to equation 3.3. In the same vein we define the LMC of the $i$-th component as

$$M_i = \sum_{\tau=0}^{\infty} m_i(\tau), \tag{3.6}$$

and the total LMC is simply

$$M = \sum_{i=1}^{N_{in}} M_i. \tag{3.7}$$

Measuring the LMC has one small but important difficulty. Due to the finite number of samples and the fact that the MF is strictly positive, there will exist a positive bias in the measurement. This bias will generally be small but it is nevertheless important when one measures $M$ for multiple input signals, since this error is multiplied with the number of input channels. In appendix A.1.1 we derive that per input channel, this bias is approximately equal to $\frac{N}{T}$, with $T$ the number of samples. We shall subtract this bias when calculating the memory capacities of the individual channels and the total MC.
The second problem appears when trying to investigate the influence of non-linearity. Ideally, we would wish to keep the degree of non-linearity constant as we play with parameters like spectral radius and the number of input dimensions. For this reason I will scale the input such that for all experiments that I perform, the variance of the hidden states is approximately equal for equal input scaling, and hence how far the states are pushed into the non-linear part of the hyperbolic tangent. Exactly how I go about this is described in appendix A.1.2. The result is an input scaling factor $\phi$, which is a crude estimate of the standard deviation of the reservoir states. This means that $\phi = 1$ is very non-linear, and $\phi \approx 0$ is a close-to-linear regime.
In this section, I always use $10^5$ input frames for measuring the MF. Results shown are always averaged over 10 reservoir instantiations, but generally, individual differences are quite small. For all our experiments I use ESNs of 100 nodes.

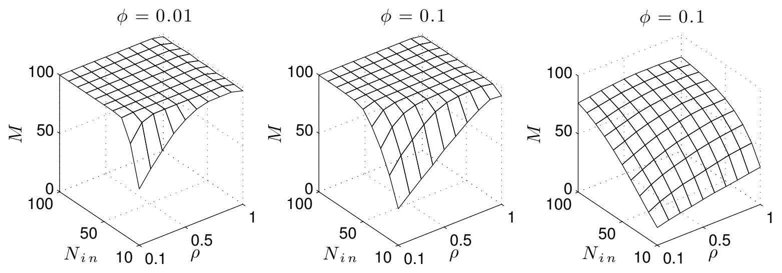

**Figure 3.3:** Total LMC $M$ for different input scaling factors. On the left the reservoir is in the quasilinear regime, in the middle moderately nonlinear and on the right highly nonlinear.

## 3.2.2 Uncorrelated input

### 3.2.2.1 Total memory capacity

The first situation I consider is when all the principal components of the input channels have equal power. I look at the total memory capacity $M$ as a function of the spectral radius and the number of input channels. Figure 3.3 shows the results for three different input scaling factors. In the quasilinear regime (left window), $M$ is mostly equal to the number of reservoir nodes, virtually independent of the spectral radius or number of input channels. Only when both $N_{in}$ and the spectral radius are small does $M$ drop significantly below $N$. This is again due to the fact that the spectral radius of a reservoir determines the speed at which its transient dynamics fade. If the number of input channels is low, the MFs of the principal components can each extend far into the past. However, if the transients are quenched too quickly by a low spectral radius, recovering the past input from the current states becomes more difficult, resulting in lower $M$. When the number of input channels increases, this effect becomes less severe since the memory of each principal component will extend less far in the past (see next paragraph).
Increasing the input scaling factor aggravates the memory deterioration described above. This is due to the fact that the reservoir states are pushed into the saturating parts of their activation function, which decreases the 'effective' spectral radius (the spectral radius of the Jacobian (Ozturk et al., 2006; Verstraeten and Schrauwen, 2009)), and obviously because the signal undergoes a non-linear transformation which cannot be compensated by the linear readout function.

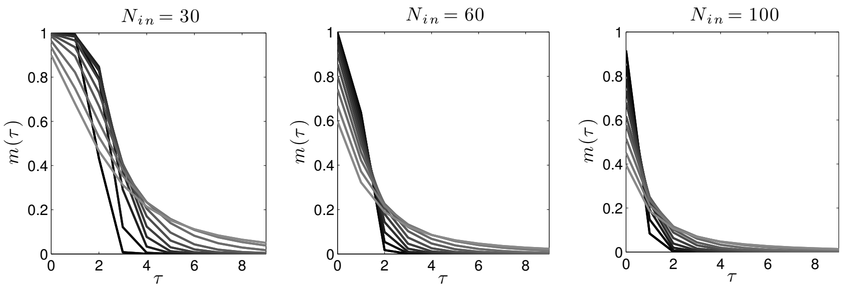

**Figure 3.4:** Average MFs for different $N_{in}$. The spectral radius varies from 0.1 (black) to 1 (light gray line). Horizontal axes have the same scale in each window. The input scaling $\phi$ is equal to 0.1.

#### 3.2.2.2 Shape of the memory function

To get a good idea of exactly what the reservoir remembers of the input, it is useful to take a look at the memory functions themselves. Figure 3.4 shows the average MF $m(\tau) = N_{in}^{-1} \sum_{i=1}^{N_{in}} m_i(\tau)$ in different situations. One obvious fact is that, as $N_{in}$ increases, the amount of memory capacity available for each individual input channel decreases. The next interesting fact is that the spectral radius will greatly determine the shape of the MF. If $\rho$ is small, the reservoir will have a good memory of only a few steps back in the past, and if it is close to one, the reservoir memory extends further but is less precise. This means that tasks with high input dimensionality that need short but precise memory may in fact benefit from choosing a small spectral radius.

### 3.2.3 Generalized signals

Typically, high dimensional input data consists of principal components with widely varying powers. Usually, only a few principal components contain the bulk of the variance and a large fraction of principal components have low powers and contain less meaningful features or noise. In this section we will consider the impact of the power contained in each component on its memory capacity. We use $N_{in} = 50$ and $\phi = 0.1$. The input signal consists of 50 white noise channels with variance $\psi_i = 0.8^i$. This allows us to measure memory capacity as a function of signal power for about 5 orders of magnitude. Figure 3.5 shows the results measured for three reservoirs with different spectral radii.

Ideally, we would wish that a good ESN allocates memory capacity proportional to the signal power. That way, the hidden state will have a 'fair'

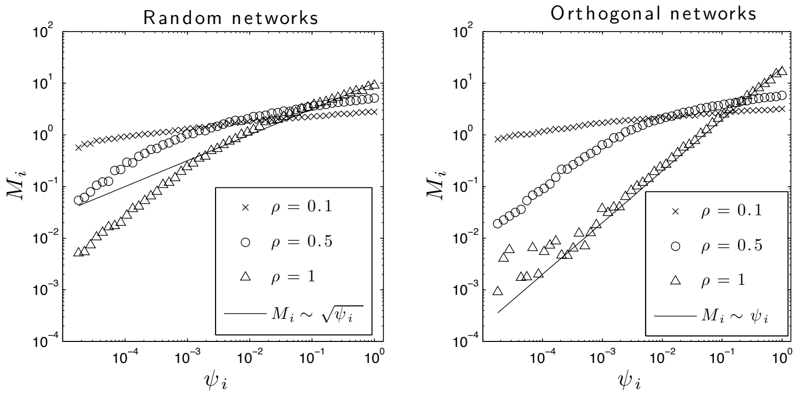

**Figure 3.5:** Individual memory capacities per channel versus the corresponding power (shown on logarithmic scale). Left panel depicts results for random networks, the right panel for random orthogonal networks.

representation of the recent input history. If we look at the left panel of Figure 3.5, we can see that first of all, in the case of a very low spectral radius, the network allocates very roughly the same amount of capacity to each input signal, regardless of signal power. Indeed, in the extreme case that $\rho = 0$, the hidden state would merely be a (slightly non-linear) projection of the current input signal, and as long as $N_{in} < N$, all of these can be perfectly reproduced by the inverse projection. As the spectral radius increases, so will the proportionality between signal power and memory capacity. Still, when we consider the case where $\rho = 1$, we find that, crudely, $M_i \sim \sqrt{\psi_i}$. It's important to know that the *total* LMC in this experiment is still equal to the number of nodes; the theoretical maximum. Yet, it seems that signals with low power hog a disproportionally large part of the available LMC.

Remember that random orthogonal networks are particularly robust against noise. In the case of a broad spectrum of input signal powers, we can state that, as far as the reproduction of each individual principal component is concerned, all the other input channels act as noise. Therefore we repeat the previous experiment, but this time we use random orthogonal connection matrices. Results are depicted in the right panel of Figure 3.5. It is immediately clear that orthogonal networks in fact do have the desired proportionality between $\psi_i$ and $M_i$. From this we can conclude that orthogonal networks have superior linear memory properties also in the case of multiple input channels.

## 3.3 Continuous-time linear dynamical systems

As I have stated in section 2.1.3, a lot of interesting problems play out in continuous time. Speech, control problems, robotics, etc., all have a continuous time character. In order to let a computer handle these tasks, time needs to be discretized. If we need high precision for input and output signals (which is usually the case), the sample period $\Delta t$ will have to be small, much smaller than the relevant time scales of the problem.
At this point we normally employ leaky integrators in our ESNs. We essentially simulate a continuous-time dynamical system, simply because discrete time dynamics does not readily lend itself to process slowly varying signals. This section deals with memory for continuous-time dynamical systems. I will redefine MFs and LMC and investigate continuous-time equivalents for orthogonal networks.
Here I limit myself to linear dynamical systems. As it turns out, in this case the quantities of LMC and the MF can be reduced to analytical forms, which greatly aids in the understanding of how memory works. The impact of non-linearity in continuous time is more or less analogous to that in discrete time: non-linearity will eat away LMC, therefore, I do not study it here. Also, I will limit the analysis to the case of one-dimensional input signals. What I do study, however, is the impact of noise on the readout of the hidden state. All results of this paragraph have been published in Hermans and Schrauwen (2010a).

### 3.3.1 Evolution of the hidden state

The hidden state of the linear DS evolves according to

$$\dot{\mathbf{a}}(t) = \mathbf{W}\mathbf{a}(t) + \mathbf{V}s(t). \tag{3.8}$$

The general solution to equation 3.8 in steady state regime (i.e., when the influence of initial conditions has faded) is given by[5]

$$\mathbf{a}(t) = \left[\mathcal{H}(t)e^{\mathbf{W}t}\mathbf{V}\right] * s(t), \tag{3.9}$$

[5]The exponential function of a matrix is defined by its Taylor expansion:

$$e^{\mathbf{W}t} = \sum_{i=0}^{\infty} \frac{1}{i!}\mathbf{W}^i t^i$$

where $\mathcal{H}(t)$ is the Heaviside-step function, and $*$ denotes convolution. If we now assume that $\mathbf{W}$ is diagonalizable, hence that $\mathbf{W} = \mathbf{QDQ}^{-1}$ with $\mathbf{D}$ a diagonal matrix with eigenvalues $\lambda_i$, then we can rewrite the above equation element-wise as

$$\begin{aligned} a_i(t) &= \left[ \mathcal{H}(t)\mathbf{Q}e^{\mathbf{D}t} \underbrace{\mathbf{Q}^{-1}\mathbf{V}}_{\mathbf{p}} \right]_i * s(t) \\ &= \mathcal{H}(t) \sum_{j=1}^{N} Q_{ij} p_j \exp(\lambda_j t) * s(t) \\ &= [\mathbf{Tz}(t)]_i \,, \end{aligned}$$

where $T_{ij} = C_{ij}p_j$ and $z_i(t) = \mathcal{H}(t)\exp(\lambda_i t) * s(t)$. This means that any linear combination of reservoir states can be written as a linear combination of the output of filters with impulse response $\exp(\lambda_i t)$ operating on the input signal. These are all well known basic results from linear dynamical system theory as found in for instance Sontag (1998).
When considering $z_i(t)$, we can immediately see the stability condition for linear continuous time DSs: the real parts of all $\lambda_i$ need to be negative. Otherwise, the filters would grow exponentially in time.

### 3.3.2 Modeling noise

Noise can appear in many different forms: one can inject noise into the neurons as extra input, noise can be superposed on the input signal, or the neurons can be inherently noisy, superposing noise on their output etc. In this work, I restrict myself to the last option, which means I superpose a term $\sigma(\mathbf{a}')\sqrt{\epsilon}\mathbf{h}(t)$ on the reservoir states, where $\sigma(\mathbf{a}')$ is the mean over all neurons of the standard deviation of the hidden states when no noise is present, $\epsilon$ is the signal-to-noise ratio (SNR), and $\mathbf{h}(t)$ is the noise signal with unit standard deviation and zero mean. Scaling the noise to the mean standard deviation of the reservoir states has the advantage that one does not need to account for internal amplification of the input signal.
If we further assume that this noise has a bandwidth which is far greater than the pass-bands of the neurons, it is easy to see that noise will propagate through the network only to a limited degree. Low-pass filtering of a signal comes down to taking its average value over an exponential window. If the noise fluctuates very fast compared to the time scale of the low-pass filtering operation, the output of this filtering will be very small in amplitude. Each neuron acts as a low pass filter or an integrator of some sort and hence the states of the neurons are assumed to filter out the noise on their input. We

will therefore assume that no noise propagation exists in the network, which leads to the following reservoir states

$$\mathbf{a}(t) = \mathbf{T}\mathbf{z}(t) + \sigma(\mathbf{a}')\sqrt{\epsilon}\mathbf{h}(t). \tag{3.10}$$

Note that in any realistic scenario, noise superposed on the reservoir states will propagate through the network to some degree, which means this model is only applicable in some situations. However, this approximation allows us to form a direct link between eigenvalues of the covariance matrix of the reservoir states, and noise sensitivity (see paragraph 3.3.7).
Extending the analytical model to fully account for noise propagation is in fact not very difficult, but since this introduces extra parameters (particularly noise spectral range) I choose this simpler model. There is another advantage to this model: if the reservoir is a physical system, reading out the reservoir state will have to be done by some sort of measurement, which is usually inherently noisy. Assuming non-propagating noise obviously also serves to model limitations on readout precision. When noise on the readout mechanism is much more intense than the noise propagating through the network, the above approximation will be valid a fortiori.

## 3.3.3 Definitions

### 3.3.3.1 Memory function for continuous time

We can readily take the discrete-time memory function and use it for continuous time. After all, all operators are well-defined in both, (mean over time, variance and covariance). Importantly, we define the operator $\langle\ \ \rangle_t$ as

$$\langle f(t)\rangle_t = \lim_{T\to\infty} \frac{1}{2T}\int_{-T}^{T} f(t)dt,$$

For simplicity I will assume that $s(t)$ has zero mean and unit variance. Completely analogous to equation 3.5 we get

$$m(\tau) = \mathbf{g}^{\mathsf{T}}(\tau)\mathbf{C}^{-1}\mathbf{g}(\tau), \tag{3.11}$$

with $\mathbf{g}(\tau) = \langle \mathbf{a}(t)s(t-\tau)\rangle$ and $\mathbf{C}$ the covariance matrix of $\mathbf{a}(t)$. We can now reduce $\mathbf{g}(\tau)$ and $\mathbf{C}$ by rewriting them in terms of $\mathbf{z}(t)$ and keeping into account the noise I superposed on the hidden state. Starting with $\mathbf{g}(\tau)$, we get:

$$\begin{aligned}
\mathbf{g}(\tau) &= \left[\langle \mathbf{Tz}(t)s(t-\tau)\rangle_t + \sigma(\mathbf{a}')\sqrt{\epsilon}\,\langle \mathbf{h}(t)s(t-\tau)\rangle_t\right]_i \\
&= \sum_{j=0}^{N} T_{ij} \left\langle \int_0^\infty dt' s(t-t')e^{\lambda_j t'} s(t-\tau)\right\rangle_t \\
&= \sum_{j=0}^{N} T_{ij} \int_0^\infty dt' \exp(\lambda_j t')\, \underbrace{\langle s(t-t')s(t-\tau)\rangle_t}_{R(t'-\tau)} \\
&= \sum_{j=0}^{N} T_{ij} \underbrace{\int_0^\infty dt' \exp(\lambda_j t') R(t'-\tau)}_{b_j(\tau)},
\end{aligned}$$

where $R(t)$ is the autocovariance function of the input signal. Writing this in matrix notation gives

$$\mathbf{g}(\tau) = \mathbf{Tb}(\tau).$$

Using the same reasoning, we can calculate the covariance matrix $\mathbf{C}$. Notice that, since $\mathbf{a}(t) = \mathbf{Tz}(t) + \sqrt{\epsilon}\mathbf{h}(t)$, with $\mathbf{a}(t)$ real, and $\mathbf{T}$ and $\mathbf{z}(t)$ generally complex, we can write $\mathbf{a}^\mathsf{T}(t) = \mathbf{a}^\dagger(t) = \mathbf{z}^\dagger(t)\mathbf{T}^\dagger + \sigma(\mathbf{a}')\sqrt{\epsilon}\mathbf{h}^\dagger(t)$, where $\dagger$ stands for the Hermitian transpose. This allows us to calculate a Hermitian form for $\mathbf{C}$:

$$\begin{aligned}
\mathbf{C} &= \mathbf{T}\underbrace{\left\langle \mathbf{z}(t)\mathbf{z}^\dagger(t)\right\rangle}_{\mathbf{C_z}}\mathbf{T}^\dagger + \epsilon\sigma^2(\mathbf{a}')\left\langle \mathbf{h}(t)\mathbf{h}^\dagger(t)\right\rangle \\
&= \mathbf{TC_zT}^\dagger + \epsilon\sigma^2(\mathbf{a}')\mathbf{I}.
\end{aligned}$$

Here, $\mathbf{C_z}$ is the covariance matrix for the responses of the filters $\exp(\lambda_i t)$, which can now be calculated the same way as the elements of $\mathbf{b}(\tau)$:

$$[\mathbf{C_z}]_{ij} = \int_0^\infty dt \int_0^\infty dt' R(t-t')\exp(\lambda_i t)\exp(\lambda_j^* t').$$

The variances of the states of the individual neurons without noise are given by the diagonal elements of the covariance matrix, which yields $\sigma^2(\mathbf{a}') = N^{-1}\mathrm{tr}(\mathbf{TC_zT}^\dagger)$. This number is also equal to the mean eigenvalue of the noiseless covariance matrix. When we denote these as $\xi_i$, we can write $N^{-1}\mathrm{tr}(\mathbf{TC_zT}^\dagger)$ as $\bar{\xi}$ for short. Combining these equations finally leads to a useful expression for the memory function:

$$m(\tau) = \mathbf{b}^\dagger(\tau)\mathbf{T}^\dagger\left(\mathbf{TC_zT}^\dagger + \bar{\xi}\epsilon\mathbf{I}\right)^{-1}\mathbf{Tb}(\tau). \tag{3.12}$$

This expression allows for a quick numerical evaluation of the memory function for linear dynamical systems. When $\epsilon = 0$, the memory function reduces to

$$m(\tau) = \mathbf{b}^{\dagger}(\tau)\mathbf{C_z}^{-1}\mathbf{b}(\tau), \tag{3.13}$$

which means that without noise, $m(\tau)$ depends solely on the eigenvalues of $\mathbf{W}$ and not on the input vector or connection topology, and it remains invariant under a similarity transformation of $\mathbf{W}$. We can for instance replace $\mathbf{W}$ with its diagonal form $\mathbf{D}$ which reduces the network to a set of decoupled complex filters. If real elements are required, each complex pair of eigenvalues can be replaced by a $2\times2$ block on the diagonal of $\mathbf{W}$, resulting in a damped oscillator. Real eigenvalues simply represent disconnected low-pass filters.

### 3.3.3.2 Memory capacity and memory quality

Here I introduce the continuous time equivalent of LMC and introduce a new variable, which I shall call *memory quality.* LMC is quite easily generalized to continuous time

$$M = \int_0^\infty m(\tau)d\tau. \tag{3.14}$$

Note that here, $M$ has dimension time, as the MF itself is dimensionless. As we will discover soon, LMC for continuous time systems has one odd feature: it becomes maximal when the MF approaches zero, but stretches out to infinity. In other words, the LMC is greatest when memory extends very far into the past, but is very bad everywhere. For this reason LMC is not the best indicator of good memory, and we introduce memory quality $M_q(x)$:

$$M_q(x) = \frac{1}{x}\int_0^x m(\tau)d\tau. \tag{3.15}$$

This measure is always smaller than or equal to 1 and is a number which denotes the average MF up to a time $x$ in the past which can be chosen at a value relevant for a certain task or type of reservoir. I will mostly take $x = M$. In this case, the memory quality expresses the relative amount of memory which is actually present in a range equal to the memory capacity. If $M_q(M) = 1$, the MF will be equal to one up to $\tau = M$, and then abruptly fall to zero.

$M$ as well as $M_q(x)$ can be straightforwardly calculated from equation 3.12. To do this, one needs to calculate the integrals over the crossproducts of the elements $b_i(\tau)$. Resulting formulas become quite complex, particularly for $M_q(x)$ and hence I omit them here.

#### 3.3.3.3 Reservoir timescale

The last parameter I introduce describes the intrinsic time scale of a reservoir, which I will name the *reservoir timescale* $\tau_R$ and define as

$$\tau_R = -\left(\frac{\mathrm{tr}(\mathbf{W})}{N}\right)^{-1} = -\left(\frac{1}{N}\sum_{i=1}^{N}\lambda_i\right)^{-1}. \tag{3.16}$$

This definition is based on a reservoir in which all neurons act as low-pass filters with the same timescale $\tau_R$, where all diagonal elements of $\mathbf{W}$ are equal to $-\tau_R^{-1}$. Note that this does not imply that, for a network with a certain $\tau_R$, all neurons have to act as low pass filters with the same timescale, which would imply all diagonal elements of $\mathbf{W}$ would have to be equal.

### 3.3.4 Input signal model

We previously saw that in the expressions for $\mathbf{b}(\tau)$ and $\mathbf{C_z}$ we automatically encounter a very important statistical feature of the input signal: the autocovariance function $R(t)$. To perform empirical studies of the MF, I will assume an input autocovariance function of the form

$$R(t) = \exp(-\alpha|t|),$$

which describes a signal which is limited in bandwidth by the finite autocovariance length $\alpha$, and where I define the signal timescale as $\alpha^{-1}$. This serves as an analogy to the signal used in discrete networks and is quite common for many natural stochastic processes. Using this function, we can calculate the elements of $\mathbf{C_z}$ and $\mathbf{b}(\tau)$:

$$[\mathbf{C_z}]_{ij} = \frac{1}{(\alpha-\lambda_i)(\alpha-\lambda_j^*)}\left(1-\frac{2\alpha}{\lambda_i+\lambda_j^*}\right) \tag{3.17}$$

$$b_i(\tau) = \frac{(\lambda_i-\alpha)e^{-\alpha\tau}+2\alpha e^{\tau\lambda_i}}{(\alpha^2-\lambda_i^2)}. \tag{3.18}$$

### 3.3.5 Empirical validation

Armed with the analytical expressions in equations 3.17 and 3.18, we can now finally validate equation 3.12 empirically. I simulated a 10-neuron toy network (the shifted random network). To approximate a continuous-time neural network, I used discrete time steps of 1 ms in a reservoir with a time scale $\tau_R = 1$ s. Generating a signal which has $R(t) = \exp(-\alpha|t|)$

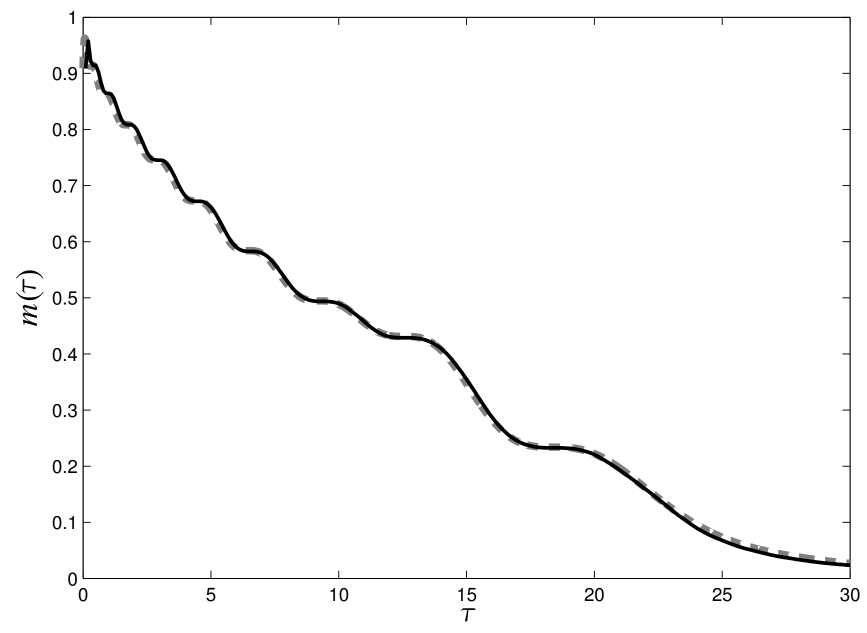

**Figure 3.6:** Comparison of the empirically determined MF versus the result obtained in equation (3.13). The grey dashed line is the analytical prediction, the black solid line the empirical measurement. Setup of the experiment is described in the text.

was performed by creating a signal which was sampled from a Gaussian distribution each time step, and then low-pass filter this with time scale $\alpha^{-1}$. The resulting signal can easily be proved to approximately have the desired autocovariance function. As parameters I chose, $\alpha = 1$ Hz, $\rho = 0.9$ (where $\rho$ is the spectral radius of the matrix used to construct $\mathbf{W}$, as fully explained in section 3.3.8.1), and the MF was numerically evaluated using equation (3.13). Simulation was performed for a duration of $2 \times 10^4$ s ($2 \times 10^7$ time steps). No noise was imposed on the signal; the only limitations on accuracy were the finite simulation time step and numerical precision. The result is depicted in Figure 3.6, which shows a very good correspondence between theory and experiment.

### 3.3.6 Asymptotic memory capacity

Before I start to investigate different ways of constructing networks, I will discuss the limiting case for the memory capacity for when $\tau_R$ goes to infinity and no noise is present. First of all, we look more closely at the definition of the memory capacity:

$$\begin{aligned} M &= \int_0^\infty m(\tau)d\tau \\ &= \sum_{i=1}^N \sum_{j=1}^N \int_0^\infty b_i^*(\tau)b_j(\tau)\left[\mathbf{C_z}^{-1}\right]_{ij} d\tau \\ &= \operatorname{tr}\left(\mathbf{C_z}^{-1}\int_0^\infty \mathbf{b}(\tau)\mathbf{b}^\dagger(\tau)d\tau\right). \end{aligned}$$

With this definition and equations (3.17) and (3.18) we can calculate the asymptotic limit. To do this, I again define normalized eigenvalues $\hat{\lambda}_i = \tau_R\lambda_i$, and normalize time on the reservoir timescale: $\theta = \tau/\tau_R$. Transforming the integration variable from $\tau$ to $\theta$ gives

$$\int_0^\infty b_i(\tau)b_j^*(\tau)d\tau = \tau_R\int_0^\infty b_i(\theta)b_j^*(\theta)d\theta.$$

Expanding $b_i(\theta)$ and taking the limit $\tau_R \to \infty$ gives

$$\begin{aligned} \lim_{\tau_R\to\infty} b_i(\theta) &= \lim_{\tau_R\to\infty} \frac{\left((\frac{\hat{\lambda}_i}{\tau_R} - \alpha)e^{-\alpha\tau_R\theta} + 2\alpha e^{\theta\hat{\lambda}_i}\right)}{(\alpha^2 - \left(\frac{\hat{\lambda}_i}{\tau_R}\right)^2)} \\ &= \frac{2e^{\theta\hat{\lambda}_i}}{\alpha}. \end{aligned}$$

This yields

$$\lim_{\tau_R\to\infty} \tau_R\int_0^\infty b_i(\theta)b_j^*(\theta)d\theta = \lim_{\tau_R\to\infty} \frac{-4\tau_R}{\alpha^2(\hat{\lambda}_i + \hat{\lambda}_j^*)}.$$

We can similarly apply the limit to the elements of $\mathbf{C_z}$, where we find

$$\lim_{\tau_R\to\infty} [\mathbf{C_z}]_{ij} = \lim_{\tau_R\to\infty} \frac{-2\tau_R}{\alpha^2(\hat{\lambda}_i + \hat{\lambda}_j^*)}.$$

Finally, we can write the the memory capacity as

$$\lim_{\tau_R\to\infty} M = \frac{2}{\alpha}\operatorname{tr}(\mathbf{I}) = \frac{2}{\alpha}N. \tag{3.19}$$

When $\tau_R \to \infty$, the memory function will stretch on to infinity and consequently, it will be infinitesimally close to zero, so at first sight the above calculation might not seem very useful. However, there are strong suggestions

that this limit might in fact be an upper bound for the memory capacity of linear first order networks. It seems the memory capacity always rises monotonically with $\tau_R$ (see following sections), and reaches an asymptotic upper limit for very large $\tau_R$. Furthermore, in Section 3.3.9.3, when researching a special kind of reservoirs where we can find approximate solutions for the memory capacity, we can also confirm that this number is the maximal value for memory capacity. So far, I have not been able to find mathematical proof that this in fact a true upper bound, and for now I will leave this as a conjecture to be proven or disproven in future research. Notice that this expression is, just like in discrete time networks, proportional to $N$ and links memory capacity to signal statistics, suggesting that each neuron is capable to store a maximal amount of "information" equal to $2/\alpha$, just like in discrete time each single neuron is capable to store one time step of the input signal.
If we recall that the typical timescale of the signal is $\alpha^{-1}$, it follows that the slower the input signal varies, the easier it is to store its history. In the next sections, I shall investigate a few important reservoir types, using results which have been acquired for discrete time networks which I translate to the continuous time domain.

## 3.3.7 Memory and noise sensitivity

Before moving to empirical testing of different reservoir types, I derive a general expression which connects the noise sensitivity of the MF to a basis of orthogonal reservoir states and the eigenvalues of the covariance matrix $\mathbf{C}$. The operations performed in this section are highly similar to those in Principal Component Analysis, where I apply it on continuous time functions instead of the more common discrete data points.
The covariance matrix $\mathbf{C} = \mathbf{T}\mathbf{C}_\mathbf{z}\mathbf{T}^\dagger$ is a real symmetric positive-definite matrix[6] and has an eigendecomposition such that $\mathbf{T}\mathbf{C}_\mathbf{z}\mathbf{T}^\dagger = \mathbf{O}\boldsymbol{\Xi}\mathbf{O}^\dagger$, where $\mathbf{O}$ is orthogonal, and $\boldsymbol{\Xi}$ a diagonal matrix with only positive real eigenvalues $\xi_i$. Notice that, since $\mathbf{O}\mathbf{O}^\dagger = \mathbf{I}$, we can write

$$\begin{aligned}(\mathbf{T}\mathbf{C}_\mathbf{z}\mathbf{T}^\dagger + \bar{\xi}\epsilon\mathbf{I})^{-1} &= (\mathbf{O}\boldsymbol{\Xi}\mathbf{O}^\dagger + \bar{\xi}\epsilon\mathbf{O}\mathbf{O}^\dagger)^{-1}\\ &= \mathbf{O}^\dagger(\boldsymbol{\Xi} + \bar{\xi}\epsilon\mathbf{I})^{-1}\mathbf{O}\\ &= \mathbf{O}(\boldsymbol{\Xi} + \bar{\xi}\epsilon\mathbf{I})^{-1}\mathbf{O}^\dagger.\end{aligned}$$

[6]For shifted random matrices, discussed in the next paragraph, some negative eigenvalues can be found, but these are quite probably due to errors originating from limited numerical precision.

This allows us to write the MF as follows

$$\begin{aligned} m(\tau) &= \underbrace{\mathbf{b}^\dagger(\tau)\mathbf{T}^\dagger\mathbf{O}}_{\boldsymbol{\beta}'^\dagger(\tau)}(\boldsymbol{\Xi}+\bar{\xi}\epsilon\mathbf{I})^{-1}\underbrace{\mathbf{O}^\dagger\mathbf{T}\mathbf{b}(\tau)}_{\boldsymbol{\beta}'(\tau)} \\ &= \sum_{i=1}^{N}\frac{\beta'^2_i(\tau)}{\xi_i+\bar{\xi}\epsilon}. \end{aligned}$$

Notice that $\boldsymbol{\beta}'(\tau)$ is strictly real. Defining $\beta_i(\tau) = \beta'^2_i(\tau)\xi_i^{-1}$, we can write this as

$$m(\tau) = \sum_{i=1}^{N}\frac{\beta_i(\tau)}{1+\bar{\xi}\xi_i^{-1}\epsilon}. \qquad (3.20)$$

This means that we can decompose the MF as a set of functions which are all real, positive and between zero and one. The terms in this equation are ordered according to decreasing size of the eigenvalues $\xi_i$, with $i = 1$ corresponding to the largest of $\xi_i$ and as such in a declining order.
There is a clear interpretation of the functions $\beta_i(\tau)$. Suppose we wish to linearly transform the reservoir states $\mathbf{a}(t)$ in the absence of noise, so as to define a basis of states $\hat{\mathbf{a}}(t)$ which have the property that $\left\langle\hat{\mathbf{a}}(t)\hat{\mathbf{a}}^\mathsf{T}(t)\right\rangle = \mathbf{I}$, i.e., a set of orthogonal (uncorrelated) states with unit standard deviation. The above procedure does in fact perform this transformation. When we implicitly define $\hat{\mathbf{a}}(t)$ as

$$\mathbf{a}(t) = \mathbf{O}\boldsymbol{\Xi}^{1/2}\hat{\mathbf{a}}(t), \qquad (3.21)$$

it can be checked that this yields the desired expression for $\left\langle\mathbf{a}(t)\mathbf{a}^\mathsf{T}(t)\right\rangle$. We can define the orthogonal base states from this expression:

$$\hat{\mathbf{a}}(t) = \boldsymbol{\Xi}^{-1/2}\mathbf{O}^\dagger\mathbf{a}(t), \qquad (3.22)$$

which yields the desired covariance matrix. Looking at the original definition of the MF, we can now rewrite it in terms of the base states $\hat{\mathbf{a}}(t)$, and in the absence of noise:

$$m(\tau) = \sum_{i=1}^{N}\left\langle s(t-\tau)\hat{a}_i(t)\right\rangle^2, \qquad (3.23)$$

so that $\beta_i(\tau) = \left\langle s(t-\tau)\hat{a}_i(t)\right\rangle^2$.
Each of the terms in (3.20) has a clear dependence on its corresponding eigenvalue $\xi_i$ and its noise sensitivity. We can make a very rough estimate of the MF for a certain noise intensity by approximating $\epsilon\bar{\xi}/\xi_i$ as zero when $\epsilon < \xi_i/\bar{\xi}$, and $\epsilon\bar{\xi}/\xi_i = \infty$ when $\epsilon > \xi_i/\bar{\xi}$. This means that we only add up terms in (3.20) up to $i = k$ where $\epsilon < \xi_k/\bar{\xi}$ and $\epsilon > \xi_{k+1}/\bar{\xi}$. This estimate is inaccurate but gives a graphical interpretation of the $\beta$-functions in (3.20).

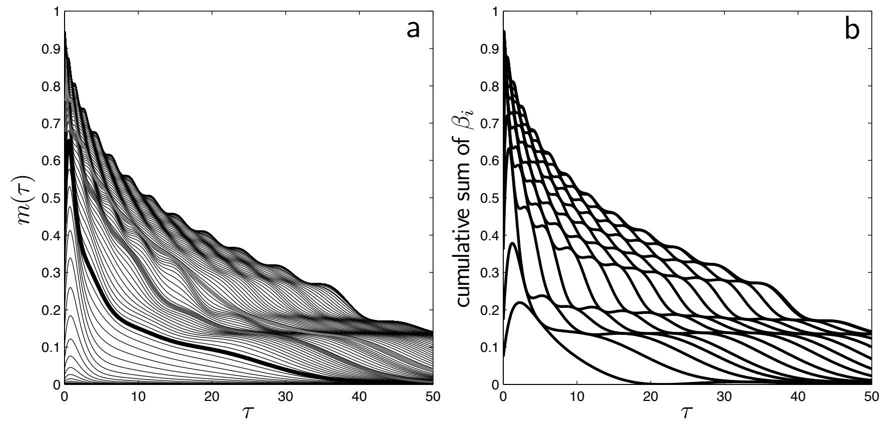

**Figure 3.7:** (a) Example of the MF for different values of $\epsilon$ of a 20-neuron network constructed as described in section 3.3.8.1 with timescale $\tau_R = 2$, and $\rho = 0.9$. The thick black line is for $\epsilon = 1$, the dark grey line for $\epsilon = 10^{-3}$ and the light grey line for $\epsilon = 10^{-6}$. Notice the strong sensitivity for noise: even for a signal-to-noise ratio of the order $10^{-6}$, the memory function is still not close to its asymptotic convergence to its ideal for $\epsilon = 0$ (highest line). (b) Cumulative sum of $\beta$-functions. The $k$-th line from the bottom up is the sum of the $\beta$-functions up to $k$. The last 4 $\beta$-functions were not added due to the fact that the smallest of the eigenvalues $\xi_i$ cannot be calculated accurately, resulting in irregular behavior

Figure 3.7 shows a graphical representation of the MF for different noise values on the right, and a cumulative sum of $\beta$-functions on the left. Notice the increasing number of oscillations on the $\beta$-functions, which gives rise to the final shape of the MF when no noise is present.
There is a very clear interpretation for this type of noise sensitivity. When writing equation 3.21 element-wise, one can see that the base states $\hat{a}_j$ are each encoded in the reservoir state $\mathbf{a}$ with a magnitude $\sqrt{\xi_j}$:

$$a_i(t) = \sum_{j=1}^{N} U_{ij}\sqrt{\xi_j}\hat{a}_j(t).$$

The smaller $\xi_j$ gets, the smaller the actual contribution of $\hat{a}_j$ is to the reservoir states, and the more it will "drown" into the surrounding noise.
This result reflects facts which have been suggested by others as well. For instance, in Jaeger (2001b) it is mentioned that the memory capacity of neural networks is limited by the conditioning of the covariance matrix, where

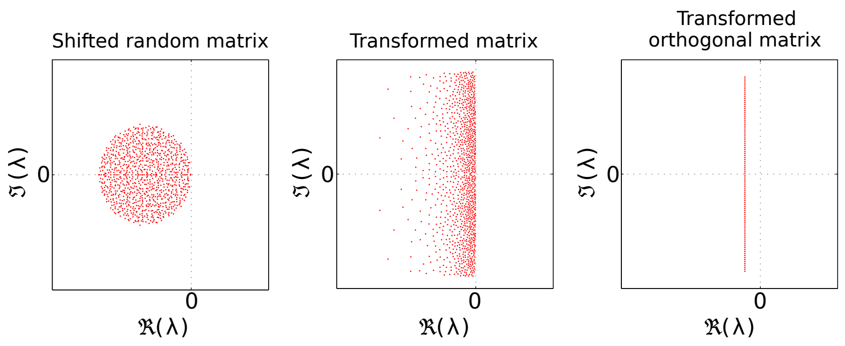

**Figure 3.8:** Example eigenspectra for the three different types of connection matrices. On the left is the shifted random matrix, in the middle the inverse $z$-transformed matrix, and on the right is the spectrum of a resonator reservoir.

it was found that for discrete time reservoirs, the measured memory capacity always is slightly smaller than its theoretical value. Since the condition number of the covariance matrix is equal to the ratio between its highest and lowest singular values, which are equal to the eigenvalues because $\mathbf{C}$ is a positive semi-definite matrix, this effect is directly linked to our result.
In Ozturk et al. (2006) a different approach is used where the goal is to maximize the entropy of the reservoir states in order to span the widest possible range of nonlinear mappings of the input signal. It was found that - in order to do this - the reservoir states should be as little correlated as possible. When the reservoir states are all uncorrelated, we essentially get the orthogonal base states $\hat{a}_i(t)$, where the eigenvalues of the covariance matrix are equal to the individual variances of the reservoir states, which means that they will not become excessively low.

### 3.3.8 Constructing weight matrices

Notice the difference between equation 3.8, and the mathematical notation I used in paragraph 2.1.3. Here, I do not explicitly add the leak term to the equation, but I rather assume that it is included in the matrix $\mathbf{W}$.
As mentioned, we require that the real parts of the eigenvalues of $\mathbf{W}$ are negative. For random matrices, this is definitely not the case. In my research I have explored three strategies to construct stable connection matrices for continuous time.

#### 3.3.8.1 Shifted random matrices

The easiest and most intuitive way to construct stable weight matrices is to reach back to the previous chapter and use a linear approximation of equation 2.3. This gives us

$$\begin{aligned}\dot{\mathbf{a}}(t) &= \frac{1}{\tau_R}\left(\mathbf{W}'\mathbf{a}(t) - \mathbf{a}(t) + \mathbf{V}\mathbf{s}(t)\right)\\ &= \frac{1}{\tau_R}(\mathbf{W}' - \mathbf{I})\mathbf{a}(t) + \frac{1}{\tau_R}\mathbf{V}\mathbf{s}(t),\end{aligned}$$

where I have used $\mathbf{W}'$ to discern it from the $\mathbf{W}$ I use in this section. If we compare this with equation 3.8, we see that

$$\mathbf{W} = \frac{1}{\tau_R}(\mathbf{W}' - \mathbf{I}).$$

Suppose $\mathbf{W}'$ is a random matrix with $\rho \leq 1$. What can we say about the eigenvalues of $\mathbf{W}$? When we use the eigenvalue decomposition of $\mathbf{W}'$ we can write that

$$\begin{aligned}\mathbf{W} &= \frac{1}{\tau_R}(\mathbf{Q}'\mathbf{D}'\mathbf{Q}'^{-\mathbf{1}} - \mathbf{I})\\ &= \mathbf{Q}'\frac{\mathbf{D}' - \mathbf{I}}{\tau_{\mathbf{R}}}\mathbf{Q}'^{-\mathbf{1}}.\end{aligned}$$

This means that $\mathbf{W}$ has the same eigenvalues as $\mathbf{W}'$, minus one, and divided by $\tau$. As all eigenvalues of $\mathbf{W}'$ are in the unit circle, the shift to the left on the complex plane will guarantee that their real parts are smaller than zero. Hence, we can very easily construct applicable continuous time connection matrices. Note that $\tau_R$ falls back to its previous definition: The average of the eigenvalues of $\mathbf{W}'$ will have an expectation value of zero[7]. The average eigenvalue of $\mathbf{I}$ is obviously equal to one, and this means that the average eigenvalue of $\mathbf{W}$ is equal to $\tau_R^{-1}$.

#### 3.3.8.2 Transformed matrices

In Ozturk et al. (2006), it is suggested that good ESNs should have an eigenvalue spectrum that evenly and uniformly distributes the eigenvalues over the unit disk. For random matrices, this is already approximately the case. The approach with shifted random matrices doesn’t take into account the

[7]A useful result from linear algebra is that the sum of all the eigenvalues of a matrix is equal to the trace of that matrix. As such, the average of the eigenvalues is also the average of the trace, which for random matrices has an expectation value of zero.

meaning of growth rate and frequency. Let us consider the case for discrete time, where eigenvalues are uniformly and evenly distributed over the unit disk. The associated eigenmodes of the system have frequencies which are uniformly distributed between the minimum frequency (zero) and the maximum ($\pi$). The growth rates of the eigenmodes are similarly broadly distributed: from the eigenvalues in the centre, (decaying instantly to zero), to the edge of the unit disk (decaying infinitely slow). In continuous time, however, the meaning of growth rate and frequency has changed. A shifted random matrix in continuous time will have eigenmodes with frequencies and growth rates that are distributed very differently than in the discrete time case.
We wish to transform the distribution of discrete-time frequencies and growth rates to continuous time. I will do this via the inverse $z$-transform. I explain the details in appendix A.2.1.1. The resulting matrices have eigenvalue distributions are exponential for the real part and uniform for the imaginary part (See Figure 3.8, central panel) .

#### 3.3.8.3 Transformed orthogonal matrices: resonator reservoirs

The third type of connection matrices I consider are those that are based on orthogonal matrices in discrete time. In appendix A.2.3 I describe their construction. The resulting distribution of eigenvalues is a vertical line in the complex plane, crossing the real axis $-\tau_R^{-1}$. I choose the imaginary parts of the eigenvalues equidistantly, with a difference in angular frequency $\omega$ between two successive eigenvalues. We now redefine the index $i$ for the eigenvalues going from $-(N-1)/2$ to $(N-1)/2$ rather than from 1 to $N$. This gives:

$$\lambda_i = \jmath\omega i - \frac{1}{\tau_R}, \tag{3.24}$$

where I use $\jmath$ as the symbol for the imaginary unit to avoid confusion with indices $i$ or $j$.
When no noise is present, we can replace the reservoir by a set of disconnected filters characterized by the eigenvalues. In this scenario the reservoir states are given by

$$a_i(t) = \int_0^\infty \exp(\jmath\omega i t') \underbrace{\exp(-t'/\tau_R) s(t-t')}_{s_W(t,t')}, \tag{3.25}$$

where $s_W(t,t')$ is what we will call the "windowed signal", the signal at a time $t-t'$, multiplied by an exponential window function $\exp(-t'/\tau_R)$. The reservoir states in the above equation are very similar to the first $N$ Fourier

coefficients of a discrete Fourier transform of $s_W$. Signal reconstruction can then be performed by constructing the Fourier series. This kind of reservoir is basically made of a set of damped resonators with equal decay rate and different frequencies. Therefore, we shall simply call them *resonator reservoirs*.
An example spectrum for a resonator reservoir is shown in the right panel of Figure 3.8.

## 3.3.9 Properties of the networks

We can use our analytical model to study the memory for the three different connection matrices. First, I shall define a criterion to optimize the MF of a given reservoir. For practical purposes, this will obviously depend on the task which needs to be performed, but as a general criterion, we wish to find a balance between memory capacity and memory quality. As an overall objective function for optimum memory, we multiply the memory quality by the memory capacity. This number is equal to $\int_0^M m(\tau)d\tau$, and signifies how much memory is in fact present within the range $\{0 \cdots M\}$.

### 3.3.9.1 Shifted random matrices

First I consider shifted random matrices. I investigate the three measures $M$, $M_q(M)$ and $MM_q(M)$ as functions of the reservoir time scale and number of neurons $N$. Results are depicted in the top three panels of Figure 3.9. The highest memory quality is found at low $\tau_R$ (between 0.01 and 0.1), and memory capacity rises monotonically with $\tau_R$. Optimal values for $MM_q(M)$ are found for $\tau_R \approx 2$ and this seems independent of the number of neurons. The shape of the MFs corresponding to high, low, and optimal $\tau_R$ are depicted in the bottom left panel of Figure 3.9. As one can expect, fast reservoirs will have very good memory, but only for a very short history. Slow reservoirs generally have very low MFs which extend very far. The optimal value for $\tau_R$ as found by our criterion tries to balance these two effects by producing a MF with a reasonable range and quality. Still, it is fairly low in a large part of its range, until it drops to about zero at $\tau = 100$.
The bottom right panel of Figure 3.9 shows the average memory capacity as a function of $N$ for different noise levels and $\tau_R = 2$. Two things can be derived from this graph. First, it seems that $M$ does not grow linearly with $N$ at all, not even when $\epsilon = 0$. Secondly, as we have already seen in Figure 3.7, random reservoirs are very sensitive to noise. Both these effects are caused by the fact that the covariance matrices for these types of networks, are very ill-conditioned, with condition numbers which reach the order of $10^{18}$ for $N = 50$, and higher still for higher $N$. Most of the normalized

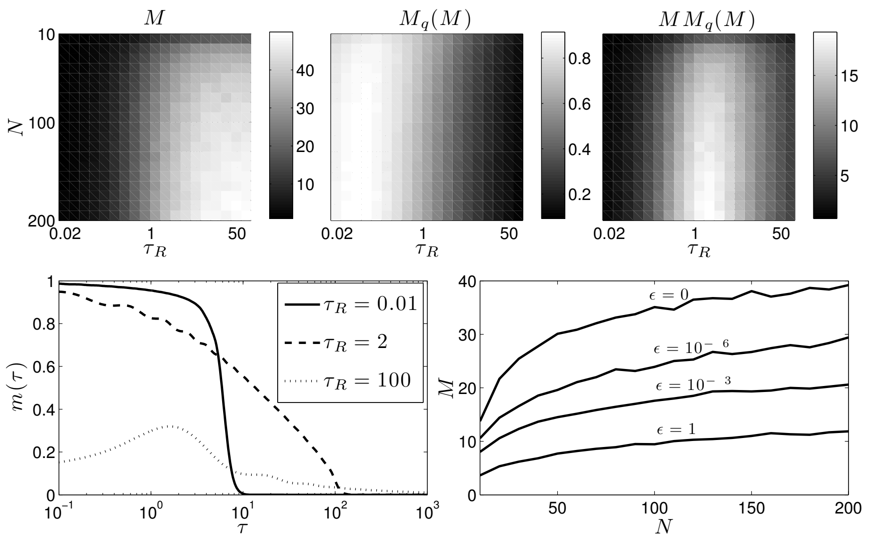

**Figure 3.9:** Top three panels: $M$, $M_q(M)$ and $MM_q(M)$ as a function of $N$ and $\tau_R$ for shifted random networks ($\tau_R$ shown on a logarithmic scale and $\epsilon = 0$). Bottom left panel: memory function with respect to different values of $\tau_r$. The argument $\tau$ is shown on a logarithmic scale, $N = 100$. Bottom right panel: memory capacity as a function of $N$ with respect to different noise levels. For this experiment, $\tau_R = 2$. All results in this figure have been found by averaging over 50 reservoir initializations.

eigenvalues $\xi_i\bar{\xi}^{-1}$ are extremely small and cause the high noise sensitivity. In fact, most numerical values of $\xi_i$ are so small compared to the highest eigenvalue, that for $N > 20$ it becomes virtually impossible to accurately calculate their values, resulting in negative $\xi_i$ (which is definitely incorrect, since $\mathbf{C}$ has to be positive semi-definite).
The relative number of numerically incalculable eigenvalues increases rapidly with $N$. This means that - theoretically - memory capacity may in fact grow linearly with $N$, but confirming this with our analytical model would require much higher numerical precision for inverting $\mathbf{C}$. Another illustration of this fact can be found when numerically trying to validate equation 3.19. For this, I took $\tau_R = 10^5$ (much higher than $\alpha^{-1}$) and calculated the according memory capacity with our analytical model for networks of 100 neurons. Over 50 trials, the average $M$ was equal to 48, approximately 4 times lower than the expected 200. Obviously, such high numerical precision requirements are undesirable and these results lead me to conclude that shifted random

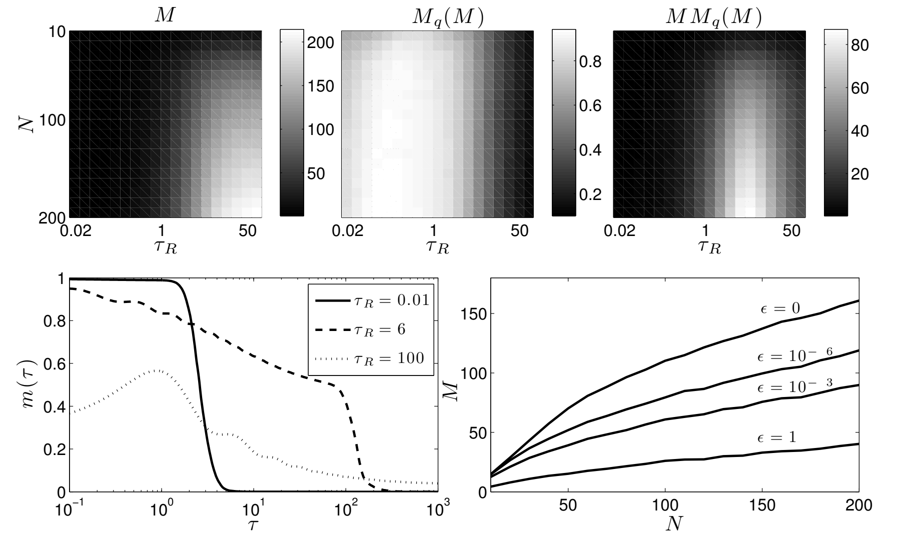

**Figure 3.10:** Top three panels: $M$, $M_q(M)$ and $MM_q(M)$ as a function of $N$ and $\tau_R$ for exponentially distributed eigenvalue networks ($\tau_R$ shown on a logarithmic scale and $\epsilon = 0$). Bottom left panel: MF with respect to different values of $\tau_R$. The argument $\tau$ is shown on a logarithmic scale, $N = 100$. Bottom right panel: memory capacity as a function of $N$ with respect to different noise levels. All results in this figure have been found by averaging over 50 reservoir initializations.

reservoirs in continuous time are unsuitable for good memory storage.

#### 3.3.9.2 Transformed matrices

We now perform the same tests for inverse $z$-transform connection matrices, where again, $\alpha = 1$. Results are depicted in Figure 3.10. This time, the optimal reservoir time scale was found for $\tau_R = 6$ again virtually independent of $N$. Memory capacity rises monotonically with $\tau_R$ as in the previous paragraph. Also, the MF for optimal $\tau_R$ is of a higher overall quality than that found for random reservoirs, with a relatively high value over most of its range until a drop-off at $\tau = 100$. The bottom right panel shows memory capacity at the optimal reservoir time scale. The LMC rises slower than linear with $N$, however, we find significantly better values than for random reservoirs. We can test equation (3.19) using 100 neurons and $\tau_R = 10^5$, averaged over 50 reservoir initializations. This gives an average memory capacity of about 154, which is still below the theoretical value of 200, but the difference is not as dramatic as for random reservoirs.

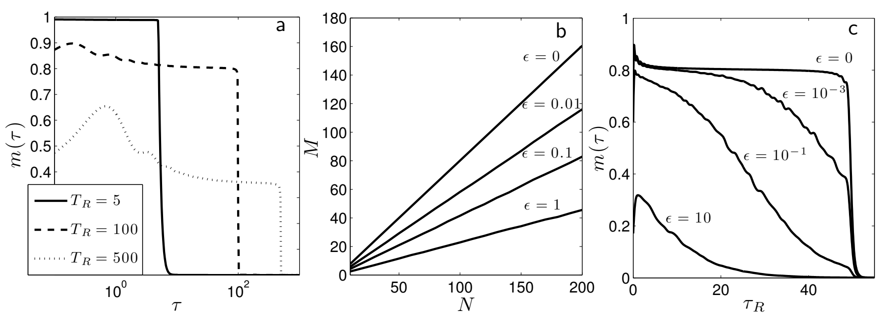

**Figure 3.11:** Properties of resonator reservoirs. (a) Memory function with respect to different values of $T_R$. $\tau$ shown on logarithmic scale, $N = 100$, and $\tau_R = 0.3T_R$. (b) LMC as a function of $N$ for different values of noise. For this experiment, $T_R = N$, and again $\tau_R = 0.3T_R$. (c) Black lines: memory functions at different values of noise. $N = 50$, $T_R = 50$, $\tau_R = 0.3T_R$. All results in this figure have been found by averaging over 50 reservoir initializations.

Clearly, choosing exponentially distributed eigenvalues will give acceptable memory capacity and quality for most tasks. However, looking at the memory capacities at different noise levels, we can again see that they are quite sensitive to noise. Relative to the memory capacity when no noise is present, noise sensitivity is in fact similar to that of random reservoirs, however an absolute comparison shows that even when $\epsilon = 1$, memory capacity is comparable to that of noiseless random reservoirs. The eigenvalue spectrum of $\mathbf{C}$ is indeed generally better conditioned than that of random reservoirs: most eigenvalues can still be calculated accurately and remain in a reasonable range. However, with increasing $N$, most normalized eigenvalues slowly drop, and eventually, more and more become incalculable, explaining the slower-than-linear increase of $M$ as a function of $N$.

#### 3.3.9.3 Resonator reservoirs

Finally I take a look at how well resonator reservoirs perform in terms of LMC. Figure 3.11a provides examples of the shape of the MF, and Figures 3.11 b and 3.11 c display LMC w.r.t $N$ for different levels of noise and the effect of noise on the shape of the MF respectively. It is clear that resonator reservoirs both have excellent noise robustness and their memory capacity scales perfectly with $N$. This reflects the fact that a Fourier decomposition is a very efficient way to decompose a signal. The condition number of the covariance matrix $\mathbf{C}$ is not necessarily low, and is for instance on average

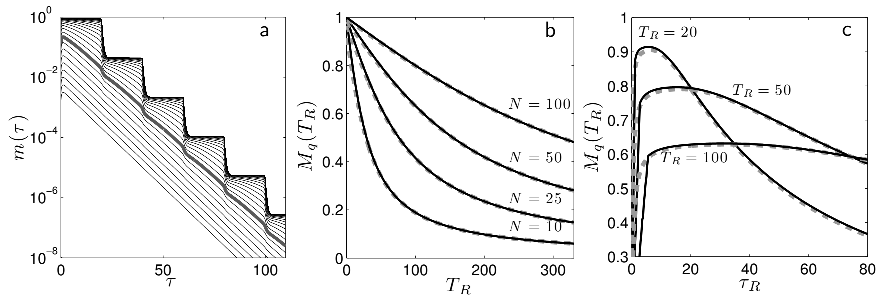

**Figure 3.12:** (a) Depiction of the MF of a reservoir with 50 neurons at different noise levels (the thick grey line is at $\epsilon = 1$). $T_R = 20$ and $\tau_R \approx 13.4$: chosen such that $\exp(-2T_R/\tau_R) = 1/20$. The $y$-axis is on a logarithmic scale to visualize its exponential decay. Notice the sudden drop-off at each multiple of $T_R$. (b) Depiction of $M_q(T_R)$ as a function of $T_R$ for different reservoir sizes. The grey dashed lines are the predictions made by equation (A.12), whereas the black lines are from the full analytical model. $\tau_R$ is chosen to be 0.3 times the reservoir period $T_R$, which is an optimal value found with equation (A.12). (c) The influence of signal interference: $M_q(T_R)$ as a function of $\tau_R$ for different $T_R$, $N$ is chosen at 50. Grey dashed lines are the theoretical prediction found by equation (A.12), black lines are the values found for the full analytical model.

of the order $10^8$ for $N = 100$. However, most of the normalized eigenvalues $\xi_i \bar{\xi}^{-1}$ are not small at all, and usually only few of the normalized eigenvalues are truly small, most are around the order $10^{-2}$
The nature of resonator reservoirs allows us to find excellent analytical approximations of their properties in the absence of noise. In appendix A.2.4 I make the necessary derivations, and here I restrict myself to only providing a list of the most important results.

- Remember that I mentioned that resonator reservoirs essentially perform a truncated discrete Fourier analysis on an exponentially decaying window of the input signal. Obviously, a discrete Fourier transform is only defined on a closed interval, the length of which is equal to the period of the lowest frequency present. This time scale I will call the *reservoir period* $T_R$, which in our case is equal to $T_R = \frac{2\pi}{\omega}$. As is clear from equation 3.25, we can state that the hidden state of the network is the (transformed) set of Fourier coefficients of the windowed signal. However, the exponential window obviously extends beyond the reservoir period $T_R$. If we wish to have good reconstruction for a history

$\tau = \{0 \cdots T_R\}$, we will need to make sure that the exponential window has faded enough to avoid interference from the signal beyond $\tau = T_R$. On the other hand, if the exponential window fades too quickly, signal reconstruction within $\tau = \{0 \cdots T_R\}$ will become compromised.
This leads to the conclusion that resonator reservoirs essentially have two important parameters to play with: the reservoir time scale and the reservoir period. In appendix A.2.4.1 I derive that the MF can be written as the product of two factors: one that drops exponentially with $\tau$, with a decay rate $\frac{\tau_R}{2}$, and a periodic factor with period $T_R$. In figure 3.12a I have drawn an example of the MF of resonator reservoirs at different noise levels, where the described shape of $m(\tau)$ is indeed quite apparent.

- For now, we will consider the memory quality of the reservoir period $M_q(T_R)$ as the parameter we wish to optimize, i.e, we wish to have very good signal reconstruction in the reservoir period and are not interested in anything beyond. Appendix A.2.4.2 provides a (rather lengthy) derivation for this quantity. We find that approximately

$$M_q(T_R) = \frac{2}{\pi}\left[1 - e^{-2\frac{T_R}{\tau_R}}\right]\arctan\left(\frac{\pi N}{T_R(\alpha + \tau_R^{-1})}\right), \tag{3.26}$$

where $N$ is the number of neurons and $\alpha$ again denotes the covariance length. This expression allows us to find an optimal solution for $\tau_R$ when no noise is present (either numerically or graphically, since this is a transcendental equation for $\tau_R$), which is $\tau_R \approx 0.3T_R$. It also clearly expresses the relation between $M_q$, $N$ and $T_R$, which is depicted in Figure 3.12b and 3.12c. Basically, memory quality has to be sacrificed to increase $T_R$, as one would intuitively expect.

- Equation 3.26 also allows us to find a good approximation of $M$. We find that

$$M = \frac{2}{\pi}T_R \arctan\left(\frac{\pi N}{T_R(\alpha + \tau_R^{-1})}\right).$$

This equation allows us to analytically check the situation where $\tau_R \to \infty$. Appendix A.2.4.3 provides the analysis, and indeed this confirms the finding that

$$\lim_{\tau_R \to \infty} M = \frac{2N}{\alpha}.$$

## 3.4 Shortcomings, non-linear extensions and future work

From the results I have presented in this chapter so far, one might be lead to conclude that the optimal choice for an ESN would be one with an orthogonal connection matrix. Only one study I am aware of has considered using a highly specific kind of orthogonal network (a permutation matrix) (Boedecker et al., 2009), and compared its performance to random networks. I have never come across strong empirical proof that for real world tasks orthogonal networks score any better than random networks. Even though the connection between orthogonality and LMC remains interesting from a theoretical point of view, its applicability seems limited at the time.

The single greatest drawback in studying linear memory capacity is the fact that it ignores non-linear transformations of the input signal. If the information of $n$ time frames ago is sufficiently distorted via a non-linearity, linear memory capacity considers this information irretrievably lost. In reality, we are not interested in reservoirs with good linear memory. If that's all we want, we will make a delay line or something similar. What we require from the reservoir are non-linear transformations such that useful features of the recent input history are produced. What we wish to understand from this in terms of memory is how long the input history is that the hidden state *depends* on, and how we would be able to control this memory depth.

A better understanding of memory in RNNs can greatly aid to understand how we can build recurrent systems with the right properties for the task we need to solve. One ML training algorithm that can possibly benefit from such a framework is backpropagation through time, where recurrent weights are trained explicitly. One of the great challenges in this domain is to find a way to make sure a network retains relevant information long enough, while simultaneously making sure that the network remains stable. A good understanding of memory can aid in the construction of network models which are particularly good at this (for instance the framework of *long short term memory networks* (Hochreiter and Schmidhuber, 1997)).

In this section I will briefly discuss two methods that try to offer an alternative look on studying reservoir memory. The first one is a generalization of linear memory capacity that essentially not only reconstructs the signal itself, but also all possible non-linear transformations of it. The second one I will suggest here is a fairly simple measure that only looks at the *dependency* of the current state on past input. It bears no quantitative information such as memory capacity, but does offer an interesting qualitative view on memory of a recurrent neural network.

### 3.4.1 Non-linear memory capacity

A strong and straightforward way in which to include non-linearity in the concept of memory capacity is to measure not just how well it reproduces the input signal, but also non-linear functions of the input signal. This was the theme of the research published in Dambre et al. (2012). The idea starts off similarly to that of LMC; we feed a dynamical system one-dimensional i.i.d. noise (in this case uniformly distributed between $-1$ and 1). Next, we attempt to reconstruct not only the input with delay $\tau$, but also a full set of orthogonal functions of the input history. In the case of i.i.d uniformaly distributed noise, this set of functions would be a set of Legendre polynomials. If we denote the Legendre polynomial of order $d$ as $\mathcal{L}_d(x)$, each of this set of orthogonal functions can be written as:

$$f_{\mathbf{d}} = \prod_{i=0}^{\infty} \mathcal{L}_{d_i}(s(t-i)).$$

Here, $\mathbf{d}$ is an infinitely long vector with elements $d_i$, the respective degrees of the polynomials in the product. Note that the Legendre polynomial of order zero is equal to 1, and that of order one is the identity function. This means that if we linearly wish to retrieve the input of delay $\tau$, we will choose a function with $d_i = \delta_{i\tau}$.
Obviously, there are an infinite number of infinitely long sequences $\mathbf{d}$, and measuring scores for each of these seems impractical. However, it is possible to make some assumptions which yield a natural way to select the sequences which carry the bulk of the non-linear memory capacity. First of all, it is reasonable to assume that we only need to consider a finite history. If the dynamical system has fading memory, functions which depend on input frames that lie very far in the past will have very low scores, and can be considered negligible. Therefore it is possible to truncate the search of the individual scores. Next, we can for instance group the remaining sequences by total degree $D = \sum_{i=0}^{\infty} d_i$, which allows us to measure the total memory capacity for each degree. That of the first degree corresponds to the LMC, that of the second order to all quadratic functions of the input, etc.
Finally we end up with a set of memory capacities for each degree. Note that for each degree we need to compute an increasing set of scores to account for all crossproducts of Legendre polynomials. Normally however, the memory capacities tail off when their degree increases, which means that it suffices to compute a limited number. The total memory capacity of the system is then defined as the sum of all memory capacities for all degrees.
It can be proven that the total memory capacity is smaller than or equal to $N$, the dimensionality of the internal state of the system, which places

a fundamental bound on the computational power of *any* dynamical system. Furthermore, measuring the memory capacities of the different degrees can give an indication of which kind of functions the DS actually computes. For example, in Dambre et al. (2012) capacities were measured for standard ESNs without input bias, and they found that only memory capacities of odd degrees were non-zero. This instantly reflects the fact that an ESN without input bias can only compute antisymmetric functions of its input.
The possible applications of non-linear memory capacity have not yet been explored. One potential use (currently being investigated by Joni Dambre, a member of my lab) would be to apply the framework directly on the input and output of a task. If it is possible to define an orthogonal basis of functions on the spatiotemporal distribution of the input signal, it is possible to measure the score on how well the output of the task can reproduce these functions. This would reveal the underlying functional connection between input history and output, and show what kind of non-linear transformation is required to solve the task. This problem is quite similar to non-linear system modeling (Wiener, 1958)
Non-linear memory capacity still has certain downsides that may limit its use in practical applications. First of all there is the problem that for high degrees of non-linearity, memory capacity becomes computationally demanding to calculate accurately, limiting how well we can measure it for very non-linear systems. Secondly, the set of non-linear transformations that the DS computes does not only depend on the DS, but also on the input signal, as temporal correlations within it will influence the dynamics of the system. There is no easy way to generalize the framework to any kind of input signal. If we consider real-world data such as, e.g., speech, it can have a very specific, and potentially very complicated, spatiotemporal distribution, and defining an orthogonal basis of functions on such a signal may be far from trivial.

### 3.4.2 The norm of the Jacobian: instant view on memory.

Here I will pursue a completely different view on 'memory', and instead of looking at how well we are able to *retrieve* input, I shall attempt to gauge *sensitivity* of the hidden state on past input frames. I shall do so by looking at the Jacobian of the hidden state w.r.t. the input signal of $\tau$ steps ago. We can take its norm to describe in one single number how much the hidden state would change for an infinitesimal change in input from $\tau$ steps ago. This approach has two important advantages. First, we can have an *instantaneous* look at the memory window of the reservoir. After all, the Jacobian is defined at each moment in time, which means that at each moment in time we can gauge the current hidden state's sensitivity on changes in past input. The second advantage is that the degree of non-linearity can be interpreted in a wholly new manner. Linear ESNs will have a fixed Jacobian, i.e., it does not depend on time. As the network starts to act more non-linearly, the Jacobian will show temporal variations. The more non-linear the system, the stronger these will be, which offers us a new measure for the non-linearity of an ESN. The great disadvantage of this method is that the norm of the Jacobian has no easy quantitative interpretation, whereas the memory function can be interpreted as a correlation coefficient.

#### 3.4.2.1 Definitions

The Jacobian of the hidden state w.r.t. the input signal at $\tau$ time steps ago is defined as

$$\mathbf{J_a}(t,\tau) = \frac{\partial \mathbf{a}(t)}{\partial \mathbf{s}(t-\tau)}. \tag{3.27}$$

This is a matrix of size $N \times N_{in}$. For an ESN with leak rate (as defined by equation 2.4), the Jacobian can be recursively calculated by

$$\mathbf{J_a}(t,\tau) = \left((1-\gamma)\mathbf{I} + \gamma \mathbf{D}(t-\tau)\mathbf{W}\right)\mathbf{J}(t,\tau-1), \tag{3.28}$$

where

$$\mathbf{J_a}(t,0) = \mathbf{D}(t)\mathbf{V}.$$

With the Jacobian we can e.g. select a single neuron in a network and see at each instance in time how the current state depends on the present input history. If we are interested in the global dependency of the hidden state on the input history, we can look at the Frobenius norm of the Jacobian, i.e. the global rate of change for the hidden state for any change in the input signal:

$$S_{\mathbf{a}}(t,\tau) = ||\mathbf{J_a}(t,\tau)||_F, \tag{3.29}$$

where I use $S$ for *sensitivity*. When interpreting equation 3.28, we see that all dependency on $t$ is present in the diagonal matrices $\mathbf{D}(t-\tau)$. When the network is quasi-linear, these matrices are approximately equal to the identity matrix, which means that they will not depend on time. Vice versa, when the hidden states are pushed into their non-linearities, and are quite variable, $\mathbf{D}(t-\tau)$ will subsequently be quite variable itself, and the dependency on $t$ will be rather strong. This offers us an intriguing perspective on nonlinearity. Non-linear dynamical systems are able to have a variable sensitivity to input signals, whereas for linear systems this sensitivity is fixed.

#### 3.4.2.2 Visualizing non-linearity

In Figure 3.13, I show the effect on non-linearity on $S_{\mathbf{a}}(t, \tau)$. I used an ESN with 100 nodes. The input was one-dimensional Gaussian noise and the leak rate was equal to one. The exact parameters used for the experiment are described in the captions of the figures. Our previous conclusion seems correct: the more strongly non-linear, the more the sensitivity window fluctuates in time. Furthermore it appears that in very non-linear networks (bottom panel in Figure 3.13), the hidden state seems to undergo abrupt cut-off moments, where the sensitivity for past input suddenly drops significantly. This might imply that such networks switch between temporarily 'unstable' regimes, in which they retain memory on a growing length of history, but are abruptly quenched in activity at irregular intervals.
It would be interesting to understand what causes these sudden cut-off moments. I have found that they can be linked with the spectral radius of the Jacobian of the hidden states w.r.t the previous hidden state, i.e., the Jacobian of the linearized system. Indeed, it seems that this spectral radius is slightly higher than one, most of the time, indicating temporarily divergent behavior, and during the cut-off moments it suddenly drops significantly below one, contracting the dynamics of the system. If we could assert control over these cut-offs, e.g., via trained feedback connections, we can in principle assert control over which historic information is present in the network.
In Figure 3.14 I show how a single neuron's Jacobian evolves in time. Again, we see that the effect of non-linearity is reflected in how much the shape of the Jacobian changes in time. The cut-off moments for very non-linear ESNs are also apparent.
If we look at the Jacobian of a single neuron for a linear system, the shape of this Jacobian is constant, and in fact it is the shape of the filter this neuron applies on the input history. In the non-linear case this interpretation is unfortunately not so straightforward. Indeed, the Jacobian is a filter corresponding to the local linearization of the function applied on the input, but it does not take into account the (variable) offset of the linearization of the hyperbolic tangents. Still, this opens the way to an alternative interpretation of reservoir functionality. Indeed, we can consider each neuron as a filter with a variable shape and offset, acting on the input history. If we train an output, we optimally combine these filters into a new one, which approximates the filter that underlies the task.
It is interesting to speculate on the possibilities this offers to visualize the input-output mapping a reservoir computes. We can train a reservoir to solve a task and next visualize the Jacobian of the output w.r.t. the input signal. Perhaps this can reveal functional dependencies that are not obvious to the experimenter, but essential to solving the task.

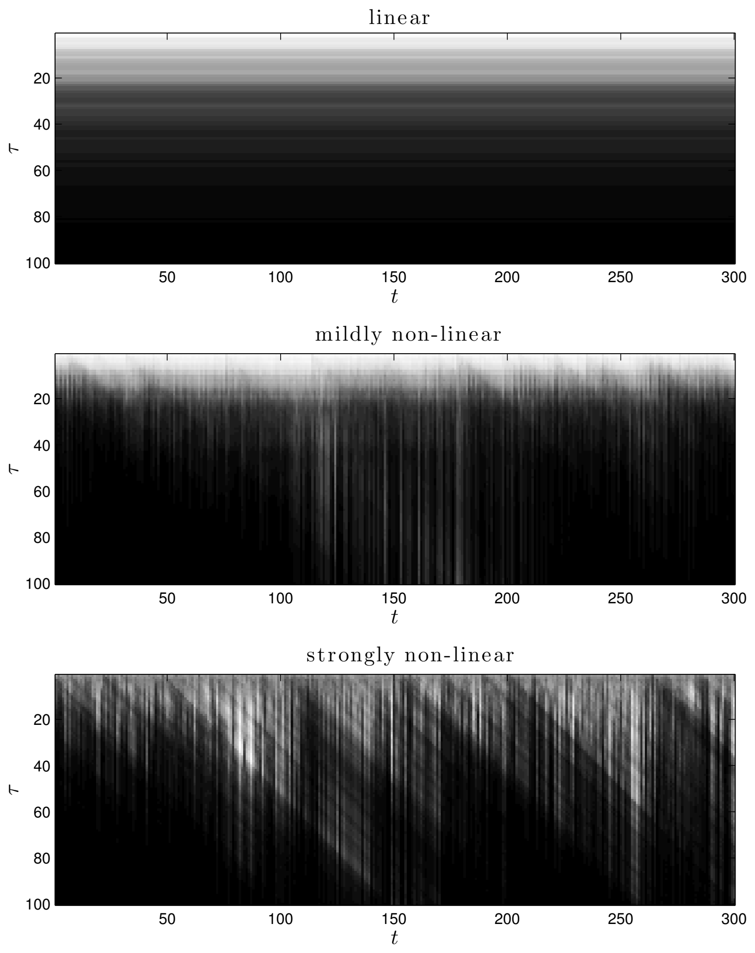

**Figure 3.13:** Example of sensitivity as given in equation 3.29 as a function of $t$ and $\tau$. Each vertical line in this plot shows the current sensitivity of the neuron on the input history. The upper panel is in the quasi-linear regime, with $\rho = 0.99$ and $\zeta = 0.001$. The middle panel is for a mildly non-linear ESN($\rho = 1.05$ and $\zeta = 0.1$), and the bottom panel is for a highly non-linear ESN ($\rho = 1.5$ and $\zeta = 0.5$).

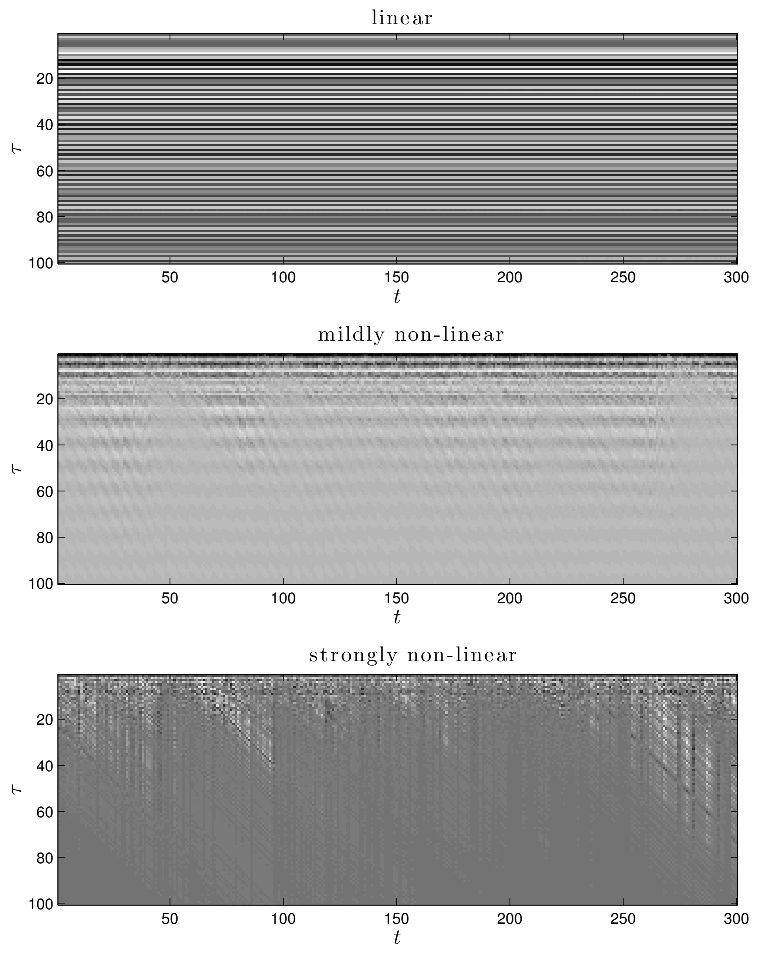

**Figure 3.14:** Example of the Jacobian of a single neuron within a network as a function of $t$ and $\tau$. Each vertical line in this plot shows the Jacobian of the neuron w.r.t. the input history. The upper panel is in the quasi-linear regime, with $\rho = 0.99$ and $\zeta = 0.001$. The middle panel is for a mildly non-linear ESN($\rho = 1.05$ and $\zeta = 0.1$), and the bottom panel is for a highly non-linear ESN ($\rho = 1.5$ and $\zeta = 0.5$).

#### 3.4.2.3 Alternative measures

We can think of other potential methods to measure 'dependence' in ESNs. One promising direction would be to use the framework of information theory. For instance, the dependency between two variables can be expressed in terms of mutual information. In our case we would be interested in the mutual information between the delayed input signal and the hidden state. Mutual information does not change after a non-linear transformation, so in principle it can be a powerful tool for memory visualization. Indeed, it is often used in the field of data visualization (Furuya and Itoh, 2009; Xu et al., 2010). Unfortunately, measuring mutual information requires very large amounts of data, especially for high-dimensional variables such as a hidden state. An accurate measure of this sort would be very difficult. Furthermore, as mutual information is a statistical property, it does not have the same instantaneity as the Jacobian, and we can only measure averages over large amounts of data.
Other information-theoretical concepts such as Fischer information have been applied to model the memory of dynamical systems (Ganguli et al., 2008). However, the authors primarily considered linear systems, which are already well analyzed and understood.

## 3.5 Conclusions

This chapter was devoted to the study of linear memory capacity of Echo State Networks. My two main contributions to this field are the generalization of LMC to the case of multidimensional input signals, and the case of continuous time systems. Through empirical and theoretical research I found that some of the well-known results of previous research can be readily extended to these two scenarios, namely that the LMC is fundamentally restricted by the number of hidden nodes of the network. Furthermore, in the case of continuous time systems, LMC is linked to the autocovariance function of the input signal and to the time scale of the network.
The conclusion that orthogonal networks are much more robust against noise seems to hold. In the case of high-dimensional input signals, random networks allocate a disproportionally large fraction of their available memory capacity to signal components with low power. This distribution is far better in orthogonal networks, where memory capacity is proportional to the power of each component.
In continuous time, an equivalent to orthogonal networks can be designed that is particularly robust against noise. These networks are tunable in the sense that they can concentrate all their memory capacity within a certain

tunable range, called the reservoir period.
Finally I elaborate on non-linear generalizations of the linear memory framework. I briefly discuss results that have been made recently in this field, and i suggest a method for memory visualization, based of the Frobenius norm of the Jacobian.

# 4 Infinite Reservoirs: Recurrent Kernels

This chapter will concern a different line of research. Here I will unite the concept of reservoir computing with that of kernel machines. I will show that it is possible to define infinitely large reservoirs, characterized by a few meta-parameters, and subsequently find a way to actually employ them on tasks. In order to be able to deal with an infinitely large hidden state, I use the so-called *kernel trick*, which means that we end up with a kernel function that is the inner product of the hidden states of infinite networks. These kernels can be readily analyzed in terms of dynamical stability.
I've structured the chapter as follows: first of all I will explain the basics of kernel machines, starting from a simple linear regression example. Next, I introduce recurrent kernels, which use a type of recursion which is inspired on that of RNNs. I show that a broad subset of existing kernel functions can be readily extended to recurrent equivalents.
Once I have formally defined recurrent kernels, I discuss a straightforward framework in which feedforward neural networks can be made infinitely large, leading to a kernel function expression. Next I apply the framework of recurrent kernels on these, and argue that such kernels are equivalent to infinitely large reservoirs. I provide a set of examples of recurrent kernels, some based on existing kernel functions, some on types of recurrent networks.
Section 4.5 addresses the stability of the dynamics of recurrent kernels. I derive stability criteria for different examples. I'll also look into the meaning of spectral radius for infinite sigmoid networks, and find a way to derive the Lyapunov exponent from the recurrent kernel function.
Finally, I looked at task performance. I tested recurrent kernel machines on both an academic task and a real-world speech recognition task, reaching performances close to the state of the art.
A lot of inspiration for the work in this chapter comes from Cho and Saul (2010), where it was found that it is possible to *stack* kernels, and that this

is the equivalent of multi-layered infinite neural networks. As recurrent networks can be considered feed-forward networks with an indefinite number of layers, building upon this work was straightforward.
Most of the results shown in this chapter are published in Hermans and Schrauwen (2011), where I first explain the idea of recurrent kernels, and Hermans and Schrauwen (2012), which deals with sparse threshold unit networks (a specific case of recurrent network).

## 4.1 Kernel machines

In the introduction chapter I already gave an intuitive explanation of what kernel machines entail. Here I start from a theoretical point of view. First of all we shall only consider static data. This means that instead of a signal that changes in time $\mathbf{s}(t)$, we consider static data points, (for example images, properties of a house, etc...) which we denote by $\mathbf{s}_i$, with $i \in \{1, \cdots, N\}$, $N$ being the number of data points. Similar to our explanation in section 2.2, we now associate a feature vector $\mathbf{x}_i$ to each $\mathbf{s}_i$, which is a set of functions of $\mathbf{s}_i$. The difference between this case and the case of reservoirs is solely that here the feature vector only depends on the data entry itself, whereas in the case of reservoir computing, it depends on a fading history of the input signal.

### 4.1.1 Kernel function

Now we may encounter the following problem: suppose the feature vectors $\mathbf{x}_i$ are so high-dimensional that constructing them and performing linear regression or classification on them becomes impractical or even impossible. Take the following example: the feature vector consists of all possible pairwise products of the elements of the input vector. The dimensionality of $\mathbf{x}_i$ in this case is $N_{in}^2$ (For simplicity we ignore the fact that most elements of $\mathbf{x}_i$ will be occur twice). Now suppose we have greyscale pictures as input, a quite common type of data in machine learning. Even for small pictures, the number of pixels, i.e., the input dimension, can easily run into several thousands. This means that the size of $\mathbf{x}_i$ can grow to several millions, far too large to be handled by any existing regression or classification technique. Nevertheless, there is something we can calculate quite easily: the inner products of the feature vectors.
Suppose we denote the $k$-th element of $\mathbf{s}_i$ as $s_k^i$. The elements of the feature vector $\mathbf{x}_i$ will consists of elements $s_k^i s_l^i$, for $k, l \in \{1, \cdots, N_{in}\}$. When we

write down the inner product of two feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, we find:

$$\mathbf{x}_i \cdot \mathbf{x}_j = \sum_{k=1}^{N_{in}} \sum_{l=1}^{N_{in}} s_k^i s_l^i s_k^j s_l^j ,$$

which can easily be rewritten as

$$\mathbf{x}_i \cdot \mathbf{x}_j = \left( \sum_{k=1}^{N_{in}} s_k^i s_k^j \right)^2 = \left(\mathbf{s}_i \cdot \mathbf{s}_j\right)^2 .$$

This expression can be evaluated much faster than it is possible to construct the feature vector explicitly. What I have defined here is in fact the second order homogeneous polynomial kernel function, one of many examples of existing kernel functions.
In many cases the feature vector is even infinite dimensional, and the summation in the inner product changes to an integral. Later we shall see some examples of this.

### 4.1.2 Applying kernels

Now we reach the second part of the derivation. I will show how to apply kernel functions and I will demonstrate this with linear regression. Let's start by contemplating the analytical solution of equation 2.5.

$$\mathbf{U} = \left(\mathbf{X}\mathbf{X}^\mathsf{T}\right)^{-1} \left(\mathbf{X}\mathbf{Y}^\mathsf{T}\right).$$

If we now wish to compute an output value $\mathbf{y}'$ for a new data point $\mathbf{s}'$ with associated feature vector $\mathbf{x}'$, we need to compute $\mathbf{U}^\mathsf{T}\mathbf{x}'$. Writing this out explicitly we get

$$\begin{aligned}\mathbf{y}' &= \left(\left(\mathbf{X}\mathbf{X}^\mathsf{T}\right)^{-1} \left(\mathbf{X}\mathbf{Y}^\mathsf{T}\right)\right)^\mathsf{T} \mathbf{x}' \\ &= \mathbf{Y}\mathbf{X}^\mathsf{T} \left(\mathbf{X}\mathbf{X}^\mathsf{T}\right)^{-1} \mathbf{x}'.\end{aligned}$$

The inverse can be written as $\left(\mathbf{X}\mathbf{X}^\mathsf{T}\right)^{-1} = \mathbf{X}^{\mathsf{T}^{-1}}\mathbf{X}^{-1}$, where the inverses are Moore-Penrose pseudo-inverses, which can be applied to non-square matrices. Furthermore we insert $\mathbf{X}^{\mathsf{T}^{-1}}\mathbf{X}^\mathsf{T} = \mathbf{I}$, the unity matrix, directly left of $\mathbf{x}'$:

$$\begin{aligned}\mathbf{y}' &= \mathbf{Y}\mathbf{X}^\mathsf{T}\mathbf{X}^{\mathsf{T}^{-1}}\mathbf{X}^{-1}\mathbf{X}^{\mathsf{T}^{-1}}\mathbf{X}^\mathsf{T}\mathbf{x}' \\ &= \mathbf{Y}\left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{x}'.\end{aligned}$$

Here, all elements of $\mathbf{X}^\mathsf{T}\mathbf{X}$ are inner products of the feature vectors, and the same holds for $\mathbf{X}^\mathsf{T}\mathbf{x}'$. This means that, if we have access to a kernel function that provides us with the inner products of the feature vectors, we can easily perform linear regression *without* ever needing to explicitly calculate the features.
The matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ is known as the *Gram-matrix*, which we denote with $\mathbf{K}$. Its elements are given by the kernel function operating on couples of the training data points: $\mathbf{K}_{ij} = k(\mathbf{s}_i, \mathbf{s}_j)$, where $k$ is the kernel function. Whereas the matrix $\mathbf{X}\mathbf{X}^\mathsf{T}$ has the number of features as dimension, the Gram-matrix has a size equal to the number of training data points.
If we define output weights $\boldsymbol{\Upsilon}$ as being the solution to the system of equations

$$\mathbf{K}\boldsymbol{\Upsilon} = \mathbf{Y}^\mathsf{T}, \tag{4.1}$$

we can rewrite the solution for $\mathbf{y}'$ as

$$\mathbf{y}' = \boldsymbol{\Upsilon}^\mathsf{T}\mathbf{r}, \tag{4.2}$$

where the elements of column vector $\mathbf{r}$ are the kernel functions: $r_i = k(\mathbf{s}_i, \mathbf{s}')$. Throughout this chapter, I will call the training data points $\mathbf{s}_i$ the *support vectors*. I should mention here that this nomenclature deviates somewhat from the strict sense in which this term is more often used (the context of SVMs), where the support vectors span the classification boundary in feature space. I use it in a broader sense, namely the data points that are eventually included in the trained model.

### 4.1.3 Least-squares support vector machines

What I have defined above is just one of the many ways to apply kernels. I have taken standard linear regression, and transformed it to work with kernels. In fact, many standard regression and classification techniques can be '*kernelized*', i.e. transformed such that they no longer operate on feature vectors, but rather include training data in the end solution and as output provide a linear combination of kernel functions. An exhaustive oversight of the different ways to obtain kernel machines is beyond the scope of this thesis, and I refer to, e.g., Shawe-Taylor and Cristianini (2004). Here I restrict myself to use the so-called *Least-Squares Support Vector Machine* (LS-SVM) (Suykens et al., 2002). This is essentially the kernel machine version of ridge regression as discussed in section 2.2.3, where the derivation is solved starting from a feature space formulation of linear regression. This has the advantage that, once we arrive at infinite reservoirs, we actually use the same cost function as we did with standard ESNs.

An LS-SVM for regression operates as follows. Given a training set of $N$ input features $\mathbf{s}_i$ with corresponding output targets $\mathbf{y}_i$, the system outputs a value $\mathbf{y}(\mathbf{s})$ for an input vector $\mathbf{s}'$ defined as

$$\mathbf{y}(\mathbf{s}') = \boldsymbol{\Upsilon}^{\mathsf{T}}\mathbf{r} + \boldsymbol{\beta}, \tag{4.3}$$

with $r_i = k(\mathbf{s}_i, \mathbf{s}')$, and where the weights $\boldsymbol{\Upsilon}$ and $\boldsymbol{\beta}$ are found by solving the system

$$\begin{bmatrix} 0 & \mathbf{1}_N^{\mathsf{T}} \\ \mathbf{1}_N & \mathbf{K} + \lambda\mathbf{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta} \\ \boldsymbol{\Upsilon} \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{Y} \end{bmatrix}, \tag{4.4}$$

with $\mathbf{1}_N = [1; \cdots; 1]$. The parameter $\lambda$ serves the same purpose as the ridge regression parameter as explained in chapter 2: keeping the weights of $\Upsilon$ limited in magnitude. Also note that the output bias weights I discussed in section 2.2 were explicitly included in the output weights, but here they are given by $\boldsymbol{\beta}$.

### 4.1.4 Selecting support vectors

The limitation of LS-SVMs, and kernel machines in general is that they are difficult to apply on large datasets. Suppose we would take an LS-SVM and directly apply it on, lets say, a training sequence of a million data points, which is relatively small for real-world tasks. We would need to construct a Gram-matrix of a million by a million elements and next invert it. Far out of reach for any conventional computer architecture. Even if we would succeed in determining the output weights, evaluating the model on new data would require the computation of one million kernel values for each data point.

For this reason, kernel machines generally employ various schemes to make the end solution sparse, i.e., to select a small subset of support vectors from the training dataset. The best-known version of kernel machines, an SVM, will use the maximal margin criterion to find this subset. The downside is that, even though this subset has a unique solution, finding it will require the construction of the full Gram-matrix, which size is equal to the number of training data points, still making it impractical for truly large datasets.

For truly large datasets, other selection algorithms are employed. One often used variation of LS-SVM that tackles this problem is the *fixed-size* LS-SVM (Espinoza et al., 2006). Here, support vectors are sampled using a criterion based on Rényi-entropy (Rényi, 1961).

At the end of this chapter, when applying recurrent kernels on ML applications, I will use a large speech data corpus, consisting of well over one million frames. To solve this, I use a technique based on Newton optimiza-

tion (Chapelle, 2007). As explaining the algorithm in detail would go beyond the scope of this dissertation, I will explain it only briefly and refer to Botton et al. (2007) (chapter 2) for specific details.
This method uses a loss function of the form $\max(0, 1 - y_i\tilde{y}_i)^2$, with $y_i$ the target ($-1$ or 1), and $\tilde{y}_i$ the output of the SVM. Essentially this is a quadratic hinge loss function. If we optimize this system using the Newton-Raphson method, this comes down to solving the problem using a quadratic loss-function, i.e., training an LS-SVM, and next selecting data points with $y_i\tilde{y}_i < 1$ as support vectors. This means that we sample the support vectors that either lie in the margin of the classifier, or beyond the margin on the wrong side. We can then iteratively solve the system with different sets of support vectors, until this set no longer changes. Next we can increase the data set and use the previously found set of support vectors as an initial solution to the search algorithm. As such, we can include more and more training data, without ever needing to construct the full Gram-matrix.

## 4.2 Recurrent kernels

Here I will explain the framework in which I define recurrent kernels. We begin by considering the previously defined feature vectors $\mathbf{x}$ more generally. Suppose we have data points $\mathbf{s}_1$ and $\mathbf{s}_2$. For a given kernel function $k(\mathbf{s}_1, \mathbf{s}_2)$, there exists a feature map $\mathbf{F}$ such that

$$\mathbf{x}_1 = \mathbf{F}(\mathbf{s}_1),$$

$$\mathbf{x}_2 = \mathbf{F}(\mathbf{s}_2),$$

$$k(\mathbf{s}_1, \mathbf{s}_2) = \mathbf{x}_1 \cdot \mathbf{x}_2.$$

The output dimension of $\mathbf{F}$ can be infinitely large in some cases. Importantly we will assume that $\mathbf{F}$ is well-defined for any dimensionality of its input argument. We now wish to introduce recursion into the feature map, and in order to do this we will take a more detailed look at how exactly recursion works in RNNs. Next we can extend this line of thought to kernel functions. Each step in time, a recurrent neural network operates on the current input frame and its hidden state to compute a new hidden state. Before I start with the explanation, it is useful to look at the top of Figure 4.1. Here I have rendered the progression of the hidden state of a recurrent neural network, folded out in time. Indeed, it is possible to consider recurrent networks as deep feed-forward networks with as many layers as there are frames in the time series. Each layer gets input from the previous hidden state and the current input frame. From this perspective it is clear that each step in time

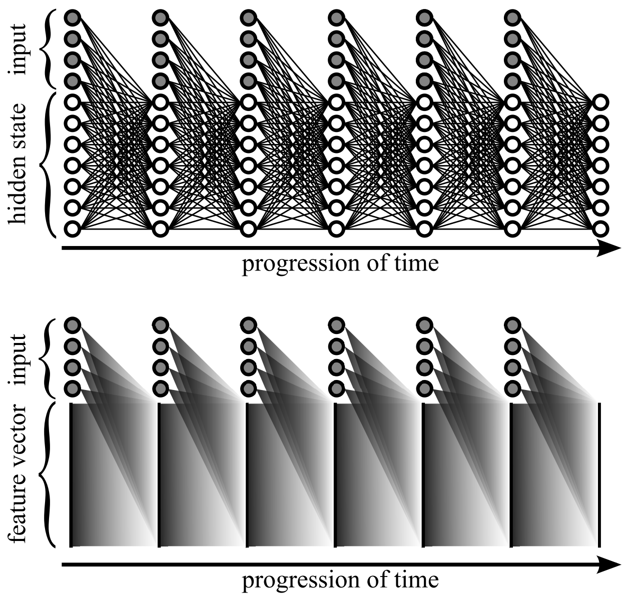

**Figure 4.1:** Top: schematic display of an RNN folded out in time. Bottom: representation of the recursion of the feature vector of a recurrent kernel. Here, the projection forward in time is replaced by an operation $\mathbf{F}$, defined by a kernel function $k$.

is simply a single feed-forward projection, where the full input vector is the concatenation of an input frame and a hidden state. Indeed, we can rewrite the progression of a recurrent neural network as follows[1]:

$$\mathbf{a}(t+1) = f(\mathbf{Wa}(t) + \mathbf{Vs}(t+1)) = f\left([\mathbf{W}|\mathbf{V}]\begin{bmatrix}\mathbf{a}(t)\\ \mathbf{s}(t+1)\end{bmatrix}\right). \tag{4.5}$$

This means that an update of a recurrent network hidden state is essentially *an operation on the concatenation of* $\mathbf{a}(t)$ *and* $\mathbf{s}(t+1)$.

In order to find an equivalent form of recursion for kernel functions, we will associate this operation[2] with the feature map $\mathbf{F}$. The hidden state we will associate with a feature vector $\mathbf{x}(t)$, of the same dimensionality as the output of $\mathbf{F}$, and instead of static data points for input, we will now consider

[1]This way of thinking of recurrent neural networks can probably be attributed to Elman (Elman, 1990)

[2]In RNNs this operation is a linear transformation followed by a hyperbolic tangent

time series $\mathbf{s}_1(t)$ and $\mathbf{s}_2(t)$. We can then write an update formula for the associated feature vectors as[3]:

$$\mathbf{x}_1(t+1) = \mathbf{F}\left(\begin{bmatrix} \mathbf{x}_1(t) \\ \mathbf{s}_1(t+1) \end{bmatrix}\right), \tag{4.6}$$

$$\mathbf{x}_2(t+1) = \mathbf{F}\left(\begin{bmatrix} \mathbf{x}_2(t) \\ \mathbf{s}_2(t+1) \end{bmatrix}\right). \tag{4.7}$$

I have depicted this in the bottom part of Figure 4.1. We can now define a *recurrent kernel* as the inner product of the associated feature vectors at time $t$ for the first and $t'$ for the second. We write this kernel as

$$\kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2) = \mathbf{x}_1(t) \cdot \mathbf{x}_2(t'), \tag{4.8}$$

where $\mathbf{s}_1$ and $\mathbf{s}_2$ are the two input time series, and I use the symbol $\kappa$ instead of $k$ to discern recurrent kernels from those operating on static data points. Note that the true arguments of this function are the complete input time-series up to times $t$ and $t'$, and I use this abbreviated notation to keep the mathematical formulations throughout this chapter readable. Often, we shall consider scenarios where $t = t'$, or the distinction does not matter, and then we simply write

$$\kappa_t(\mathbf{s}_1, \mathbf{s}_2). \tag{4.9}$$

Now I will show that for a broad family of existing static kernel functions a recurrent equivalent can be readily defined. More precisely, let us consider kernel functions which have the following property:

$$k(\mathbf{s}_1, \mathbf{s}_2) = \mathbf{F}(\mathbf{s}_1) \cdot \mathbf{F}(\mathbf{s}_2) = r(\mathbf{s}_1 \cdot \mathbf{s}_2\ ,\ \mathbf{s}_1 \cdot \mathbf{s}_1\ ,\ \mathbf{s}_2 \cdot \mathbf{s}_2), \tag{4.10}$$

i.e., the kernel function is a function $r$ of the quadratic norms of the input arguments and their inner product. Several commonly used kernel functions, including the popular Gaussian RBF kernel, fall into this category. As is written in equation 4.7, in the case of recurrent kernels, the arguments are the concatenations of previous feature vectors with current input frames. If we state that kernel function $k$ has feature map $\mathbf{F}$ associated with it for

[3]Here an important remark needs to be made. I did not study the validity of concatenating a finite-dimensional vector with an infinite-dimensional one. Even though the concept of an inner product remains well defined in this case, which is what we need in the end, the operator $\mathbf{F}$ itself may become ill-defined, as it needs to operate on an infinite-dimensional vector. This problem is avoided due to the fact that we do not require an expression for $\mathbf{F}$ if the underlying kernel function is known. Nevertheless, the framework of recurrent kernels might greatly benefit from some improved mathematical rigor.

which equation 4.10 is valid, this means that this same feature map can be applied for recurrent kernels, where the arguments of the function $r$ need to be replaced by inner products between these concatenations of input frames and feature vectors. We can write:

$$\begin{aligned}\kappa_{t+1,t'+1}(\mathbf{s}_1,\mathbf{s}_2) &= \mathbf{x}_1(t+1)\cdot\mathbf{x}_2(t'+1)\\ &= \mathbf{F}\left(\begin{bmatrix}\mathbf{x}_1(t)\\ \mathbf{s}_1(t+1)\end{bmatrix}\right)\cdot\mathbf{F}\left(\begin{bmatrix}\mathbf{x}_2(t')\\ \mathbf{s}_2(t'+1)\end{bmatrix}\right)\\ &= r\Bigg(\begin{bmatrix}\mathbf{x}_1(t)\\ \mathbf{s}_1(t+1)\end{bmatrix}\cdot\begin{bmatrix}\mathbf{x}_2(t')\\ \mathbf{s}_2(t'+1)\end{bmatrix},\\ &\qquad \begin{bmatrix}\mathbf{x}_1(t)\\ \mathbf{s}_1(t+1)\end{bmatrix}\cdot\begin{bmatrix}\mathbf{x}_1(t)\\ \mathbf{s}_1(t+1)\end{bmatrix},\\ &\qquad \begin{bmatrix}\mathbf{x}_2(t')\\ \mathbf{s}_2(t'+1)\end{bmatrix}\cdot\begin{bmatrix}\mathbf{x}_2(t')\\ \mathbf{s}_2(t'+1)\end{bmatrix}\Bigg).\end{aligned}\tag{4.11}$$

If we write out these inner products, we see that

$$\begin{bmatrix}\mathbf{x}_1(t)\\ \mathbf{s}_1(t+1)\end{bmatrix}\cdot\begin{bmatrix}\mathbf{x}_2(t')\\ \mathbf{s}_2(t'+1)\end{bmatrix} = \mathbf{x}_1(t)\cdot\mathbf{x}_2(t') + \mathbf{s}_1(t+1)\cdot\mathbf{s}_2(t'+1),$$

and similar for the other two arguments. The left term of this equation is the recurrent kernel value $\kappa_{t,t'}(\mathbf{s}_1,\mathbf{s}_2)$, which allows us to rewrite equation 4.11 as follows:

$$\begin{aligned}\kappa_{t+1,t'+1}(\mathbf{s}_1,\mathbf{s}_2) = r\Big(&\kappa_t(\mathbf{s}_1,\mathbf{s}_2) + \mathbf{s}_1(t+1)\cdot\mathbf{s}_2(t'+1),\\ &\kappa_t(\mathbf{s}_1,\mathbf{s}_1) + \mathbf{s}_1(t+1)\cdot\mathbf{s}_1(t+1),\\ &\kappa_{t'}(\mathbf{s}_2,\mathbf{s}_2) + \mathbf{s}_2(t'+1)\cdot\mathbf{s}_2(t'+1)\Big),\end{aligned}\tag{4.12}$$

i.e., a recurrent formula that can be applied for any kernel function of the form specified in equation 4.10. Notice that this recurrent formula also requires us to compute the recurrent kernels $\kappa_t(\mathbf{s}_1,\mathbf{s}_1)$ and $\kappa_{t'}(\mathbf{s}_2,\mathbf{s}_2)$. Often a kernel function is only a function of the inner product of the arguments, and not their quadratic norms. In this case the corresponding recurrent kernel can be simplified to the following form:

$$\kappa_{t+1,t'+1}(\mathbf{s}_1,\mathbf{s}_2) = r(\kappa_{t,t'}(\mathbf{s}_1,\mathbf{s}_2) + \mathbf{s}_1(t+1)\cdot\mathbf{s}_2(t'+1)).\tag{4.13}$$

Before providing a set of important examples of recurrent kernels, I will first introduce the concept of infinite-sized feedforward neural networks, which have equivalent kernel functions. These kernel functions can then be made

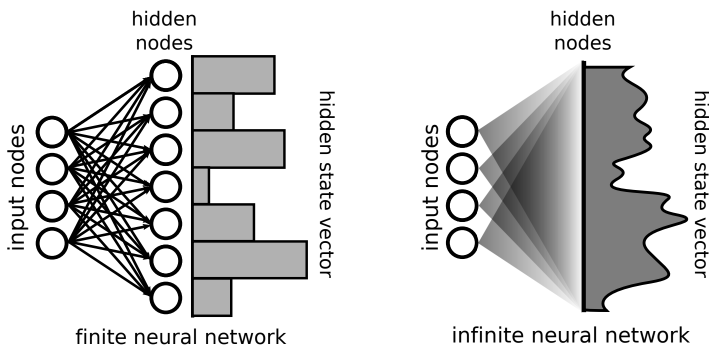

**Figure 4.2:** Schematic display of a finite versus an infinite neural network.

recurrent, resulting in an expression that is associated with infinite-sized recurrent networks.
Kernel machines have been extended to recurrent forms before (Suykens and Vandewalle, 2000), but generally what is adapted is the framework of the kernel machine. The used kernel functions are still the classical static kernels. Instead, I directly start from a recurrent dynamical system and derive a kernel function from that, which is a fundamentally different approach.

## 4.3 Infinite neural networks

I start by taking a look at single-layered feedforward neural networks. Typically, the $i$-th element of the hidden state $\mathbf{a}$ is given by

$$a_i = f(\mathbf{v}_i, \mathbf{s}),$$

with $\mathbf{s}$ the input vector, $\mathbf{v}_i$ the weights associated with the $i$-th neuron, and $f$ the activation function. Two important examples of neurons are sigmoid neurons, for example:

$$a_i = \tanh(\mathbf{v}_i \cdot \mathbf{s}),$$

and Gaussian RBF nodes, where

$$a_i = \exp(-||\mathbf{v}_i - \mathbf{s}||^2/2\sigma^2).$$

We can consider the hidden state of the network as the feature vector of the neural network. Each dimension of this feature vector, i.e., each neuron activation, is uniquely characterized by $\mathbf{v}_i$. The set of weights $\mathbf{V}$ obviously will determine which computations the network actually performs. Suppose we have a fixed number of neurons $N$. For any given task with a set of training data, there exists a set of weights $\mathbf{v}_i$ (not necessarily unique) which minimizes the cost function for the task (together with the output weights). Finding these optimal weights may be very difficult or impractical. Therefore we simply define a network that has *all possible neurons*, and so all possible $\mathbf{v}_i$. Within this network the optimal weights are definitely present[4], and we can in theory solve the task perfectly. I have depicted this idea schematically in Figure 4.2.

As we have no explicit way to compute an infinite-dimensional hidden state, we now define the associated kernel. The inner product of the hidden states for two input vectors $\mathbf{s}_i$ and $\mathbf{s}_j$ is simply

$$k(\mathbf{s}_i, \mathbf{s}_j) = \sum_{\mathbf{v}} f(\mathbf{v}, \mathbf{s}_i) f(\mathbf{v}, \mathbf{s}_j).$$

Since there are an infinite number of terms here, we will replace it by an integration:

$$k(\mathbf{s}_i, \mathbf{s}_j) = \int_{\Omega_{\mathbf{v}}} f(\mathbf{v}, \mathbf{s}_i) f(\mathbf{v}, \mathbf{s}_j) d\mathbf{v},$$

with $\Omega_{\mathbf{v}}$ the space in which the input vectors exist. Very often, (for instance for a sigmoid activation function), the above integral is undefined or diverges, as it covers an infinitely large domain. Therefore we will add another restriction, and we say that $\mathbf{v}$ is drawn from a probability distribution $P(\mathbf{v})$. This leads to the final expression for the infinite neural network kernel:

$$k(\mathbf{s}_i, \mathbf{s}_j) = \int_{\Omega_{\mathbf{v}}} d\mathbf{v} P(\mathbf{v}) f(\mathbf{v}, \mathbf{s}_i) f(\mathbf{v}, \mathbf{s}_j). \tag{4.14}$$

If this integral is analytically tractable, we will find a kernel function which we can plug in into any existing kernel machine.

Using the integral expression of such kernels on recurrent networks is an ill-defined problem, as the argument over which needs to be integrated would become infinite-dimensional. However, if the resulting kernel function for an infinite-sized feedforward network is of the type given by equation 4.10, we can readily make this kernel recurrent. As the line of thought to derive recurrent kernels is directly based on the recursion as it appears in RNNs, we can readily find the recurrent kernel associated with infinite-sized recurrent networks. In some specific cases as, e.g., the sparse threshold unit network, which will be discussed in the next section, we cannot directly apply the

[4]In the case of sigmoid nodes we need to include input bias, i.e. an additional input dimension which is constant and equal to one. This in order to ensure that the network can include non-antisymmetric functions.

framework of recurrent kernels. This is because the underlying feature map does not operate on the input frame and the feature vector in the same manner. In the case of sparse threshold unit networks specifically, however, it is possible to explicitly solve an integral expression that takes the recursion into account directly.

## 4.4 Examples

Now that we have a comprehensible way to produce recurrent kernels, I will list some important examples in this section. I will use the following convention in this chapter: input scaling and bias are *included* in the input time series $\mathbf{s}(t)$; as the expressions for recurrent kernels are often already quite lengthy I don't wish to introduce additional parameters. Therefore, if I solve tasks, I will mention an input scaling factor, i.e., a number with which I multiply the raw input signal. If we wish to include bias, we will use the convention that $\mathbf{s}(t)$ has an additional input dimension with a constant value, but for the tasks in this thesis I never use input bias.

### 4.4.1 Linear and polynomial

The simplest kernel in existence is the linear kernel: $k(\mathbf{s}_i, \mathbf{s}_j) = \alpha \mathbf{s}_i \cdot \mathbf{s}_j$, with $\alpha$ a scaling factor. The recurrent version of this kernel is given by

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s}_2) = \alpha \mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t'+1) + \alpha \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2).$$

We can explicitly write down the recursion in this equation as

$$\kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2) = \sum_{i=0}^{\infty} \alpha^{i+1} \mathbf{s}_1(t-i) \cdot \mathbf{s}_2(t'-i). \qquad (4.15)$$

From this equation it is clear that, if $\alpha > 1$, $\kappa$ will grow exponentially. Therefore we need to choose $\alpha$ between zero and one.
Interestingly, if we consider equation 4.14 and take the linear function where $f(\mathbf{v}, \mathbf{s}_i) = \mathbf{v} \cdot \mathbf{s}_i$, and we take $P(\mathbf{v})$ to be a Gaussian distribution with covariance matrix $\alpha \mathbf{I}$ and mean at the origin, we end up with the above linear kernel. Notice that here we find an immediate link between the *scaling* of the recurrent weights (determined by $P(\mathbf{v})$) and the dynamical stability.
Recurrent polynomial kernels of order $p$ can similarly be defined as

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s}_2) = \left(\alpha \mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t'+1) + \alpha \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2)\right)^p.$$

In practice, the dynamical stability of this kernel can only be guaranteed for $p > 1$ if $\alpha \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2) < 1$ at all time, hence we shall not use it.

### 4.4.2 Gaussian RBF kernel

The Gaussian RBF-kernel is defined as

$$k(\mathbf{s}_i, \mathbf{s}_j) = \exp\left(\frac{-||\mathbf{s}_i - \mathbf{s}_j||^2}{2\sigma^2}\right), \tag{4.16}$$

with a parameter $\sigma$, known as the *kernel width.* Applying the rules of equation 4.12, we find that

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s}_2) = \exp\left(\frac{-||\mathbf{s}_1(t+1) - \mathbf{s}_2(t'+1)||^2}{2\sigma^2} + \frac{\kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2) - 1}{\sigma^2}\right). \tag{4.17}$$

It is not directly clear what exactly defines dynamical stability for this kernel, and later on I will derive a criterion. Here I ask a different question. If we actually reverse our earlier line of thought and try to discover a feedforward network that, if made infinitely large, has the Gaussian RBF as its associated kernel. If we would arrive at such a network, we would be able to make it recurrent and actually produce a reservoir that is the finite-dimensional equivalent of the recurrent Gaussian RBF kernel.
In appendix A.3.1, I will show that it is indeed possible to define such a network. For nodes with activation function

$$f(\mathbf{v}_i, \mathbf{s}) = \frac{\exp(\mathbf{v}_i \cdot \mathbf{s})}{\sqrt{\sum_j \exp(2\mathbf{v}_j \cdot \mathbf{s})}}, \tag{4.18}$$

i.e., if the nodes have an exponential activation function, and the hidden state is normalized, this behavior can be found if the weights are drawn from a Gaussian distribution. Notice the similarity that this equation has with the softmax function (as given in equation 2.7) which does not normalize its activation, but rather divides it by its sum. As I derive in appendix A.3.1, the kernel associated with an infinitely large softmax function can also be derived. The kernel that is associated with an infinitely large softmax layer operating on data points $\mathbf{s}_1$ and $\mathbf{s}_2$ is given by

$$k(\mathbf{s}_1, \mathbf{s}_2) = \frac{1}{\sigma\sqrt{2\pi}} \exp(\sigma^2 \mathbf{s}_1 \cdot \mathbf{s}_2).$$

The recurrent version of this kernel can only be stable under strict conditions, and hence I do not apply it on time series here.
Softmax-like models, i.e., models that assume the total activity of a dynam-

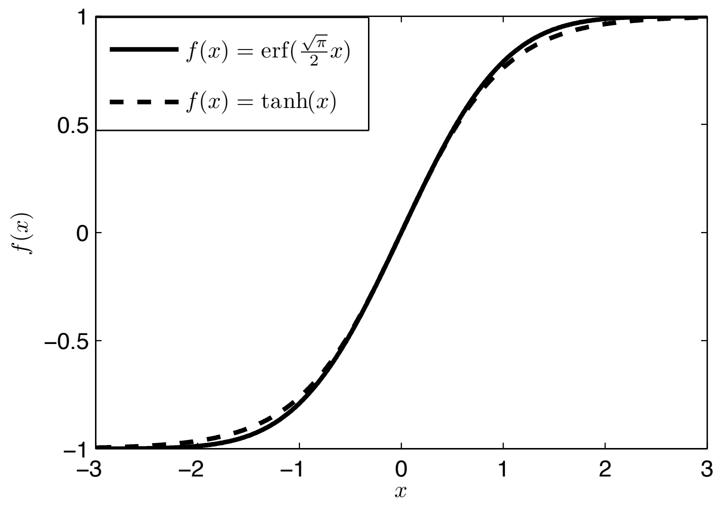

**Figure 4.3:** Shape of the error function compared to that of the hyperbolic tangent. The argument of the error function has been rescaled to match a slope of one around the origin.

ical system is bounded, are interesting models for certain cognitive functions in the brain, and especially the neocortex (Walley and Weiden, 1973; Rutishauser et al., 2012). Recurrent kernels that are based on such bounded DSs can potentially serve as mathematical models that will enable to provide insight in cognitive processes.

### 4.4.3 Sigmoid network kernel

Obviously, what we wish to achieve most is the recurrent kernel equivalent of ESNs, i.e., with a sigmoid activation function. As it turns out, equation 4.14 has no analytical solution for $f(\mathbf{v}, \mathbf{s}) = \tanh(\mathbf{v} \cdot \mathbf{s})$. Nevertheless, in Williams (1998) an analytical solution for the integral was found for the case where the non-linearity is the error function[5]. The error function has a sigmoid shape. The slope near the origin is equal to $2/\sqrt{\pi}$ instead of 1 as in the case of a hyperbolic tangent. Figure 4.3 shows a comparison between the shape of a hyperbolic tangent and the error function. The difference is relatively small, and therefore I believe it is reasonable to assume that RNNs with an error function non-linearity will behave qualitatively similar to ESNs.
I choose the probability distribution $P(\mathbf{v})$ to be a Gaussian distribution with covariance matrix $\sigma^2\mathbf{I}$ and mean at the origin. The equivalent kernel is given

[5]The error function is defined as $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)dt$

by

$$k(\mathbf{s}_i, \mathbf{s}_j) = \frac{2}{\pi} \arcsin \left( \frac{2\sigma^2 \mathbf{s}_i \cdot \mathbf{s}_j}{\sqrt{(1 + 2\sigma^2 \mathbf{s}_i \cdot \mathbf{s}_i)(1 + 2\sigma^2 \mathbf{s}_j \cdot \mathbf{s}_j)}} \right). \tag{4.19}$$

The recurrent version of this kernel is given by

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s_2}) = \frac{2}{\pi} \arcsin \left( \frac{2\sigma^2 \left(\mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t'+1) + \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2)\right)}{\sqrt{g_{t+1}(\mathbf{s}_1) g_{t'+1}(\mathbf{s_2})}} \right), \tag{4.20}$$

with

$$g_{t+1}(\mathbf{s}) = 1 + 2\sigma^2 \left( ||\mathbf{s}(t+1)||^2 + \kappa_t(\mathbf{s}, \mathbf{s}) \right),$$

and

$$\kappa_t(\mathbf{s}, \mathbf{s}) = \frac{2}{\pi} \arcsin \left( 1 - \frac{1}{g_t(\mathbf{s})} \right).$$

I will call this kernel the *recurrent arcsine kernel.* In the next section I will derive the stability criteria for this kernel, and I will show that we can connect $\sigma$ with the concept of spectral radius.

### 4.4.4 Sparse threshold unit networks

The last network model I will mention are sparse threshold unit networks (STUNs)[6]. A threshold unit has a threshold function as activation function:

$$f(x) = \begin{cases} -\theta_n & \text{if } x < 0 \\ \theta_p & \text{if } x \geq 0 \end{cases} \tag{4.21}$$

In the case where $\theta_n = 0$, $\theta_p = 1$ this is a Heaviside function, and these nodes are known as *binary* nodes. If $\theta_n = \theta_p = 1$, we end up with the sign function. Similar to sigmoid nodes, the threshold unit operates on the inner product of the input signal with a weight vector. Solving equation 4.14 for this situation is relatively easy. In appendix A.3.2.1 I derive the equivalent kernel, however, when we make it recurrent, the resulting kernel has an infinitely high Lyapunov exponent, which means that it will never be dynamically stable.
In order to still be able to get a stable recurrent kernel from an infinite recurrent threshold unit network, the recurrent connections need to be sparse. More precisely: we state that each node in the recurrent network has $K$

[6]The idea of defining recurrent kernels for STUNs comes from Stefan Depeweg, an intern working in our lab for a few months. Having forgotten to acknowledge him in my original paper on infinite STUNs, I will set this straight here.

incoming connections, where $K$ is the so-called *in-degree.*
So far, we have always implicitly assumed that our networks were fully connected, which made the transfer from normal kernels to recurrent kernels straightforward. For sparsely connected networks, this line of reason will no longer apply, and we need a more in-depth analysis. Appendix A.3.2.2 offers the full derivation, which is rather lengthy. Here I simply present the result. I will only focus on the two previously mentioned situations: those with $\theta_n = \theta_p = 1$ and with $\theta_n = 0$, $\theta_p = 1$. The first case gives us the following kernel

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s}_2) = \frac{2}{\pi} \sum_{i=0}^{K} (h_+)^i (h_-)^{K-i} \binom{K}{i} H_i, \tag{4.22}$$

where

$$H_i = \arcsin\left( \frac{2i - K + \mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t+1)}{\sqrt{(K + ||\mathbf{s}_1(t+1)||^2)(K + ||\mathbf{s}_2(t+1)||^2)}} \right) \tag{4.23}$$

and

$$h_+ = \frac{1 + \kappa_{t,t'}(\mathbf{s_1}, \mathbf{s_2})}{2}, \quad h_- = \frac{1 - \kappa_{t,t'}(\mathbf{s_1}, \mathbf{s_2})}{2}. \tag{4.24}$$

The second case is slightly more complicated, giving:

$$\kappa_{t+1,t'+1}(\mathbf{s}_1, \mathbf{s}_2) = \frac{1}{2} \sum_{i=0}^{K} \sum_{j=0}^{K-i} (h_1)^{i+j} (h_0)^{K-i-j} \sum_{k=0}^{K-i-j} \binom{K}{i\ j\ k} G_{Kijk}, \tag{4.25}$$

where

$$h_1 = \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2), \quad h_0 = \frac{1}{2} - \kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2), \tag{4.26}$$

$$G_{Kijk} = 1 - \frac{1}{\pi} \arccos\left( \frac{i + \mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t+1)}{\sqrt{i + k + ||\mathbf{s}_1(t+1)||^2} \sqrt{K - j + ||\mathbf{s}_2(t+1)||^2}} \right), \tag{4.27}$$

and we use the shorthand notation

$$\binom{K}{i\ j\ k} = \frac{K!}{i!j!k!(K-i-j-k)!}.$$

The interest in STUNs stems from the fact that they have often been used as mathematical abstractions for a variety of phenomena. First of all there was Kauffman (1969), who studied the dynamics of binary threshold unit networks as a model for the genetic regulatory network in living cells. More recently these networks have been studied in the context of order-chaos transitions as a function of the internal connectivity and input scaling, using a

mean field approximation (Natschläger and Maass (2004)). Here, they serve as an abstraction for liquid state machines, (where the binary node's on-off behavior models spikes).
Extending the above kernel functions to networks with a variable in-degree is straightforward, as I also show in appendix A.3.2.2. Suppose that the probability of a randomly drawn neuron having $K$ incoming recurrent connections is given by $p_K$, and suppose the kernel associated with a network with fixed in-degree $K$ is denoted by $\kappa^K_{t,t'}(\mathbf{s}_1, \mathbf{s}_2)$, the kernel for the variable in-degree network is given by:

$$\kappa_{t,t'}(\mathbf{s}_1, \mathbf{s}_2) = \sum_{K=1}^{\infty} p_K \kappa^K_{t,t'}(\mathbf{s}_1, \mathbf{s}_2). \tag{4.28}$$

This allows us in principle to study the dynamics of a variety of network models. Special interest would go to the so-called *scale-free* networks, which model a large variety of natural and artificial phenomena, such as the internet and social networks (Barabasi and Albert, 1999), and the spread of diseases (Pastor-Satorras and Vespignani, 2001).

## 4.5 Recurrent kernel dynamics

In this section I shall explain how we can investigate the dynamical stability of infinite recurrent networks. The most straightforward method, which I will discuss first is to find an analytical expression for the Lyapunov exponent corresponding with the infinite network. This is only feasible for a certain type of kernel though, and for others we will use different strategies.

### 4.5.1 Lyapunov exponent

The definition of the Lyapunov exponent is given by:

$$\lambda = \lim_{t\to\infty} \lim_{D(0)\to 0} \frac{1}{t} \ln\left(\frac{D(t)}{D(0)}\right), \tag{4.29}$$

where $D(t)$ is the distance between two state vectors at time $t$, $D(0)$ is the initial infinitesimally small distance between two initial conditions for the system. In appendix A.3.3 we find that, for recurrent kernel functions that have the property that $\kappa_t(\mathbf{s}, \mathbf{s})$ is constant (independent of $t$ or the signal), we are able to analytically derive the Lyapunov exponent[7]. If we denote it

[7]Here I need to add an important remark. Commonly the Lyapunov exponent is defined for autonomous systems, i.e., not driven by external input. In the case of

with $\ell$, we find that

$$\ell = \frac{1}{2}\left\langle \ln\left(\frac{\partial\kappa_t(\mathbf{s}_1,\mathbf{s}_2)}{\partial\kappa_{t-1}(\mathbf{s}_1,\mathbf{s}_2)}\right)_{\mathbf{s}_1=\mathbf{s}_2}\right\rangle_t . \tag{4.30}$$

In what follows I apply this formula to both recurrent Gaussian RBF kernels and sparse threshold unit kernels.

#### 4.5.1.1 Recurrent Gaussian kernel

In the case of recurrent Gaussian kernels we yield

$$\frac{\partial\kappa_t(\mathbf{s}_1,\mathbf{s}_2)}{\partial\kappa_{t-1}(\mathbf{s}_1,\mathbf{s}_2)} = \frac{1}{\sigma^2}\kappa_t(\mathbf{s}_1,\mathbf{s}_2).$$

If $\mathbf{s}_1 = \mathbf{s}_2$, and we consider the remark made in paragraph 4.4.2 that the hidden state of the associated recurrent network is normalized, we can write that $\kappa_t(\mathbf{s},\mathbf{s}) = 1$. Applying equation 4.30, we find that

$$\ell = -\ln(\sigma). \tag{4.31}$$

This has two very interesting consequences. First of all, in order to make sure that $\ell < 0$, we shall need to pick $\sigma > 1$. Secondly, the Lyapunov exponent depends in no way on the input signal. This implies that, quite unlike ESNs, we cannot get stable dynamics by increasing the input scaling.

#### 4.5.1.2 Sparse threshold unit networks

The Lyapunov exponent of STUNs can be worked out similarly. The derivation I have left for appendix A.3.3.1, and here I will provide the results. For sign-node networks we find:

$$\ell = \frac{1}{2}\left\langle \ln\left(\frac{K}{\pi}\arccos\left(1-\frac{2}{K+||\mathbf{s}(t)||^2}\right)\right)\right\rangle_t , \tag{4.32}$$

Binary networks yield:

$$\ell = \frac{1}{2}\left\langle \ln\left(\frac{K}{\pi 2^{K-1}}\sum_{i=0}^{K-1}\binom{K-1}{i}\arccos\left(\sqrt{\frac{i+||\mathbf{s}(t)||^2}{i+1+||\mathbf{s}(t)||^2}}\right)\right)\right\rangle_t . \tag{4.33}$$

recurrent kernels this is definitely not true. Instead, we consider the input signal to be *part of the system.* Essentially, the Lyapunov exponent as provided by equation 4.29 includes the input signal distribution as an important system variable. If the input signal distribution changes, so will the Lyapunov exponent.

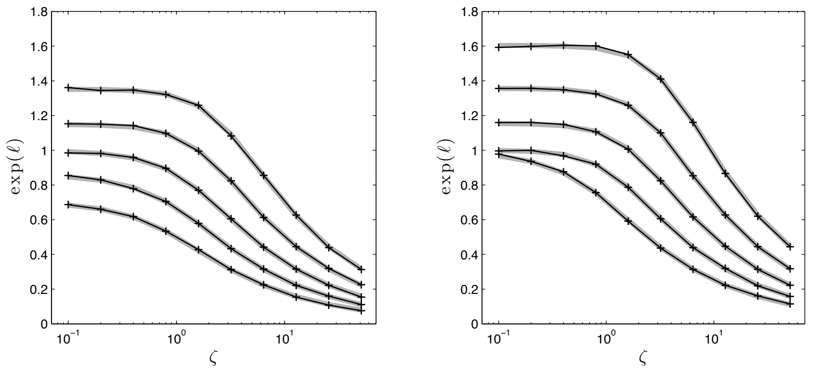

**Figure 4.4:** Lyapunov exponent as a function of input scaling $\zeta$. The left panel is measured for binary-node networks, the right panel for sign-node networks. From bottom to top $K = \{1, 2, 4, 8, 16\}$. The light grey lines are the theoretical predictions as given by equations 4.32 and 4.33, and the black plus signs are the empirical measurements

These results connect the influence of input signal scaling and in-degree. We can provide empirical validation of these expressions by simulating finite STUNs and measure the average Lyapunov exponent. To do so I will make two simplifications. First of all I assume that the input signal has the property that $||\mathbf{s}(t)|| = \zeta$ for all $t$, which avoids the need to compute the average in equations 4.32 and 4.33. Furthermore, we approximate $\exp(\ell)$ by $D(t+1)/D(t)$ where $D(t)$ is the distance between the unperturbed and the perturbed hidden state. The perturbation itself is performed by flipping one randomly chosen node.

I performed the experiment with STUNs of 500 nodes. The input signal is Gaussian noise of dimensionality 50 of which each frame has been normalized and next multiplied with $\zeta$. To get rid of initial conditions, I let the network run for 100 time steps before I applied the perturbation and perform the measurement. I studied $\ell$ for combinations of $K$ and $\zeta$ which I compared to the analytical prediction. Each empirically obtained value is averaged over 2000 network initializations.

Results are shown in Figure 4.4. We see a very good correspondence between the empirical and analytical values. Figure 4.4 also clearly shows the relation between the in-degree, input scaling and the Lyapunov exponent. In all scenario's, $\ell$ decreases as input scaling increases, as the input signal starts to dominate the internal dynamics. Secondly we notice that the higher the in-degree becomes, the stronger the input signal will need to be to obtain a regime in which $\ell < 0$, i.e., with fading memory. We see the difference between sign-node STUNs and binary STUNs. In the case of a very small input magnitude, binary STUNs are always stable as long as $K < 5$, whereas sign-node networks need $K < 3$.

### 4.5.2 Iterated function: cobweb plots

Here I will present another way in which to study recurrent kernel dynamics. A recurrent kernel is essentially an iterated function, where each iteration an outside argument, the input time series, is added. If we suppose that the input signal is constant, we can graphically represent the evolution of the recurrent kernel using a *cobweb diagram.* Consider an iterated function $x_{i+1} = q(x_i)$. We can show the evolution of $x_i$ by plotting the function $q(x)$ and the identity function. Each iteration the new value will be the new argument of the next, and this can be represented by plotting vertical and horizontal strokes between the identity function and $q(x)$, and repeating this process, leading to the cobweb plot.
Cobweb plots can be used to study kernel stability as follows. If the recurrent kernels are dynamically stable, there should exist a single stable fixed point attractor to which all orbits converge, just like in an ESN the hidden state should converge to zero when no input is presented. There is a mathematical theorem called the Banach theorem (Banach, 1922). It states that, given a non-empty, complete metric space $X$ with a distance metric $D$, the contractive mapping $q$ has one and only one fixed point $c^*$ if there exists a number $0 < p < 1$ such that

$$D\big(q(a), q(b)\big) \leq pD(a, b), \tag{4.34}$$

with $a$ and $b$ any two elements from $X$. Put in simple terms, this means the following. Suppose we have an iterated function, and we choose two different starting points $a$ and $b$ as initial values, with a distance $D(a, b)$ between them. In order to have only one fixed point for the iterated function, this distance must decrease for each step in the iteration. Notice the similarity to the definition of the Lyapunov exponent, where the distance needed to be infinitesimal, and we only looked at the long-term evolution of the distance. In Hermans and Schrauwen (2011), I have presented the derivation needed to find the conditions under which Banach theorem holds. Here it suffices to say that, as long as $q(\kappa)$ increases monotonically, we only need to look at its derivative with respect to $\kappa$. If it rises faster than the identity function (see the right panel of the middle row of Figure 4.5), the orbits will diverge, and the dynamics are locally unstable.
I will study the cobweb plots for the three important examples of recurrent kernels: the recurrent Gaussian RBF, the recurrent arcsine kernel and

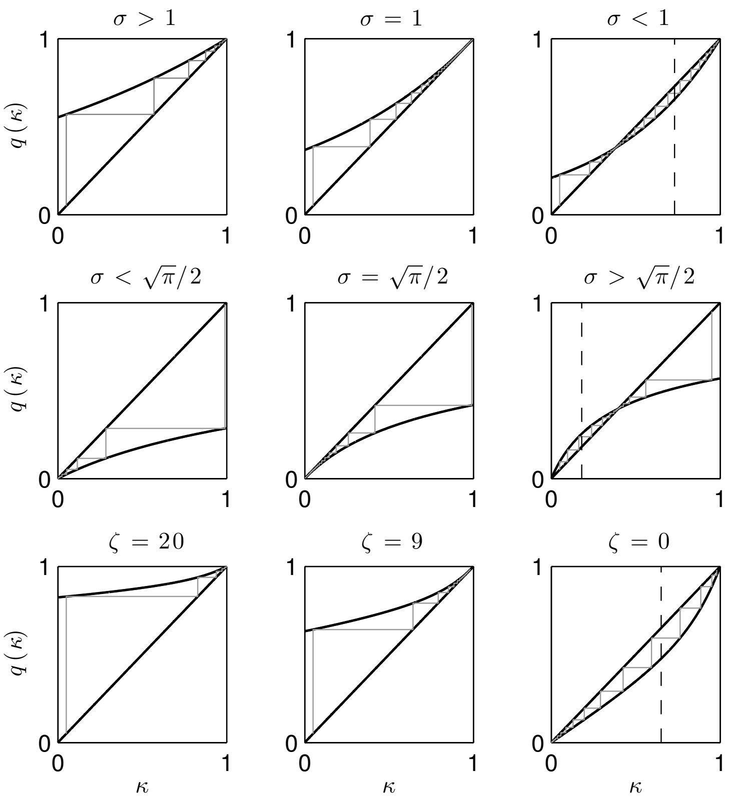

**Figure 4.5:** Cobweb plots, representing the dynamics of recurrent kernels as iterated functions. The top row of panels are for recurrent Gaussian kernels, the middle row for recurrent arcsine kernels, and the bottom row for STUN-kernels for sign nodes and in-degree $K = 16$. Each three panels in the rows represent situations for different parameters, such that the left panel corresponds to asymptotically stable behavior, the middle panel to the edge of stability, and in the right panels the dynamics of the underlying infinite DSs are unstable. The grey lines are example orbits that converge to a stable fixed point. The dashed lines in the right panels show the separation between the stable and unstable region.

the STUN kernel. For the recurrent RBF kernel I'll suppose that the input signals are identical (such that their relative distance is always zero). The recurrent arcsine kernel I will study for input signals equal to zero. In the case of STUNs, the parameter that decides dynamical stability is input scaling. Therefore I will here assume that the input signals are constant and equal to $\zeta$.

In Figure 4.5 I show the resulting cobweb plots for the three kernels, each for three dynamical regimes, associated with different kernel parameters. When we first consider the recurrent RBF kernel (top row of Figure 4.5), we find that we can confirm the earlier conclusion that $\sigma$ is the decisive parameter. When $\sigma \geq 1$, the recurrent kernel has one stable fixed point at $\kappa = 1$ to which all orbits converge. As soon as $\sigma < 1$, this point becomes an unstable fixed point, and a second fixed point emerges. In the vicinity of the unstable fixed point at $\kappa = 1$, the derivative of the iterated function is larger than one, and within this region, different orbits will diverge. When the orbits approach the second stable fixed point, the dynamics will become convergent again.

Using the criterion of having one stable fixed point, we can derive for which $\sigma$ the recurrent arcsine kernel will be stable. The iterative function, using input signals equal to zero, is given by:

$$\kappa_{t+1} = \frac{2}{\pi} \arcsin\left(1 - \frac{1}{1 + 2\kappa_t\sigma^2}\right), \tag{4.35}$$

Taking the derivative of this equation we find

$$\frac{\partial\kappa_{t+1}}{\partial\kappa_t} = \frac{2}{\pi}\frac{2\sigma^2}{\sqrt{1 + 4\kappa_t\sigma^2}(1 + 2\kappa_t\sigma^2)}. \tag{4.36}$$

When $\kappa_t = 0$, the fixed point, this leads to:

$$\frac{\partial\kappa_{t+1}}{\partial\kappa_t} = \frac{4\sigma^2}{\pi}. \tag{4.37}$$

In order for this to be smaller than one, we need

$$\sigma < \frac{\sqrt{\pi}}{2}. \tag{4.38}$$

The number $\frac{\sqrt{\pi}}{2}$ is the inverse of the slope of the error function. In the next section I will show that we can take the spectral radius stability argument for ESNs, and extend it to infinite networks. I also show that $\sigma$ is the direct equivalent of the spectral radius of an infinite weight matrix. Hence, if we linearize the error function at zero, we would indeed find that a net gain of

one can be achieved with a spectral radius of $\frac{\sqrt{\pi}}{2}$.
The bottom row of Figure 4.5 shows the iterated function corresponding to the STUN kernel for sign nodes. I chose a fixed in-degree of $K = 16$. It is interesting to note that when $\zeta = 0$, the stable fixed point of this kernel is zero. In terms of network nodes this means that there is no correlation between the hidden states of the two infinite STUNs.
The occurrence of a new stable fixed point is intriguing and can be understood intuitively. In its vicinity, the dynamics are stable, so if we assume that the input signals have a small magnitude, the recurrent kernels have fading memory, even though their associated dynamical systems have not. Suppose we have two identical sign node STUNs with high in-degrees and input signals equal to zero. As shown in the right bottom panel of Figure 4.5, the stable fixed point of the associated kernel is zero, corresponding to completely uncorrelated hidden states. If we would at a certain time frame start to inject input signals into both systems, the kernel value will respond to this, and become non-zero. This means that some of the nodes in the two networks start to exhibit short-lived correlations, which correspond to relations between both input signals. When after some time the input signals are set back to zero, the two networks will continue with purely chaotic behavior, and whatever correlation there was will fade away again and the kernel value will converge to zero. This is how a highly chaotic dynamical system will show no fading memory, but its associated kernel actually will. The more unstable the dynamics, the faster the correlations will disappear, and the more stable the dynamics of the recurrent kernel are, indicating that the role of fading memory and chaos have been switched around.
In section 4.6.3 I will provide an empirical example of this new type of fading memory by showing that kernels operating in such a regime can perform well on a task that requires fading memory.

### 4.5.3 Spectral radius

We can reconsider equation 4.14 for sigmoid nodes, and look at the linear approximation of the sigmoids around the origin, very much like the line of thought followed to reach the spectral radius stability argument. The linear approximation is given by

$$k(\mathbf{s}_i, \mathbf{s}_j) = \int_{\Omega_\mathbf{v}} P(\mathbf{v})(\mathbf{v} \cdot \mathbf{s}_i)(\mathbf{v} \cdot \mathbf{s}_j) d\mathbf{v}.$$

When we wish to make a finite Monte Carlo approximation of this integral using $N$ samples, we can rewrite this as

$$k(\mathbf{s}_i, \mathbf{s}_j) \approx \frac{1}{N}\sum_k (\mathbf{v}_k \cdot \mathbf{s}_i)(\mathbf{v}_k \cdot \mathbf{s}_j) = \sum_k \left(\frac{\mathbf{v}_k}{\sqrt{N}} \cdot \mathbf{s}_i\right)\left(\frac{\mathbf{v}_k}{\sqrt{N}} \cdot \mathbf{s}_j\right),$$

where each instance $\mathbf{v}_k$ is drawn from the distribution $P(\mathbf{v})$. We wish to find out what happens when $N$ goes to infinity. In other words, what is known about the behavior of the spectral radius for very large random matrices?
In Geman (1986), it was shown that, if a square matrix with spectral radius $\rho$ has elements which are drawn i.i.d. from a distribution with zero mean and variance $\sigma^2$, the following relation applies:

$$\lim_{N\to\infty} \frac{\rho}{\sqrt{N}} \leq \sigma. \tag{4.39}$$

In our case the factor $1/\sqrt{N}$ appears naturally as a consequence from the summation approximation, such that we can effectively speak of the spectral radius of an infinite recurrent neural network, being the standard deviation of the distribution of the weights.

## 4.6 Tasks

In this section I will explain the practical side of recurrent kernel machines and investigate their performance on both academical and real-world tasks. First I will explain how I build kernel machines using recurrent kernels operating on time series, and next I will provide examples of both theoretical and practical applications of recurrent kernels.

### 4.6.1 Recurrent kernel machines in practice

Traditionally, in order to train a kernel machine, we need to have a set of input data points $\mathbf{s}_i$, associated with desired output values $y_i$ which are a function of $\mathbf{s}_i$. For a time series, we assume that the desired output $y(t)$ is a function of the recent input history, not only the frame $\mathbf{s}(t)$. Therefore, for recurrent kernels, a support vector will be a time series, ending at the frame associated with a certain output value.
In principle the hidden states that we associate with these kernels will depend on the full input history of the input time series, and as such, a support vector would be the full time series up to a certain frame. In practice, however, this is computationally demanding, and we assume that, due to the fading

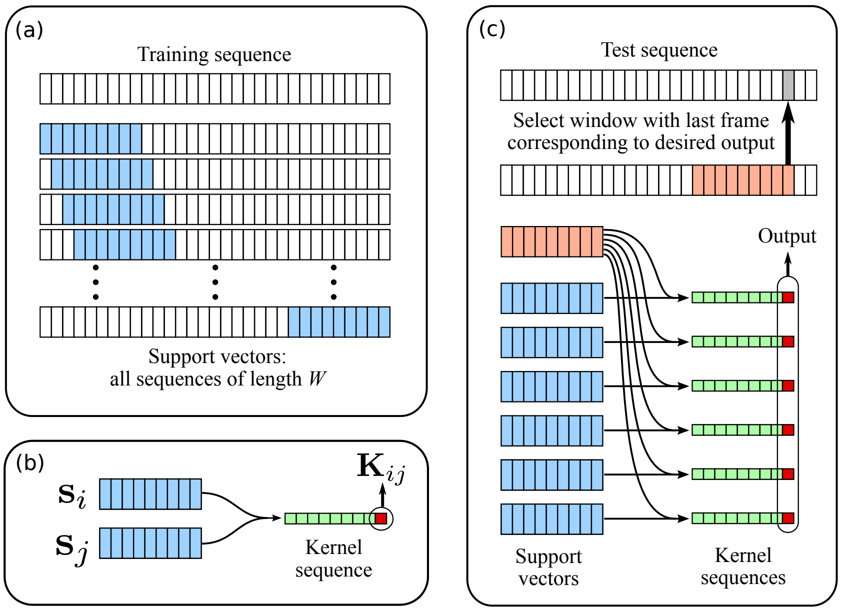

**Figure 4.6:** Schematic display of a recurrent kernel machine. The explanation is in the text.

memory, we can assume that we only need a short history of the input time series to compute a kernel value. We will call the length of this history the window length or recursion depth $W$. We can choose a value $W$ by empirically checking for which window length the kernel value of two time series no longer significantly changes. Note that we need to provide an initial value for the recurrent kernel as well. We will select this as being the kernel value corresponding for input time series equal to zero for an infinitely long history.

Once we have chosen $W$, we need to select a subset of support vectors $\mathbf{s}_i(t)$. In the extreme case we could for instance take all possible sequences of length $W$ from the training set (as depicted in Figure 4.6a), which would be feasible if the training set is relatively small. For larger data sets we will need to make a selection, and we can use any of a number of well-known support vector selection strategies.

When we have selected the set of support vectors, we can construct the Gram-matrix. The support vectors $\mathbf{s}_i(t)$ are defined for $t \in \{1, \cdots, W\}$, such that the entries of $\mathbf{K}$ are given by $\mathbf{K}_{ij} = \kappa_W(\mathbf{s}_i, \mathbf{s}_j)$. This is because the last frames of the support vectors are the frames that are associated with their corresponding output. I have depicted the construction of the

Gram-matrices in Figure 4.6b.
Using, e.g., equation 4.4 we can compute output weights $\boldsymbol{\Upsilon}$. Once these are determined we now wish to apply the kernel machine on new input time series. We will need to compute the kernel values of the support vectors and a recent input history of the input time series, and linearly combine them to form an output. Suppose we have an input time series $\mathbf{s}(t)$, and we wish to compute an associated model output at time $t'$, we will then select a sequence of a certain length $W$ with its last frame being $t'$. If we call this sequence $\mathbf{s}_{t'}(t)$, where $t \in \{1, \cdots, W\}$, the model output $\tilde{y}(t')$ can be computed as:

$$\tilde{y}(t') = \beta + \sum_{i=1}^{N} \upsilon_i \kappa_W(\mathbf{s}_i, \mathbf{s}_{t'}). \tag{4.40}$$

I have given a schematic representation of this principle in Figure 4.6c.
Compared to ESNs, the great downside of recurrent kernel machines is that, in order to obtain a single output frame $\mathbf{y}(t)$, we need to compute the recursive kernels over the full window length, and we need to repeat the process for all $t$ if we want the full sequence of outputs of the test data. By contrast, a single output frame in the ESN setup only requires a single step in the recursion, which is computationally more efficient. The advantage is that we are no longer limited in network size: we can truly find out how well an infinitely large reservoir scores on a task for any finite amount of training data (i.e., a finite number of support vectors), as I have argued in more detail in Hermans and Schrauwen (2011).
In what follows I will make comparisons between recurrent kernels and traditional kernels that operate on time windows. The principle remains largely the same, as support vectors are sequences of length $W$, and their desired output values are associated with the last frame. In order to make single vectors from the time windows, we concatenate all frames into a single large vector to compute the kernel value. Traditional kernels have no fading memory, and contrary to the case of recurrent kernels, the window length $W$ is an additional parameter that needs to be optimized.

### 4.6.2 Memory capacity

Not a 'task' in the true sense of the word, memory capacity tells us a lot about the fundamental restrictions of recurrent kernel machines. We are able to analytically compute the memory capacity for linear recurrent kernel machines using equation 4.15. I will use the same line of thought as described in Chapter 3, i.e., we evaluate how well the linear kernel machine is able to remember a history of a one-dimensional input signal with each frame i.i.d., drawn from a certain distribution with zero mean and unit variance. I apply

equation 3.1 to find an expression for the memory function, from which we can analytically derive the memory capacity $M$. We assume a set of $N$ support vectors $s_i(t)$, which are defined for $t \in \{-\infty, \cdots, 0\}$.
As I show in Appendix A.3.4, it turns out that the linear memory capacity of linear kernel machines is equal to the number of support vectors $N$. This shows that, whereas neural networks encode the history in an $N$-dimensional hidden state, for recurrent kernel machines the history is encoded in the output values of the $N$ kernel functions that are evaluated.

### 4.6.3 NARMA

The second task we consider is the so called NARMA-task, or *Nonlinear Auto Regressive Moving Average*, which has been used for benchmarking in many papers that consider time-series processing (Atiya and Parlos, 2000; Jaeger, 2003; Steil, 2005). The task is a single input single output system, with input $s(t)$, which is a sequence of i.i.d. numbers drawn from a uniform distribution between 0 and 0.5. The desired output $y(t)$ is then constructed as follows:

$$y(t+1) = 0.3y(t) + 0.05y(t)\sum_{i=0}^{9} y(t-i) + 1.5s(t+1)s(t-8) + 0.1. \quad (4.41)$$

As error metric to evaluate performance on this task I used the *Normalized Root Mean Square Error*, or NRMSE, defined as

$$\text{NRMSE} = \sqrt{\frac{\langle y(t) - \tilde{y}(t)\rangle_t^2}{\text{var}(y(t))}}, \quad (4.42)$$

in which $\tilde{y}(t)$ is the output of the trained system.
To compare different effects and results, I performed four experiments, the first three using LS-SVMs.

- First of all, I have used a classic windowed Gaussian RBF kernel (as given by equation 4.16) as a reference value. I optimized both the window length and kernel width by a two dimensional grid search. Using a validation set, I found the optimal window length to be 27 frames and the optimal kernel width $\sigma = 5$, although performance does not change much for a relatively broad range around this optimal value.

- Secondly, I have measured the performance of the recursive Gaussian RBF-kernel in relation to its corresponding spectral radius equivalent: $\rho = \sigma^{-1}$. I limited the recursion depth to 50 frames, although a shorter

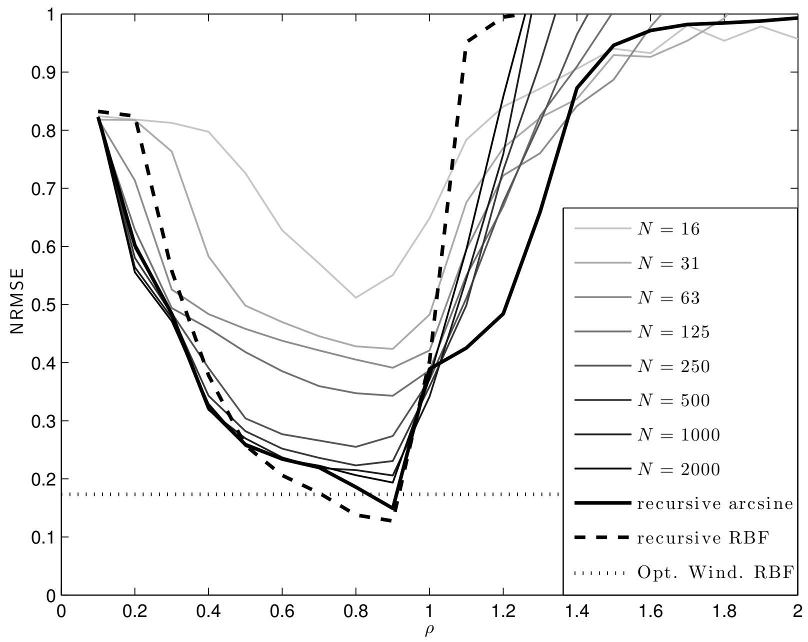

**Figure 4.7:** Mean NRMSE of the NARMA-task for different setups in relation to the corresponding spectral radius $\rho$. The thin, light to dark grey lines are NRMSEs for ESNs with increasing numbers of nodes $N$ (specified in the legend). The thick black line is for the arcsine kernel, i.e. for $N \to \infty$. The dashed line is the performance of the recursive Gaussian RBF kernel, and the dotted line (independent of $\rho$) is the mean NRMSE for optimized windowed Gaussian RBF kernels.

time would likely give very similar results.

- Thirdly I did the same experiment for the arcsine kernel in relation to its corresponding spectral radius $\rho = \frac{2}{\sqrt{\pi}}\sigma$. I again used a recursion depth of 50 frames.
- Finally, as arcsine kernels are strongly related to ESNs, I used the opportunity for a comparison. I measured the performance of ESNs with error function non-linearities for an increasing number of nodes and in relation to the corresponding spectral radius.

In all of the above experiments I used a training set of 500 time series, which consists of all possible time series drawn from a sequence with length $500+W$, ($W$ being window length), a validation set of 2000 frames, used to determine the optimal regularization parameter, and a test set of 5000 frames. For the

recursive kernels and the ESNs, performance as a function of input scaling has a broad, shallow optimum (data not shown), but nevertheless the scaling factors were optimized by a grid search at a corresponding spectral radius of 0.9, which resulted in an optimal scaling factor of the input of 0.1 for the arcsine kernels and the ESNs, and 0.4 for the recursive Gaussian RBF. All results were found by averaging out over 100 different trials with newly generated data and/or reservoirs.
Results of the experiments are shown in figure 4.7. Optimal performance can be found around $\rho = 0.9$. Performance of the ESNs gradually increases with the number of nodes, converging slowly to the performance of the arcsine kernel. As $\rho$ becomes greater than one, performance rapidly deteriorates. The recursive Gaussian RBF kernel performs best, and both recursive kernels perform better than the classic time window RBF kernel.
In Hermans and Schrauwen (2012) I measured performance of STUN kernels for sign nodes on the NARMA task. Their performance is not particularly good, but I found that they do neatly show how recurrent kernels can still exhibit fading memory, even when their associated DSs are strongly chaotic. The training setup is completely the same as described above, except for an additional preprocessing step on the input signal that needed to be performed. I define the preprocessed input signal as $s^*(t) = 1 + \epsilon(s(t) - 0.25)$, i.e., I first center the signal around zero and rescale it with a small parameter $\epsilon$, and next shift it to have a mean equal to one. This will make sure that the assumptions on the input signal I made in section 4.5.1.2 are met approximately, and we can apply the formula for the Lyapunov exponent. The parameter $\epsilon$ I optimized using a grid search. Finally, before inserting $s^*(t)$ in the model, I scaled it with $\zeta$, which will provide control over the Lyapunov exponent.
I have measured the average NRMSE for a logarithmic range of $\zeta$ values for STUN kernels with in-degree $K = 16$, spanning its full potential range of associated Lyapunov exponents. In Figure 4.8, I have plotted the average NRMSE against the exponential of the LE, which has a similar interpretation as the spectral radius in Figure 4.7.
The first minimum we encounter we can readily associate with fading memory, with $\exp(\ell)$ slightly smaller than one. The second one we can find at a higher Lyapunov exponent, where the underlying dynamics are unstable. Here we encounter the type of fading memory I have explained in section 4.5.2. This indeed confirms that the role of fading memory and chaos can be switched around in recurrent kernels.

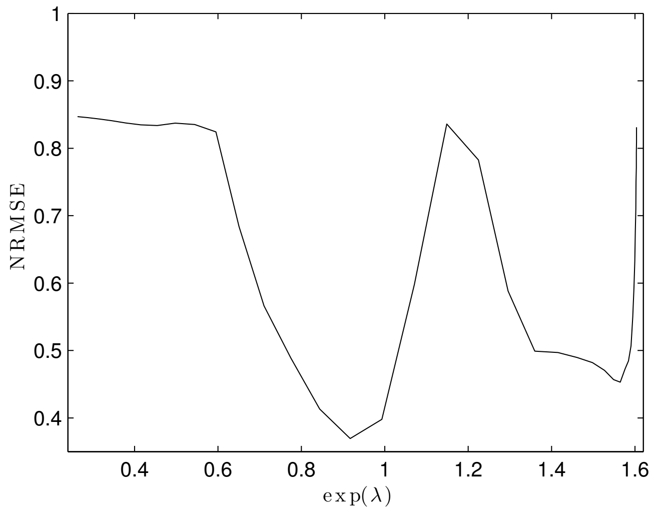

**Figure 4.8:** Mean NRMSE of the NARMA-task for STUN kernels with sign nodes and in-degree 16, as a function of the associated LE.

### 4.6.4 TIMIT

The second task I consider is a speech recognition task in which the goal is to classify phonemes, which are the smallest segmental unit of sound employed to form meaningful contrasts between utterances. I used the internationally renowned TIMIT speech corpus (Garofolo et al., 1993) which consists of 5040 English spoken sentences from 630 different speakers representing 8 dialect groups. About 70% of the speakers are male and 30% are female.
The speech is labeled by hand for each of the 61 existing phonemes, which was reduced to 39 symbols as proposed by Lee and Hon (1989). The TIMIT corpus has a predefined train and test set with different speakers. The speech has been preprocessed using Mel Frequency Cepstral Coefficient (MFCC) analysis (S.Davis and Mermelstein, 1980), which is performed on 25 ms Hamming windowed speech frames and subsequent speech frames are shifted over 10 ms with respect to each other. Each frame contains a 39-dimensional feature vector, consisting of the log-energy of the first 12 MFCC coefficients, as well as their first and second derivatives (the so-called $\Delta$ and $\Delta\Delta$ parameters).

#### 4.6.4.1 One vs. one classifiers

In order to classify each frame into one of the 39 possible classes, I use a voting system that starts from a set of all possible one vs. one classifiers. Each of these classifiers is trained to distinguish between two specific phonemes, and as there are 39 classes there are $(39 \times 38)/2 = 741$ one vs. one classifiers. Each classifier is only trained on data labeled with its corresponding phonemes and outputs either 1 or $-1$ (the sign of the output value). Final classification is performed by letting the classifiers each cast a vote.

#### 4.6.4.2 Training method

One of the difficulties of using SVMs to train on TIMIT is the fact that the dataset is very large. The training set consists of 1,124,823 frames, and the test set of 353,390 frames. The number of frames per one vs. one classifier is of the order $10^4$ to $10^5$. Traditional SVM methods for classification would run into practical computational problems for such large datasets, and therefore I will use the Newton approximation I elaborated on in section 4.1.4.
I trained each one vs. one classifier on a subset of $10^4$ samples and chose a separate validation set of 2000 samples, both randomly drawn from the total training dataset associated with the corresponding labels. If the total number of samples in the set was smaller than 12000, I randomly drew 1000 samples as validation set and used the rest for training.

#### 4.6.4.3 Subsampling and parameter optimization

I have tested on both windowed Gaussian RBF kernels, recursive Gaussian RBF kernels, and arcsine kernels. For both types of recursive kernels I also investigate the effect of subsampling the MFCC-data. It was found in Triefenbach et al. (2010) that large recurrent neural networks perform better on phoneme recognition if the nodes are leaky integrators, conform the network model discussed in section 2.1.3. Rather than incorporating this into our kernels[8], I subsampled the data by a factor of 2, 3 and 5. This essentially means that we speed up the data rather than slowing down the dynamics of our system. In Schrauwen et al. (2007) it has been shown that both strategies yield similar performances.
For the non-subsampled variants of the data I classify on the third frame of the time window or recursion depth, i.e., the SVM needs to classify the phoneme of two frames in the past. For the subsampled versions, I classify

[8]As I will discuss at the end of this chapter, finding a recurrent kernel equivalent for leaky integrator nodes seems far from trivial

**Table 4.1:** Results on TIMIT. Results found in literature are listed under the line.

| | FER | $\overline{N}$ | $W$ | $\sigma$ | inp. sc. f. |
|---|---|---|---|---|---|
| Windowed RBF | 31.5% | 1465 | 9 | 0.06 | |
| Rec. RBF | 30.6% | 1386 | 10 | 1 | 0.045 |
| Rec. RBF 2× subs. | 29.4% | 1499 | 10 | 1 | 0.08 |
| Rec. RBF 3× subs. | 28.7% | 1504 | 10 | 1 | 0.06 |
| Rec. RBF 5× subs. | 28.5% | 1100 | 5 | 0.8 | 0.125 |
| Arcsine | 30.5% | 1105 | 15 | $1.75\frac{\sqrt{\pi}}{2}$ | 0.026 |
| Arcsine 2× subs. | 29.3% | 1511 | 8 | $2.25\frac{\sqrt{\pi}}{2}$ | 0.035 |
| Arcsine 3× subs. | 28.6% | 1377 | 8 | $2\frac{\sqrt{\pi}}{2}$ | 0.04 |
| Arcsine 5× subs. | 28.9% | 1210 | 8 | $2\frac{\sqrt{\pi}}{2}$ | 0.045 |
| Cheng et al. (2009) | 39.3% | | | | |
| Crammer (2010) | 30.0% | | | | |
| Crammer (2010) | 29.2% | | | | |
| Keshet et al. (2011) | 27.7% | | | | |
| Keshet et al. (2011) | 26.5% | | | | |
| Cheng et al. (2009) | 25.0% | | | | |

on the second frame (effectively the third, fourth and sixth frame in the respective non-subsampled datasets). Rather than optimizing the parameters for each one vs. one classifier, I looked for globally optimal parameters by randomly selecting 250 from the 820 classifiers and trained them on a small training, validation, and test set of 1000 samples each, drawn randomly from the corresponding full training set and measured the average test error over a relevant range of parameters. The window size of the Gaussian RBF kernel was determined this way. Recursion depths of the recursive kernels were determined by making sure the kernel value differed on average less than one percent from its asymptotic value.

#### 4.6.4.4 Results

Typically, the performance on the TIMIT dataset is evaluated based on the phoneme error rate. However, this requires an additional mechanism such as an HMM to segment the frames into groups corresponding to phonemes. As we are only interested in the relative performance of the kernels, I restricted myself to only measuring the frame error rate (FER), i.e., the percentage of input windows which were classified incorrectly. The result (FER), average number of support vectors per one vs. one classifier ($\overline{N}$), window size / recursion depth ($W$), optimal $\sigma$ and the scaling factor of the input for each variant are shown in Table 4.1. FER for the subsampled versions of the test set were determined by labeling the missing frames with the nearest classified frame in the case of subsampling $3\times$ and $5\times$. In the case of $2\times$ subsampling, the FER was calculated twice by using the classification of both the previous and next frame as label for the missing frames, and I took the average of both FER's.

The fact that most literature doesn't mention FER makes it hard to compare our results to the state of the art, but some papers actually do mention FER. To give some idea of our performance in general I have included some representative results in the table.

All the techniques with recursive kernels outperform the classical windowed Gaussian RBF kernels, even without subsampling, and it is obvious that subsampling gives a boost in performance. Intriguingly, I found that the optimal spectral radius of the arcsine kernels is greater than one. Upon examining the necessary recursion depth I found that these kernels do indeed only depend on a finite history of the input time series. This is due to the relatively high variance of the input, which pushes the kernels into the saturating part of their non-linearity.

It is interesting to note that in the case of the non-subsampled dataset we find that the number of support vectors is lower for the recursive kernels than for the windowed kernels. This seems to suggest that the recursive kernels are better at capturing the inherent structure of the speech data[9].

In Triefenbach et al. (2010), the same task was studied by (among other techniques) using a very large reservoir of 20,000 nodes. The FER found for this setup was 29.1% (FER is not mentioned in the paper, but I know so from personal communication with the authors), which is comparable to our own results. Currently, large-scale reservoirs have been shown to attain performance matching state-of-the-art in word recognition (Jalalvand et al., 2012).

[9]This comparison would be unfair for the subsampled datasets as these are smaller.

## 4.7 Comparing recurrent kernel machines with reservoirs

In section 4.5.3 I used a Monte Carlo method to approximate the dynamics of an infinitely large DS. This offers an intriguing new interpretation of the functionality of reservoirs: they can be considered as finite approximations of infinite DSs. In this section I will elaborate on the link between recurrent kernel machines and RC.
Recurrent kernels and reservoirs are defined by only a small set of parameters: the nature of the nodes and the distribution of the weights. Both ML techniques are used to model the dynamical process that defines an applications. The set of parameters that we use to define a reservoir can be seen as a prior assumption on the process that generated the data you wish to model. If we generate an ESN with a certain spectral radius, input scaling and leak rate, and use this to solve a speech recognition task, we generally assume that the process that generated the data is similar to the dynamical behavior of the ESN, such that there is a good chance it will contain this dynamical behavior at least approximately within the dynamics of its hidden state. How do we need to view the role of optimal parameters in recurrent kernels?

### 4.7.1 Recurrent kernel machines

As I stated before, under the right conditions, infinite non-linear DSs are computationally universal. This means that within the infinite-dimensional hidden state there will be some subspace that perfectly models the task you wish to solve. So why do we need to optimize parameters at all?
When we consider a recurrent kernel machine, each kernel value we use is the inner product of the last hidden state, caused by the time series support vector and that of the hidden state caused by the time series we wish to process. The hidden states of the last frames of the support vectors are fixed. This means that each time we evaluate the recurrent kernel machine, we project the infinite-dimensional hidden state on a fixed set of infinite dimensional vectors. Essentially, the hidden states associated to the last frames of the support vectors can be seen as a set of untrained readout weights, as we know them from classic RC. This means that the set of kernel outputs provides a finite-dimensional projection of the infinite-dimensional hidden state.
Why do we project onto this set of hidden states, and not just take any projection? Very likely, any randomly chosen direction in this infinite-

dimensional feature space will by almost completely orthogonal on the hidden state vector and as such produce very little useful information. If we use actual hidden states to project on, these are much more likely to produce informative kernel values, as these hidden states will reside within the part of feature space that contains the hidden states.
This means that we can view recurrent kernel machines as nothing more than a way to provide a finite set of well-chosen projections of the infinite hidden state. I will now address the next question: what is the role of the parameter set?
As I mentioned, the universally optimal solution to the task you wish to solve is present within some subspace of the infinite-dimensional feature space, and a recurrent kernel machine will try to approximate this projection by linearly combining the projections provided by the kernels. I believe that optimizing the parameters increases the likelihood that such an approximation will work well. In precise terms: if we consider the distribution of the infinite-dimensional hidden state, a good parameter set will ensure that the optimal projection direction will have a high likelihood to lie within this distribution. This will ensure that sampling projections from within this space can efficiently approximate this optimal direction.
From the point of view of someone modeling an ML task, we can state that good parameters for recurrent kernels essentially makes sure that the dynamics that define the application are well captured by the inherent dynamics of recurrent kernels.

### 4.7.2 Reservoir Computing

Recurrent kernels provide sample projections from the infinite-dimensional hidden state, which then need to be linearly combined in order to produce an output. The approximation here lies in the fact that we need to use a finite set of kernels to work on. Reservoir Computing makes the approximation on a different level. Instead of directly projecting from the infinite hidden state, reservoirs will provide a finite-dimensional model of it, and use the model hidden state as a feature vector that can be directly used for linear regression. The inherent assumption that is made here is that most of the dynamics of the infinite-dimensional hidden state can be captured by a finite dimensional DS with similar properties.
Notice that the approximation here is much rougher. The only guarantee that you have is that the reservoir will start to approach the dynamical behavior of the infinite DS for very large numbers of neurons. For any limited training data set, recurrent kernels will always produce the asymptotical performance for $N \to \infty$, $N$ being the number of neurons. It should be

stated that this does not mean that recurrent kernels will always perform better than each individual reservoir. It is still possible to sample a reservoir that happens to work very well on a certain task. In reality, however, this probability is low, and an experimenter might need to take a prohibitively large number of samples before finding one that beats recurrent kernel machine performance.

The plus side of the reservoir approximation is that we can directly find the optimal projection direction, i.e., the optimal readout weights $\mathbf{U}$. In the case of recurrent kernels we could only project from a subspace spanned by the hidden states of the support vectors. With reservoirs we have no such limitation, and especially in the case of very large training sets, this provides an important advantage.

## 4.8 Conclusions

In this chapter I have defined recurrent kernels, which can be associated with infinitely large dynamical systems. The output value of such a kernel operating on two time series is the inner product of the hidden states of an infinite-dimensional DS, associated with the two input sequences.

I provide a range of examples of recurrent kernels. Some are based on specific neural network models, such as ESNs and sparse threshold function networks, others are not explicitly based on network models, but on existing kernel functions that can be made recurrent, such as the Gaussian RBF kernel.

Using the recurrent kernel function, it is possible to make predictions on the stability of the DS it is associated with. I present a method for some types of recurrent kernels to derive the Lyapunov exponent of the associated DS. This allows us to find a closed expression which reveals the role of all relevant parameters in the dynamical stability of the DS. For STUNs, I show that the expression for the Lyapunov exponent can be validated empirically with finite networks.

For other types of recurrent kernels, especially that of infinite-sized ESNs, the Lyapunov exponent cannot easily be derived, and I resort to the Banach fixed-point theorem and cobweb diagrams to analyze the dynamics of the recurrent kernel more directly. As it turns out, recurrent kernel dynamics can introduce a stable fixed point, even when the dynamics of the underlying DSs have a Lyapunov exponent greater than zero. Around this stable fixed point, the recurrent kernel dynamics will have fading memory, which still allows for good performance on time series processing, even in an unstable regime.

In the final part of the section on kernel dynamics, I link the concept of the spectral radius with recurrent kernels. If we approximate the recurrent kernel by taking a finite hidden state, we can directly associate the stability criterion of recurrent kernels with the notion of the spectral radius. Inversely, this leads to the notion that reservoirs can be considered as a finite sample from an infinitely large DS.
I explain how we can use recurrent kernels in common kernel machine setups. First I use this framework on an academic task, where I show that the performance of a recurrent kernel machine is the asymptotic limit of increasing the number of nodes in ESNs. Next I apply recurrent kernels on a challenging, real world speech recognition task, which yields performance close to the state-of-the-art.
I concluded this chapter by arguing that recurrent kernel machines and reservoir computing can be seen as two different approximations of the same mathematical concept: infinite DSs. Recurrent kernel machines are particularly practical for small training sets, as they yield performance that is the asymptotic limit of infinitely large reservoirs. Reservoirs have the advantage that, even though they only crudely approximate infinite-dimensional DSs, they can be trained on unlimited amounts of data.

## 4.9 Future work and open questions

### 4.9.1 Expanding beyond simple kernel machines

As I mentioned before, recurrent kernels have one strong disadvantage compared to ESNs: their computational demand. Each kernel value is obtained recursively through a large set of operations on the data, and all these computations only deliver a single piece of information: the final kernel value in the sequence. One of the most obvious extensions of the current framework would be to not simply limit the output to this single value, but also to include the intermediate kernel values. If we have two time series $\mathbf{s}_1(t)$ and $\mathbf{s}_2(t)$, with $t = 1 \cdots W$, we have to calculate all $W$ kernel values $\kappa_t(\mathbf{s}_1, \mathbf{s}_2)$ to obtain the desired output value $\kappa_W(\mathbf{s}_1, \mathbf{s}_2)$. In the current setup, whatever information these other kernel values might contain, is thrown away. It is an interesting thought to redesign the current kernel machine systems such that it is able to include all this information. Each of these outputs would then depend on an increasingly long context. Most importantly, the expressive power of our model would increase $W$-fold at roughly the same computational cost.
Such a system could not be trained in the classical LS-SVM setup, which

requires a single output value for each kernel evaluation. Using simple least squares to find optimal weights for such a system might also be problematic, due to the fact that we now will have $W \times N$ trainable parameters ($N$ the number of support vectors), the output weights for each output. Perhaps finding these parameters is best tackled with gradient descent or similar iterative learning strategies.
A similar idea would be to define a linear combination of the sequence of output values, which would produce a new kernel function that can be optimized for the task. This idea is related to the field of *multiple kernel learning* (MKL) (Bach et al., 2004; Lanckriet et al., 2004; Sonnenburg et al., 2006). Here, the goal is to solve a task by creating a novel kernel as a linear combination of several different kernels. Insights in the domain of MKL could be readily applied on this problem.

### 4.9.2 Building efficient recurrent models from feature space

Compared to the ordinary RC setup, kernel machines have the disadvantage of being difficult to scale to large datasets. Whereas RC delivers a finite feature vector, which is independent of the size of the dataset, for kernel machines one could say that each additional data point generates a new feature.
One method that tackles this problem is the so-called *Nyström-approximation* (Williams and Seeger, 2001); a way to create a finite-dimensional feature vector from a small set of support vectors. In essence the Nyström-approximation boils down to selecting a small subsample of data points, and use (a linear transformation of) the kernel values of these data points with those of the whole dataset as feature vectors, such that we can include as much training data as we like.
With recurrent kernels it would be far more interesting to attempt to go one step further: to actually try and use the sequence of these feature vectors to create a finite-dimensional recurrent model that approximates the dynamics of the infinite-dimensional hidden state. Suppose we have a sequence of feature vectors $\mathbf{x}(t)$, created from a time series $\mathbf{s}(t)$. We can then for instance try to optimize weights $\mathbf{W}$ and $\mathbf{V}$ such that

$$\mathbf{x}(t+1) = \tanh(\mathbf{W}\mathbf{x}(t) + \mathbf{V}\mathbf{s}(t+1)).$$

If this would be possible, it would mean that we can create a recurrent neural network that approximates the dynamics of the infinite dynamical system. This would be based on the data of the time series, in the sense that it is based on a set of support vectors, and it would provide a truly recurrent system, where each new hidden state can be computed instantly from the previous hidden state and the current input frame. This is much faster than the cumbersome windowed method in which recurrent kernel machines are executed at the moment.

### 4.9.3 Recurrence versus fading memory

Given that we have shown a clear advantage of recurrent kernels over classic kernels operating on time windows, we run into the question as to why this is the case. Most importantly, we need to ask the question: *is it because of fading memory, or is it due to the recurrence?*
For example, we could artificially introduce fading memory into time windows, simply by reweighting each frame such that they become gradually less important towards the start of the time window. As a consequence, the last few frames will matter most, and frames further in the past less and less. Next, we could use this reweighted input (which is nothing but a linear transformation) as input into a more common kernel, such as Gaussian RBF. Would such a kernel function be able to compete with, or even transcend recurrent kernels? Perhaps part of the relative success of recurrent kernels truly resides in the recurrence itself; the fact that it inherits the prior assumption of causality. Notice that this question is not limited to kernels, and applies equally well to recurrent neural networks.
It is important to mention that these questions are tightly bound to non-linearity. For a linear kernel, there is no distinction between recurrence and fading memory. For non-linear kernels this difference is more obvious. In the example I gave above, the fading memory is due to a linear transformation. For recurrent kernels, the iterated non-linearity directly contributes to fading memory, making the two approaches fundamentally different. Therefore, if experiments are conducted that need to confirm or deny this question, I would strongly suggest to try it on a highly non-linear task, as it will amplify the difference between the two.

### 4.9.4 Recurrent kernels for continuous media

In chapter 2 I discussed some of the more exotic implementations of RC. One of the most prominent candidates are physical reservoirs, where we for instance connect a set of weights via fixed bars, springs, and dampers. Such a system, when set into motion, can serve perfectly well as a physical reservoir (Hauser et al., 2012), where the hidden state would consists of the current position and speed of the weights. Using the same line of thought as we did

at the start of this chapter, is it possible to increase the size of such a system to infinity, where we exchange detailed knowledge on the connections for a simple statistical model.
Such an infinitely large set of connected weights could be considered a continuous medium. If we would assume that the nodes of the dynamical system are only locally connected, and we zoom out sufficiently from the microscopic view of springs and dampers, we would end up with an elastic medium.
Other candidates for continuous media reservoirs would be models of chemical reaction-diffusion systems. These are systems in which you have a medium in which local chemical reactions occur, and of which the resulting reactants diffuse throughout the medium, influencing chemical reactions elsewhere. Many phenomena can be modeled by reaction-diffusion systems, such as the famous BZ-reaction (Belousov, 1959; Zhabotinsky and Zaikin, 1973), the FitzHugh-Nagumo equation that models how action potentials travel through nerves (FitzHugh, 1955; Nagumo et al., 1962), and blood clotting (Ataullakhanov et al., 2007). Indeed, their interesting spatiotemporal dynamics have been suggested as a potential computational entity (Dale and Husbands, 2010).
Recurrent kernels could form an interesting approach to model spatiotemporal dynamics in continuous media. If it is analytically tractable to find the kernel, it is also possible to investigate the dynamical stability of the system, or to explore how well such media may actually perform when applied as reservoirs.

### 4.9.5 Continuous time recurrent kernels

Very related to the previous part, one truly interesting challenge is to try and extend the whole framework of recurrent kernels to continuous time. Not only would this have useful practical applications (in the same vein as adding leak rate can boost ESN performance), continuous time recurrent kernels would make it possible to analyse models of very large continuous time dynamical systems, similar to the way STUNs are models of gene regulation networks. An interesting example of a continuous time dynamical system are spiking neural networks. Spikes can occur at any moment in time, and the underlying dynamics of the cell are continuous, governed by a differential equation.
Let us consider a generic differential equation, based on 2.3, which would describe the hidden state of a continuous time reservoir:

$$\dot{\mathbf{a}}(t) = \frac{1}{\tau_R} \left( f(\mathbf{a}(t), \mathbf{s}(t)) - \mathbf{a}(t) \right).$$

If we have two signals $\mathbf{s}_1(t)$ and $\mathbf{s}_2(t)$, these will have two associated hidden states $\mathbf{a}_1(t)$ and $\mathbf{a}_2(t)$. The kernel function we are after is then given by

$$\kappa_t(\mathbf{s}_1, \mathbf{s}_2) = \mathbf{a}_1(t) \cdot \mathbf{a}_2(t).$$

Solving this equation directly seems intractable. Possibly there exists a solution for $\kappa_t(\mathbf{s}_1, \mathbf{s}_2)$ in the form of a differential equation. We can write:

$$\begin{aligned}
\dot{\kappa}_t(\mathbf{s}_1, \mathbf{s}_2) &= \dot{\mathbf{a}}_1(t) \cdot \mathbf{a}_2(t) + \mathbf{a}_1(t) \cdot \dot{\mathbf{a}}_2(t) \\
&= \frac{1}{\tau_R} \left( f(\mathbf{a}_1(t), \mathbf{s}_1(t)) \cdot \mathbf{a}_2(t) + f(\mathbf{a}_2(t), \mathbf{s}_2(t)) \cdot \mathbf{a}_1(t) - 2\mathbf{a}_1(t) \cdot \mathbf{a}_2(t) \right) \\
&= \frac{1}{\tau_R} \left( f(\mathbf{a}_1(t), \mathbf{s}_1(t)) \cdot \mathbf{a}_2(t) + f(\mathbf{a}_2(t), \mathbf{s}_2(t)) \cdot \mathbf{a}_1(t) - 2\kappa_t(\mathbf{s}_1, \mathbf{s}_2) \right)
\end{aligned}$$

The last term in the above equation is a leaky integration. The two first terms are the drivers of the evolution of $\kappa_t(\mathbf{s}_1, \mathbf{s}_2)$. Solving these however, seems to be far from trivial. Hopefully this conundrum will be resolved in future research.

# 5 Training Recurrent Networks

Thus far, the main theme of my doctoral research was focused on reservoir computing. Nevertheless, I have spent some thought and effort on working with more traditional means of training recurrent networks. This chapter will discuss both published and unpublished results in this field. I will start by elaborating on gradient descent via error back propagation, and more specifically back propagation through time. Next I will talk on simplifying this concept, and I will look at training only the input and output weights of a network. The second part deals with very preliminary work on designing layered architectures of recurrent networks which may have some advantages in terms of training speed and stability.

## 5.1 Where reservoirs fail

Before we begin with this chapter, I will first explain why trained RNNs are in fact needed at all. Consider the following task: we have one-dimensional input time series $s(t)$, where each element is randomly drawn from the set $\{1, 2, \cdots, 9, 10\}$. The desired output at each moment in time we define as $y(t) = s(t - s(t))$, i.e it must provide $s(t)$ delayed with the current input value.
Solving this task may *seem* straightforward at a first glance, simply because its prescription is so short and easy to understand. Nevertheless, solving this task with a standard reservoir is extremely hard. My coworker, Francis wyffels, who is experienced in optimizing reservoir systems, has applied reservoirs of 1500 nodes on this task, optimizing spectral radius, input and bias scaling. The best result he found after searching parameters for about one day was a test NMSE of 0.54, still rather high. When I trained an RNN

of 38 neurons explicitly[1], which has about the same number of trainable parameters as the reservoir, it took only a few minutes to reach a test NMSE of about 0.006.

Why is there such a great difference? The reason is likely the *very* non-linear nature of the task. The RNN is required to store the history of the last ten input entries, on its own an easy assignment, and next pick out a different item from this registry each frame in time and send it to the output. If we put this in terms of the reservoir as a filter, the network output needs to emulate a filter that completely changes shape every moment in time.

Even an extremely non-linear and relatively large ESN is unlikely to contain this highly specific quality within its random set of functions. In fact I believe that this demonstrative exercise gives insight into a larger problem, which I can put into the following, intuitive terms:

*There are more ways in which to be strongly non-linear than weakly non-linear; the number of functions for each given degree of non-linearity increases rapidly as this degree goes up.*

For example: we can express every possible $N_{out}$-dimensional linear function on an input vector of size $N_{in}$ using a matrix of size $N_{in} \times N_{out}$. On the other extreme of the spectrum, for modeling *any* non-linear function, we need an infinite amount of parameters. If we would have a task that requires an close-to-linear mapping of the output, chances are good that in any random ESN instantiation the right set of filters will be present. As a task becomes more and more non-linear though, the chance of finding the right *kind* of non-linearity decreases dramatically, simply because there are so many ways a function *can* be non-linear.

Herein the strength of explicit training of networks becomes obvious. If the wanted non-linearity is highly complex, but can in principle be encoded in the parameters of an RNN, a directed search has a chance of finding it, whereas random ESN instantiations are sampling from such a tremendously large search space that the chances of discovering an adequate one are practically zero.

Obviously, since we know the nature of the example task that I proposed here, it is still possible to have a reservoir-like setup which can solve it. For instance we could split the problem into two parts: we train one reservoir to act as a delay line, already transforming the problem from a temporal to a spatial one, and next take a (memoryless) random NN that needs to select the right entry from this delay line. Two altogether much easier tasks. Alternatively, we could reformulate the task as a classification task, where we transform in- and output to ten-dimensional classifiers (the input being all zeros and a single one on the corresponding channel of the current input

[1]Using backpropagation through time, which I'll explain later.

frame). For real-world tasks however, such 'expert knowledge' is not necessarily available, or far too difficult to find. If we cannot have a directed search to a good solution, random dynamical systems will never provide a practical solution.
Real world tasks that are very non-linear, and cannot be readily tackled with RC do indeed exist, and can be quite interesting. One of the tasks I will consider in this chapter is next-character-prediction in a text. Here, the difficulty lies in the fact that, even before grammar and punctuation comes to question, the model needs to learn a very large number of words. Put differently: it needs to encode a very large list of highly specific sequences. A classical reservoir will only be able to explicitly encode words into its output weights, as they are the only parameters that are trained. Other than that, the number of words the reservoir can learn will then largely rely on chance. The expressive power of trained RNNs is far greater, hence the number of words they can encode will be substantially larger. For this reason, in this chapter I will study explicit training strategies.

## 5.2 Gradient descent: backpropagation through time

Reservoir computing has a quite specific perspective on Machine Learning. In its rawest form, it takes a very large number of nonlinear spatiotemporal functions of the input data, and correlates the desired output function with a direction in this high-dimensional space. From this point of view, it is clear that the larger the reservoir, and hence the larger the feature space from which the output is projected, the better the system will work. The best reservoir in existence is infinitely large.
The table turns when we think of a wholly different scenario: I only give you a finite number of neurons, and hence a fundamentally limited representational power, to solve a particular task. But to keep things fair, I will give you complete freedom of choice over the parameters (weights) of the network. How should you go about to tweak and tune them? This question is what this section is concerned with, and I will discuss a widespread solution to this problem: gradient descent.
First, I describe the idea of minimizing a cost function by searching the direction in parameter space which points in the direction of the largest reduction in cost. Next I will discuss the limitations of such an approach and finally I will elaborate on how we can apply it on recurrent neural networks.

### 5.2.1 Cost gradient

We start from a similar objective as in Chapter 2: we wish to minimize a certain cost, more specifically the Mean Square Error of the output of our model. Suppose that we have a model, defined by a set of parameters $\boldsymbol{\theta}$. Given a certain amount of training data which defines the task we wish to solve, we can define the total MSE as a function of $\boldsymbol{\theta}$. This function is the cost we wish to minimize.
If we assume that MSE($\boldsymbol{\theta}$) is continuous and sufficiently smooth, there exists a rather straightforward optimization method. Start with a model that has random parameters. Obviously the odds that it will perform even remotely satisfactory on the task you wish to solve are minute. But what we can do is to see in which direction in parameter space the MSE will decrease. In order to do so we need to define the *gradient* of the MSE with respect to $\boldsymbol{\theta}$:

$$\nabla_{\boldsymbol{\theta}}(\text{MSE}) = \frac{d\text{MSE}(\boldsymbol{\theta})}{d\boldsymbol{\theta}}. \tag{5.1}$$

This gradient has the same dimensionality of $\boldsymbol{\theta}$ and points in the direction for which the MSE will increase the fastest. When we wish to obtain a model that scores better on the task we have to change the parameters in the opposite direction. We define new parameters such that

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}}(\text{MSE}). \tag{5.2}$$

Here, $\eta$ is a (usually small) parameter that determines the size of the step we take in parameter space. We let this algorithm run until the cost function no longer decreases or until we reach a maximal number of iterations. When the cost function no longer decreases we say that the learning algorithm has converged. I have made an illustration of the principle of gradient descent in Figure 5.1.

### 5.2.2 Weaknesses of gradient descent and their solutions

Here I discuss some of the typical problems that occur when using gradient descent algorithms, together with some of the solutions.

#### 5.2.2.1 Computational demands

Normally, each gradient calculation will require you to run the model on the complete dataset with the current parameters. For very large datasets

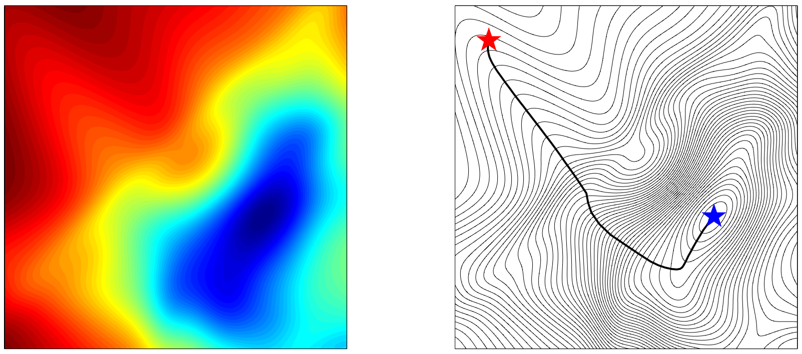

**Figure 5.1:** Illustration of gradient descent. On the left I have depicted a color coded cost function. The two dimensional plane is the space of the parameters, and the color indicates cost (red is high and blue is low). The right panel is a depiction of the same function, but now as a contour plot. The algorithm starts in the red star. It finds the local gradient, the direction perpendicular to the contours, and moves a step in that direction. It repeats that process until it has converged at the minimum value at the blue star.

this means that a single iteration might be a computationally expensive operation. Normally a relatively large number of iterations is required before the solution converges, and running gradient descent becomes infeasible. One of the solutions to this problem is to use *stochastic gradient descent* (Bottou and Bousquet, 2008). Instead of running all data through the model, we sample small subsets of data and calculate the gradient on that. Even though the gradient is less accurate, the number of parameter updates can be greatly increased this way and convergence to an optimal value is much faster.

#### 5.2.2.2 Initial parameters

It remains a challenge to find a good heuristic for choosing the initial values for the parameter set $\boldsymbol{\theta}$. If the region with low cost is very far away from the initial parameters, gradient descent will wander around aimlessly, and it might never reach a region with good performance. There are no real guidelines as to what is a good starting point, as it depends on the model and the task, and an experimenter usually has no choice but to use trial and error.
Related to the problem of good initial parameters is the choice of the learning speed. If it is too high, the jumps in parameter space are too large and the

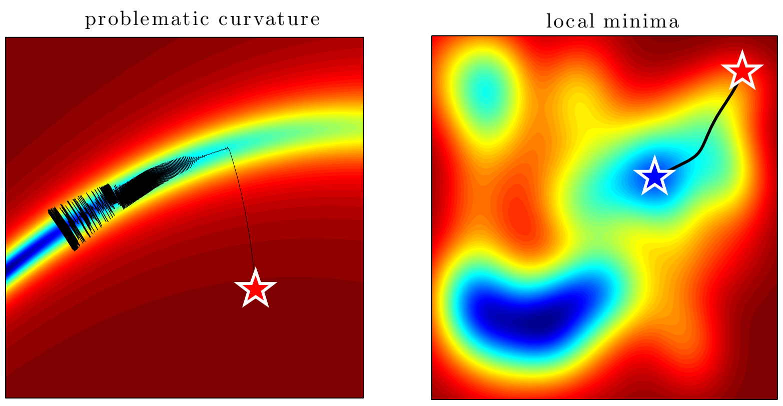

**Figure 5.2:** Illustration of two common problems with gradient descent. On the left panel I show a cost landscape as a valley that gradually narrows and deepens. The gradient descent algorithm starts at the red star, finds the valley and descends into it. As soon as the gradients at the edges of the valley grow too large however, the search algorithm will start to oscillate and jump back and forth between the valleys edges. Finally it no longer moves in the direction in which the valley deepens. The right panel show an example of a local minimum. Even though a lower cost can be attained, the gradient descent algorithm doesn't find it and converges to a suboptimal parameter set.

algorithm may overshoot the optimal value or even be unstable and cause the parameters to grow unboundedly. If it is too small, convergence of the learning algorithm will be too slow and a good solution might not be found within a reasonable time. Often, experimenters will gradually reduce the learning speed as performance increases. This ensures that the search algorithm will settle at the local minimum in the cost landscape.

#### 5.2.2.3 Local optima and problematic curvature

Gradient descent works best when the cost landscape is smooth and has one single deep valley, such that it doesn't really matter that much where you start. In many cases however, this assumption is not valid, and the cost landscape has many local minima in which the search algorithm may get stuck. Sometimes these local minima perform quite adequately, and there is no real issue, but it may also lead to a parameter set which performs very poorly.
Related to this problem, and probably even more common, is the issue of

problematic curvature. When the cost function leads to a very narrow and very steep ravine, with very high gradients on the edges, the finite step size of the algorithm will make the parameter set oscillate in between the two edges, and convergence will be either very slow or not happen at all. The true direction in which the cost would decrease is essentially masked by the high gradients at the edges.

Figure 5.2 illustrates both these problems. Notice how the finite step size in the left panel starts to matter, and the gradient at the edges of the valley completely dominates the search. Obviously, choosing a sufficiently small learning speed $\eta$ may solve this problem, but in reality the curvature can be so strong that it would need to be extremely small, leading to extremely slow convergence.

The issue of local optima can be largely resolved by using stochastic gradient descent or by introducing a so-called *momentum* term in equation 5.2 (Rumelhart et al., 1986; Qian, 1999). Stochastic gradient descent works in the sense that, due to the approximation, the gradient is noisy. This means that it essentially 'hops around', and can easily escape a shallow local minimum. Momentum will make sure that the parameter evolution will have a certain inertia. Intuitively it behaves like a ball rolling around through the cost landscape, which allows it to overcome local minima.

Problematic curvature can be solved by using so-called second order methods. Rather than only considering the gradient, these methods will calculate the local quadratic curvature of the cost landscape and compensate for it. The success of this approach is striking, and usually the number of iterations needed for convergence is significantly smaller. The great downside is the fact that it is required to calculate the Hessian matrix, which is a square matrix with as many rows and columns as trainable parameters. Even small models often have several thousands of parameters, such that this computation quickly becomes impractical. Therefore there exist approximations that only look at the diagonal of the Hessian (Broyden, 1970; Goldfarb, 1970), and another method avoids calculating the Hessian altogether (Martens, 2010).

### 5.2.3 Backpropagation in neural networks

Here I discuss how gradient descent works for neural networks. The actual derivation, though rather straightforward, is quite involved, and has been explained thoroughly in literature. For details I refer to e.g. Bishop (2006). Here I only provide a brief practical explanation.

To start with, let us consider the MSE for $T$ data points $\mathbf{s}_n$ with associated one-dimensional target outputs $y_n$. If the model we wish to train gives output $\tilde{y}_n$, the MSE is equal to

$$\text{MSE} = \frac{1}{T}\sum_n (\tilde{y}_n - y_n)^2. \tag{5.3}$$

Taking the gradient to the parameters then yields

$$\nabla_{\boldsymbol{\theta}}(\text{MSE}) = \frac{2}{T}\sum_n \frac{d\tilde{y}_n}{d\boldsymbol{\theta}}(\tilde{y}_n - y_n) = 2\sum_n \frac{d\tilde{y}_n}{d\boldsymbol{\theta}} e_n \tag{5.4}$$

Here, $e_n$ is the error of the output of the $i$-th data point. Let us now consider the case of a single-layered neural network as an example. The output is given by $\tilde{y}_n = \mathbf{U}[\mathbf{a}_n; 1]$, $\mathbf{a}_n$ being the hidden state associated with the $n$-th input data instance. The derivative to the parameters $\mathbf{U}$ is then simply the hidden state plus bias. We find that

$$\nabla_{\mathbf{U}}(\text{MSE}) = \frac{2}{T}\sum_n e_n[\mathbf{a}_n; 1]^\mathsf{T}, \tag{5.5}$$

which already yields the updates for the output weights. Next we wish to calculate the derivative to the input weights $\mathbf{V}$. We find

$$\begin{aligned} \frac{d\tilde{y}_n}{d\mathbf{V}} &= \frac{d[\mathbf{a}_n; 1]}{d\mathbf{V}} \frac{d\tilde{y}_n}{d[\mathbf{a}_n; 1]} \\ &= \mathbf{D}(\mathbf{a}_n)\mathbf{U}_0^\mathsf{T}\mathbf{s}_n^\mathsf{T}. \end{aligned}$$

Here, $\mathbf{D}(\mathbf{a}_n)$, is a diagonal matrix with as diagonal elements the derivatives of the activation functions of the hidden state, in the case of hyperbolic tangent units the diagonal elements are $1 - a_i^2$. We use $\mathbf{U}_0$ as notation for the output weights without the bias elements. The input weights $\mathbf{V}$ are then updated with

$$\nabla_{\mathbf{V}}(\text{MSE}) = \frac{2}{T}\sum_n \left\{\mathbf{D}(\mathbf{a}_n)\mathbf{U}^\mathsf{T} e_n\right\} \mathbf{s}_n^\mathsf{T}. \tag{5.6}$$

The quantity between curly brackets we call the *backpropagated error*, as it is the error projected over the transpose of the output weights, as if it is projected onto the neurons in the hidden layer. Generalizing to multi-dimensional output signals is quite straightforward. We can simply replace $e_n$ by a vector $\mathbf{e}_n$ in the above expressions.
The backpropagation algorithm is often attributed to Paul Werbos (Werbos, 1974).

### 5.2.4 Schematic error backpropagation

Gradient descent for more complex neural architectures with multiple layers can be quite easily abstracted using the concept of a backpropagated error signal. Consider equation 5.6. The form of the equation is the column vector of the backpropagated error which undergoes an outer product with the input signal. The update rule for the output weights is essentially the same, but since they are linear, the derivative is simply the unity matrix. Furthermore, the 'input' of the output weights is the hidden state vector and bias: $[\mathbf{a}_n; 1]$.
In fact, this form of the update rule generally holds true for any layer in any conventional neural architecture. The weights between every two layers are updated in the same manner. I will skip the full derivation and only provide the resulting algorithm. Suppose we have a complicated neural architecture as depicted in Figure 5.3

- Run the network on a data instance. For each layer in your architecture, store the activations.
- Determine the error at the output.
- Propagate the error back through the network. Each hidden layer should have an error vector of the same size as the activation, which can be attained with the following rules:
  - Start at the output. Project the error to all layers that connect to the output over the transpose of their respective weight matrices (without bias elements).
  - Suppose there are connections from hidden layer $i$ to hidden layer $j$. The error of layer $j$ can be propagated to layer $i$ over the intermediate connections without bias elements.
  - If a layer gives output to several other layers, the above process is performed for all of them, and the backpropagated errors are added up.
  - The error on a hidden layer is complete when all backpropagated errors from all its outgoing connections are added. The resulting sum is multiplied with the derivative of the activations of the respective layer. Only then can the error be propagated further.
- Each set of weights now obtains its respective gradient by calculating the outer product of the errors on its outputs with its input vector, optionally with an additional bias term equal to one.
- Repeat the process for all data entries in the dataset and add up all the resulting gradients. Usually, for feedforward networks this can be

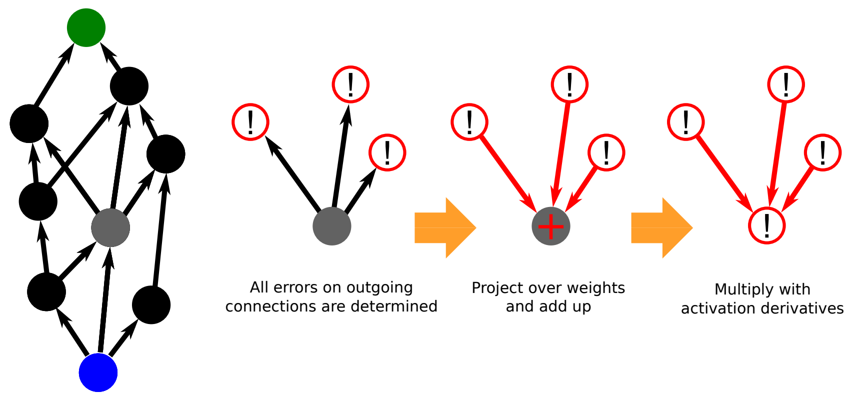
!
+
All errors on outgoing connections are determined
Project over weights and add up
Multiply with activation derivatives

**Figure 5.3:** Schematic depiction of error backpropagation. On the left I have depicted a feed-forward network with seven hidden layers, interconnected in a complex manner. The blue circle represents the input data and the green one the output layer. Each black (or grey) circle is a hidden layer, and each arrow represents a set of connection weights. The three left pictures take out the grey hidden layer and show how the error on it can be determined.

executed in parallel, and the whole set of calculations can be transformed into a number of matrix-matrix multiplications which can be computed very efficiently.

Any neural architecture that is described by a directed graph, i.e. where no feedback loops are present, can be trained in this manner with error backpropagation.

### 5.2.5 Backpropagation through time

In the case of an RNN we need to slightly change the above line of thought, and we need to think of the hidden layer in each instance in time as a separate entity. This allows us to unfold the network evolution in time, and we again obtain a directed graph. Figure 5.4 shows the graph associated with an RNN. The difference between this graphical representation and the previous one is of course that this time, the input, recurrent and output weights are no longer represented by a single arrow, but are repeated in the graph for each network iteration. We can still use the same reasoning, but this time we need to add up all contributions to the gradient from each step in the backpropagation. The idea to unfold an RNN in time to perform error backpropagation has

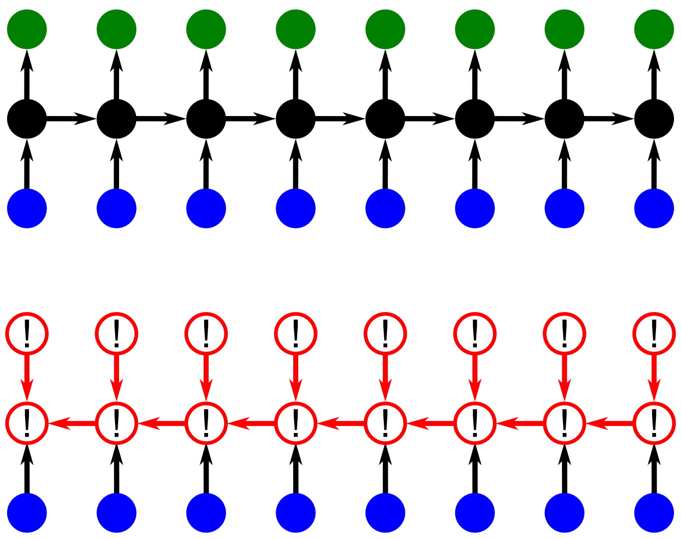

**Figure 5.4:** Schematic depiction of backpropagation through time. The top part shows the evolution of a recurrent neural network unfolded in time. The blue circles are the input frames, the black ones the hidden states, and the green ones the output frames. The bottom diagram shows the direction in which the error is propagated backwards through the unfolded network, which shows that it is propagated back in time.

been invented independently several times (Mozer, 1995; Rumelhart et al., 1986; Robinson and Fallside, 1987; Werbos, 1988), and is commonly known as *backpropagation through time* (BPTT).
Notice that a single data entry in the framework of BPTT is a time series, not a single input frame. If our data set consists of a single large time series, the proper way to obtain the gradient is to run the network on the whole time series, and then propagate the error back over the whole sequence. In reality it is more common to run the network on short, randomly sampled subsequences of the full time series, and apply stochastic gradient descent.

### 5.2.6 Common problems with training recurrent networks

Here I discuss three special challenges that appear when applying BPTT. Later in this chapter I propose a specific network architecture that tries to resolve both these issues at the same time.

#### 5.2.6.1 Vanishing gradients

One of the most prominent difficulties is to find long-term historic relationships in the data. If an error on an output frame is caused by input that has been inserted a significant number of frames before, the only way the BPTT-algorithm will find this connection is by propagating this error back in time, up to the frame where the relevant input instance was inserted in the network. The magnitude of the backpropagated error, however, will in many cases rapidly decline as it is propagated back in time. This means that by the time the gradient from the original frame has arrived at the relevant input frame it might have become too small in magnitude to be efficiently used.
One of the most successful approaches that tackles this problem is the so-called *Long Short-Term Memory* network (LSTM) (Hochreiter and Schmidhuber, 1997; Schmidhuber et al., 2002). Here, special gated linear neurons are present within the network, that can carry contextual information through time indefinitely. More recently it has been argued that the vanishing gradient problem is related to problematic curvature, and can be solved using second-order training methods (Martens, 2010).

#### 5.2.6.2 Bifurcations

The opposite situation, namely that of exponentially growing gradients, is also a potential obstacle for BPTT. This happens when the network suddenly has become chaotic. If we consider this from the original point of view, this translates in the fact that the cost landscape is not necessarily continuous. In Doya (1992) it is argued that an infinitesimal change in the parameters may lead to an abrupt change in dynamical behavior, called a bifurcation. In the cost landscape this means that the gradient descent may suddenly hit a nearly vertical wall. Here, the gradient can be very large, and the parameters make a sudden jump. If the dynamics of the network have switched into a strongly chaotic regime, recovery from this bifurcation may be very slow or not happen at all. Indeed, it is the experience of many researchers, myself included, that such bifurcations do occur, and sometimes with catastrophic consequences.

#### 5.2.6.3 Computational considerations

As mentioned earlier, in the case of feed-forward NNs, it is possible to process large amounts of data in parallel, as each data instance can be computed independently of all others. This is an advantage that partially disappears for RNNs. In order to compute the next hidden state we need to calculate

the current one first. Each of these operations is a matrix-vector multiplication, which is only computable in parallel to a limited degree.
It is possible to split up the training data set in shorter sequences, and run the RNN on each of those in parallel, still providing a significant speedup. Unfortunately, gradient descent using the full gradient is often prohibitively slow for RNNs, and in reality we often need to use stochastic gradient descent, trading gradient accuracy for increasing the weight update frequency. Stochastic gradient descent is truly not possible to compute in parallel, as the weight updates are fully sequential.

## 5.3 Simplifications: Training only the input and output weights

The content of this section has been published in Hermans and Schrauwen (2010c). I have explored a highly simplified variant of BPTT, in which I only train by propagating the error backwards for one step: from the outputs to the hidden layer, and only base the gradient on this, as if we take a batch length of a single frame. This allows to train an RNN during its operation. The main research question I wished to address was how well a classical RC setup, where only output weights are trained, holds up against the more advanced setup of training the whole system. I considered three training variants: training output weights only, training output and input weights, and training the full network. The setup where I train only the output and input weights has the inherent safety that bifurcations cannot occur, given that the network is inherently stable. Indeed, the inherent dynamics of the system do not change when the input weights adapt, only the kind of data that enters the network changes.
Training the input weights of an ESN has been considered before in Čerňanský et al. (2009). Here, the authors applied ESN on linguistic time series, where the input weight training is based on co-occurrence of input symbols. In my work however, the learning rule takes into account the inherent dynamics of the ESN, and will employ the interplay of the input projection and the network transients to improve the hidden-state representation of the input.
I have validated the performance of the three variants on a simple spoken digit recognition task. Not only have I evaluated performance itself, I have also investigated the way the structure of the input data is embedded in the network dynamics.

### 5.3.1 Network setup

All experiments were performed using RNNs of 50 neurons. I used nodes with a fermi activation function, and a leak rate of $\gamma = 0.2$. The output, input, and recurrent weight matrices, $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ respectively, were initialized with elements drawn uniformly from the interval $\{-1, \cdots, 1\}$, and then scaled by dividing them by the spectral radius of $\mathbf{W}$. Note that, due to the fact that I use fermi nodes instead of hyperbolic tangent nodes, a spectral radius of one is not associated with the dynamical stability of the network. The only reason I use it here is to have a convenient scaling measure. Scaling all three weight matrices with the spectral radius proved to be a good parameter initialization for this task.

### 5.3.2 Training setup

I use online training, which means that I update the weights each time step as I run the network on the input data. The update rules can be written as

$$\begin{aligned}\mathbf{U} &\leftarrow \mathbf{U} - \eta\mathbf{e}(t)\mathbf{a}^{\mathsf{T}}(t)\\ \mathbf{V} &\leftarrow \mathbf{V} - \eta\mathbf{D}(t)\mathbf{U}^{\mathsf{T}}\mathbf{e}(t)\mathbf{s}^{\mathsf{T}}(t-1)\\ \mathbf{W} &\leftarrow \mathbf{W} - \eta\mathbf{D}(t)\mathbf{U}^{\mathsf{T}}\mathbf{e}(t)\mathbf{a}^{\mathsf{T}}(t-1),\end{aligned} \tag{5.7}$$

where the elements of $\mathbf{D}$ are given by

$$\mathbf{D}_{ij}(t) = \delta_{ij}\frac{1}{2\tau}\left(1 - \tanh^2([\mathbf{Wa}(t-1) + \mathbf{Vs}(t-1)]_i)\right). \tag{5.8}$$

The update rules for $\mathbf{V}$ and $\mathbf{W}$ are in fact equivalent with the setup of training an Elman network (Elman, 1990). An Elman network is basically a feedforward network which has the hidden states of the previous time step as extra inputs. It is easy to see that an Elman network always has an equivalent recurrent network, and that propagating the error back one time step is equivalent with classic backpropagation in an Elman network.
The update rule for the readout weights converges to the same weights that would be obtained via linear regression, as the optimization criterion in both cases is the minimization of the MSE. When I tested performance of the three learning variants, however, it quickly became clear that only updating output weights leads to extremely slow convergence. For this reason I have opted to use a more advanced online learning strategy, called *Recursive Least Squares* (RLS) (Grant, 1987). This technique will make an online estimate of the inverse of the covariance matrix of the hidden state, which will significantly speed up the convergence.

### 5.3.3 Spoken digit recognition

To investigate the performance of our learning algorithm, I applied it to a spoken digit classification task. I used a subset of the TI46 isolated digit corpus where the digits from "zero" to "nine" are spoken 10 times by 5 different women (a total of 500 words). The resulting data was preprocessed using the Lyon passive ear model Lyon (1982), which initially produced 88 frequency channels. I reduced the data to 20 input channels and a sample rate of approximately 35 frames per word. Next, I randomly selected half of the words to be part of the test set, and the other half as the training set. The training datasets were then constructed by randomly sampling digits from the training set and separating them with intervals of 15 frames. Testing was performed by presenting all 250 words of the frames in a random order and measuring the classification error, which we call the Word Error Rate (WER). A new training and testing set was drawn for each network.

The readout layer consists of 10 classifiers where initially each classifier had a target output of 1 when their corresponding digit is uttered, $-0.1$ for the other classifiers, and zero for all classifiers during the 15 frame intervals. However, it appears that the networks have some trouble with this signal as a desired output signal, probably since it is virtually impossible to recognize a word at the very start of its utterance. This results in slightly worse performance and I suspect that this is due to the fact that the network will make wrong associations at the beginnings of the words. To reduce this effect, I have low-pass filtered the target output with the leak rate of the network, effectively slightly softening and delaying the desired output. Put in a formula: if $\hat{\mathbf{y}}$ is the original desired output, we produce the actual desired output by calculating $\mathbf{y}(t+1) = (1-\gamma)\,\mathbf{y}(t) + \gamma\hat{\mathbf{y}}(t)$.

Classification is finally performed by taking the mean of the output signals for the duration of the utterance, and selecting the digit which corresponds to the channel with the highest mean output.

### 5.3.4 Experimental setup

First of all I measured the evolution of the WER for each of the three setups. To monitor the progress of the training algorithms, I froze all the weights after every 1000 presented digits and measure the WER on the test set. In total I have initialized and trained 50 networks for each strategy, and show the average WER as a function of the number of presented digits.

Second, I have investigated whether or not the network internally adapts specifically to the task, i.e., whether or not the internal representation of the digits depends more on the digit itself than its specific utterance. For this purpose I have measured the centroids of the reservoir states during each

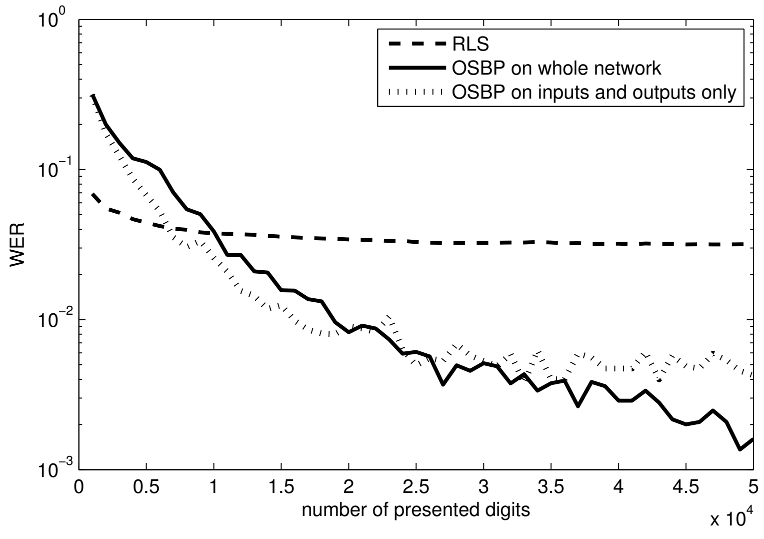

**Figure 5.5:** WER of the three training setups as a function of the number of presented digits. Results are averaged over 50 network initializations. OSBP stands for *One Step BackPropagation*, which describes the nature of the training algorithm.

word (the mean over time of the network activation during the utterance of a word), which I denote $\mathbf{q}^k$ for the $k$-th word. I have investigated the clustering of these before and after training $\mathbf{V}$ and/or $\mathbf{W}$. As a distance measure[2], I used $D = 1 - r$, in which $r$ is overall correlation between the centroids, i.e., the distance between the $k$-th and $l$-th word is given by

$$D_{kl} = 1 - \frac{\sum_{i=1}^{N_d} (q_i^k - \bar{q}^k)(q_i^l - \bar{q}^l)}{\sqrt{\sum_{i=1}^{N_d} (q_i^k - \bar{q}^k)^2 \sum_{i=1}^{N_d} (q_i^l - \bar{q}^l)^2}}, \tag{5.9}$$

where $\bar{q}^k = {N_d}^{-1} \sum_{i=1}^{N_d} q_i^k$ and $N_d$ the total number of digits.

### 5.3.5 Results

Basic performance results are shown in Figure 5.5. It appears that the RLS algorithm very rapidly converges to optimal readout weights and gives a final performance of about 3% WER. However, it is clear that training all weights with the update rules from equation 5.7, though slower in convergence gives

[2]Euclidian distance is less meaningful in high dimensional spaces (Aggarwal et al., 2001).

superior final results. Remarkably, it seems that only training the input weights already gives a great improvement over random networks but with obviously less computational demands than training the full network.
Figure 5.6 gives an example of the dendrograms of the centroids of the network dynamics for the different digits before and after adaptation. Untrained networks seem to have very little to no tendency to cluster the different digits, suggesting that the dynamics of random networks depend more strongly on the specific utterance of the digit than on the actual digit. The learning rule automatically seems to cluster the data, trying to enforce similar trajectories for all digits within a single class while enlarging the separation between the classes.
It is interesting to note that the clustering of the centroids is also very good when only the input and output weights are trained. This can be due to two factors. First of all, the input mapping can already improve the clustering of the digits itself, i.e., the input mapping can try to find a projection which performs optimal spatial clustering of the input data (where each datapoint corresponds to the mean over time of the input data of a digit, i.e., the centroids of the digits). On the other hand, the input mapping could also take into account the dynamics of the network itself, where the temporal structure of the data will play a crucial role.
To investigate this we consider the clustering of the centroids of the digits and the centroids of the digits after the linear input mapping (i.e., the centroids of $\mathbf{Vs}(t)$). Examples of the resulting dendrograms are shown in Figure 5.7. Though some improvement is apparent, the linear mapping alone cannot fully account for the nearly perfect clustering of the centroids of the network states. This means that the learning rule finds an optimal mapping from the input data to the network which also accounts for temporal information.
Apparently, finding an optimal projection into the state space of a random dynamic network - at least for digit recognition - already offers a very significant improvement over a random projection. Most importantly, training only the input and output weights of a network has the great advantage that problems such as bifurcation or loss of stability can no longer occur, since none of the recurrent weights are trained. The network itself can still be considered as a separate, stable-by-construction dynamic system which is left unchanged.
There seems to be a great advantage in using even a simple learning rule to adapt the input weights, and this is probably more true as input dimensionality increases. As is suggested in section 3.2, high-dimensional input data causes random networks to perform poorly in terms of memory. Likely a similar effect can be spotted in task performance. The network receives a random projection of the input data and will be sensitive for essentially

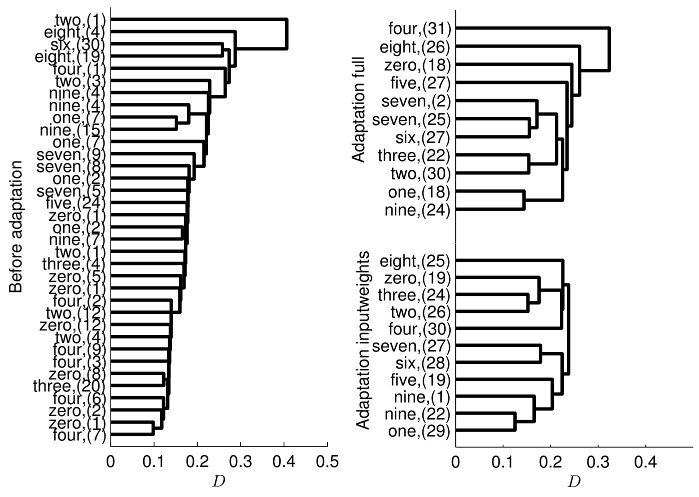

**Figure 5.6:** Examples of typical dendrograms of the centroids of the dynamics for different digits before and after adaptation. The written words are the digits, the number between brackets are the number of digits in each leaf.

random features in the input time series. The larger the input dimensionality, the smaller the odds are that these features will actually capture useful information. Training input weights can greatly improve this mapping, and allow the inherent network dynamics (which remain random in the reservoir setup) to be put to use far more efficiently.

## 5.4 Layered approach

Propagating the error back only one step may work in tasks that do not require a long history of the input data. For more challenging tasks though, it will not suffice, and the full BPTT scheme will need to be applied. In this section I elaborate on a specific RNN architecture, which allows for a very large number of parameters while remaining relatively fast to train.

### 5.4.1 Layered architectures

A large research domain focuses on training so-called *deep* neural architectures. These are feedforward networks, but with more than one hidden layer. There is a large body of evidence that supports the notion that large

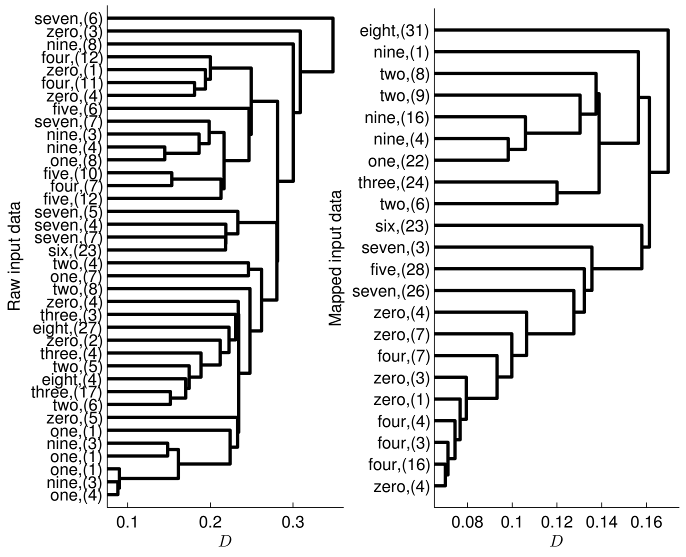

**Figure 5.7:** Examples of typical dendrograms of the centroids of the input data and the projected input data, the number between brackets are the number of digits in each leaf.

parts of the human brain are organized in hierarchical layers, and nowhere has this been as well-studied and understood as in the visual cortex (Chen, 1982). The image our eyes receives is sent through a series of increasingly high-level filters. The first layer of processing is already in the retina, where the image is compressed with a factor 100 in order to send it to the brain through the limited capacity of the optic nerve (Kuffler et al., 1953). Once the signal enters the first layer of visual processing, individual neurons are sensitive to local features that build up an image, mostly edges, spots and lines. Subsequent layers then recognize continually higher level features such as objects, shapes, etc. (Van Essen and Maunsell, 1983). Much higher in the cortical hierarchy, in the hippocampus[3], we know that there exist single cells or small groups of cells that encode highly specific concepts, such as the famous ‘Homer Simpson’-neuron that only becomes active when viewing or thinking about the animated TV-series ‘The Simpsons’ (Gelbard-Sagiv et al., 2008).
Machine learning too has benefited from hierarchical processing. In vision tasks, where the input consists of images and the goal is to recognize objects, faces, or handwritten text, hierarchical processing seems to be particularly successful (LeCun and Bengio, 1995). Still, training a feedforward NN with

[3]Seen by some as the highest layer in the brain (Hawkins and Blakeslee, 2004).

many layers poses some unique problems. First of all, as I mentioned it is important to initialize the parameters of the network more or less correctly. The larger the number of weight matrices defining the network, the more initial parameters will need to be set. This quickly becomes a difficult search. Secondly, error backpropagation in deep networks suffers from the so-called *fading gradient* problem (which incidentally also occurs in RNNs). This means that usually only the layers close to the output layer effectively 'learn' the desired mapping. When the error is propagated deeper down, it quickly decreases in magnitude and as a consequence, the weights in these layers barely change during training, and they barely play a useful role in processing. As is suggested in Erhan et al. (2010), the challenge lies in the fact that all layers are trained *simultaneously*. The role of deeper layers, closer to the input would ideally be preprocessing of the data, which can then be efficiently mapped to the desired output by the higher layers. However, as the higher layers are constantly changing during the training phase it is not obvious what this preprocessing exactly might entail, and gradient descent may never find it.

A new strategy to learn deep networks was developed, in which the actual training for a task is preceded by a phase of unsupervised *pre-training* of each layer individually. This idea has been put forward a few times independently, and the most notable two examples are *Denoising Autoencoders* (Vincent et al., 2008) and *Deep Belief Networks* (Hinton et al., 2006). Such strategies overcome many of the previous problems of training deep networks. The concept of the denoising autoencoder I will explain a bit more thoroughly, as it may have potential applications for the network model I will propose in the next section.

A denoising autoencoder will take an input instance which has been corrupted with a form of noise. It is then projected onto a hidden layer, and the resulting activations are projected back over the transposed input weights. The goal is to reproduce the uncorrupted data point, i.e., to *denoise* it, which can be performed with common gradient descent.

The easiest way the network can in fact reconstruct the input frame, is by learning the distribution of the input frames. More particularly it will encode likely features of the input data in its weight matrix. This makes that the representation of the hidden layer can be a compact way to specify a specific input instance, which makes it more 'useful' than that of a random network. The denoising part of the setup makes sure that the network is less sensitive to unlikely input features (the noise), and more so for more likely ones (which it needs to reproduce).

The next step is to perform the same process in the consecutive layers of an NN hierarchy. The hidden state of the first layer is corrupted, send to the

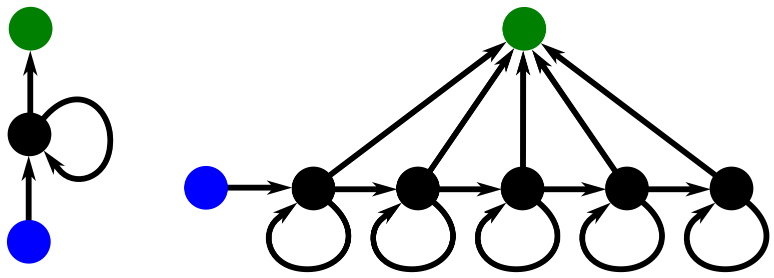

**Figure 5.8:** Structure of the LRNN. Blue circles are input and green ones output layers. On the left I have drawn a diagram of a common RNN, where the looped arrow represents a projection from the previous time frame. The right diagram shows an LRNN, acting like a multilayered feedforward network, but where each layer receives input from the previous hidden state. The output is connected to all layers.

second hidden layer, and projected back, and the second hidden layer needs to learn to restore the correct hidden state of the first layer. As such, a hierarchy of different layers can be built. After a sufficient number of layers has been stacked and trained in this manner, we obtain the pre-trained multilayered NN. This network can then be used as an initialization for a specific task applied on the input data, and can be trained further with gradient descent (known as the *fine-tuning* phase). The presence of both low and high level features in the hierarchy will provide a very good starting point for common training strategies, and has been proven to be an excellent method to boost performance in e.g. written digit recognition (Vincent et al., 2010).

### 5.4.2 Layered RNNs

Extending the idea of deep learning to a recurrent architecture has been attempted to some level (e.g., the recurrent temporal Restricted Boltzmann Machine (Sutskever et al., 2008)). What I explored in my research is to see how well a rather simplistic hierarchical recurrent architecture might score compared to a common RNN in terms of performance, computational cost, and stability. In Figure 5.8 I have provided a schematic depiction of the *Layered Recurrent Neural Network* (LRNN). Each step in time, each layer receives input from the hidden state of the previous layer like in a multilayered feedforward NN, and each layer also has its own recurrence. If we denote the hidden state of the $i$-th layer with $\mathbf{a}_i(t)$, we can write the update

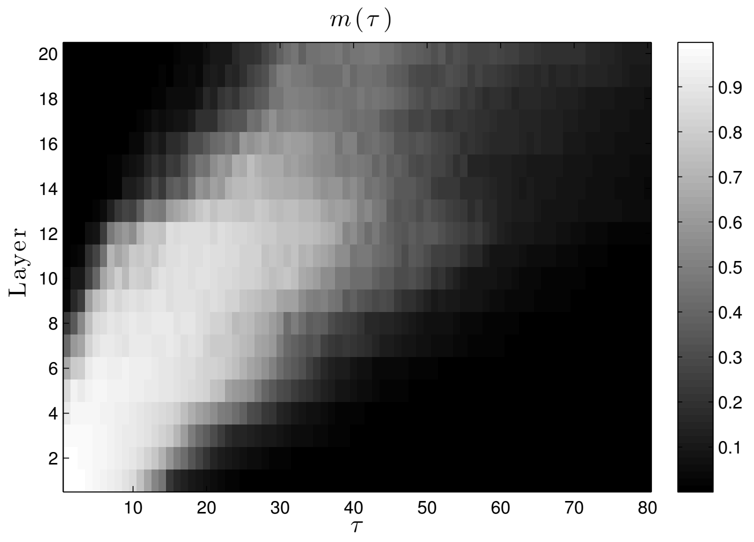

**Figure 5.9:** Linear memory functions for each layer in an LRNN with 20 layers. The input signal consists of one-dimensional i.i.d normally distributed noise. The number of nodes in each layer is equal to 50. Each recurrent weight matrix $\mathbf{W}_i$ has a spectral radius of 0.95, the input layer $\mathbf{V}$ is initialized with input scaling $\zeta = 0.2$, and each of the intermediate weight matrices $\mathbf{Z}_i$ is initialized with elements drawn from a standard normal distribution, and next rescaled with a factor 0.07. The measurement of the memory function is performed using 250,000 input frames.

equations as

$$\mathbf{a}_i(t+1) = \tanh(\mathbf{W}_i\mathbf{a}_i(t) + \mathbf{Z}_{i-1}\,[\mathbf{a}_{i-1}(t+1); 1]) \quad \text{for } i > 1$$

$$\mathbf{a}_1(t+1) = \tanh(\mathbf{W}_1\mathbf{a}_1(t) + \mathbf{V}\,[\mathbf{s}(t+1); 1])$$

$$\tilde{\mathbf{y}}(t) = \mathbf{U}\,[\mathbf{a}_1(t); \cdots ; \mathbf{a}_L(t); 1]\,.$$

Here, $\mathbf{Z}_i$ is the set of weights that projects the $i$-th hidden layer onto the $i+1$-th, $\mathbf{W}_i$ are the recurrent weights of the $i$-th layer, and $L$ is the number of layers. I have opted for letting the output weights tap into all the hidden layers, not just the last one. Not only does this give the output layer more information to work with, it also partially helps the problem of fading gradient as each layer directly receives a backpropagated error.
I have designed the LRNN with some perceived advantages in mind. I will list and briefly discuss these here.

- **Computational speed**
  Undoubtedly one of the greatest advantages of the LRNN over a common RNN is the fact that the training can be sped up quite significantly. The computational bottleneck in running an RNN is the matrix-vector multiplication of the recurrent weights with the hidden state. This process is slow as it cannot be computed in parallel. With the LRNN this problem is partially solved. We still need to run the recursion for each layer, but the hidden states that are obtained this way can be projected onto the next hidden layer in one step, which can be fully run in parallel. As I will show later, this leads to a speedup of roughly a factor 4.

- **Bridge between delay line and fading memory**
  As I have discussed in Chapter 2 and 3, the hidden state of an RNN is a function of the fading history of the input signal. In the case of LRNNs, each hidden layer will provide a successively broader view over the input signal, as it can be said to convolve its own memory with that of previous layers. This may lead to a naturally appearing hierarchy, where lower layers process short text features (at e.g., the level of syllables), and higher layers will model features that require longer memory (up to the level of complete sentences).
  In Figure 5.9, I have depicted the linear memory functions for the hidden state of each layer of an LRNN with 20 layers. More details are provided in the caption, but what matters is that indeed, each layer has a view on the past input that progressively broadens and lies further in the past as the layers are higher in the hierarchy. If a large number of layers is used, we could state that the memory of the network begins to resemble a delay line, i.e., each layer remembers input history further in the past.

- **Unloading the burden of nonlinearity**
  As I have explained in Chapter 3, RNNs have to obtain a balance between non-linearity and memory depth. For a trained RNN this is not completely true. If the recurrent weights are scaled up sufficiently, the mapping it performs can be both highly non-linear and depend on a long input history. Nevertheless I suspect that in such a regime, where the network really operates on the edge of stability, BPTT might quickly encounter difficulties with bifurcations.
  A layered approach has the advantage of stacking its nonlinear mapping the higher it goes into the hierarchy. This means that the hidden state of each layer does not necessarily need to be driven into the very non-linear domain. To put it shortly, the dynamics of each layer don’t need to be strained so much to reach a sufficiently non-linear mapping.

Another fact that holds true for LRNNs is that they *instantaneously* compute a hierarchical non-linear function of the input. As I explained earlier, deep architectures are desirable in the sense that they can efficiently represent complicated non-linear functions of the input data. Common RNNs are also "deep" in the sense that the non-linear mapping is iterated in time; generally the longer in the past an input frame has been presented to the network, the more times it underwent a non-linear transformation, and the more non-linear its current representation in the hidden state will be. In the case of text prediction we actually may need an *immediate* complicated nonlinear function of the input frame, which perhaps cannot be represented efficiently by a transfer through a single layer. The instantaneous transfer of the input data through multiple non-linear layers can potentially resolve this issue.

### 5.4.3 Wikipedia dataset

As I specifically built LRNNs with training speed in mind, I will validate their performance on a truly challenging task which requires a significant time to train: next character prediction for an English text corpus sampled from Wikipedia. This text is extremely varied, containing large numbers of names of places and persons, abbreviations, scientific jargon from a broad set of research fields, uncommon characters and generally rare words. Capturing such complexity needs a very large representational power, i.e., a model with a very large number of parameters. Not only is the set of words very large, the nature of text makes that a relatively long history is required to predict the next character.
The data for this task was collected and made publicly available[4]. It was first used in Sutskever et al. (2011), where the authors trained a special kind of RNN with multiplicative nonlinearities, for which the cost function has a particularly difficult curvature. They used an approximate second order gradient descent technique called *Hessian-free gradient descent* (Martens, 2010).

### 5.4.4 The dataset

The full dataset consists of a single large text file containing over 1.3 billion characters, roughly the amount of text you would find in about 2500 novels. I converted the dataset to a set of 95 character symbols (the ones that can be meaningfully represented in the Matlab console where the experiments were

[4]`http://www.cs.toronto.edu/~ilya/mrnns.tar.gz`

executed). They include small and capital letters, digits, spaces, punctuation and a number of arbitrary symbols such as the dollar sign, the ampersand, etc. All characters which were not part of this set, such as characters used in, e.g., the Cyrillic alphabet, were mapped on an additional 'unknown' character. The input signal consists of a 96-dimensional vector of zeros, except for the element corresponding to the current input character.

### 5.4.5 Next character prediction

The goal of the task is to predict the next character. Obviously, this is a non-deterministic process, so in reality the best we can do is predict the probabilities for the next character. In order to do so I use a softmax output layer, given by equation 2.7. To train a softmax layer, we do not use MSE as a cost function, but rather we wish to maximize the log-likelihood of the output of the correct label. The derivation is given in e.g. Bishop (2006). The result is that we do not need to take the non-linear nature of the softmax into account when performing backpropagation. We define the error vector $\mathbf{e}$ as the differences between the output of the softmax layer and the correct output, being a vector with zeros for all characters except for the correct one, which is equal to one. We can then use the same update rule for the output weights as we used to, and propagate this error back into the network.
I will use two measures of performance: first of all I simply will consider the character error rate (CER), which is the rate at which the output character with the highest predicted probability is the correct one, and secondly I shall measure the cross entropy between the network output and the text. This number expresses the average number of bits that are needed on top of the model prediction to correctly predict the next character, hence I call it *bits per character* (BPC). It is calculated by:

$$\mathrm{BPC} = -\left\langle \log_2(\tilde{y}_c(t)) \right\rangle_t ,$$

where $\tilde{y}_c(t)$ is the model's output for the correct next character, i.e BPC is the average of the negative log likelihood for the correct next character, as predicted by the model.

### 5.4.6 Experiments

I investigated computational speed and task performance on LRNNs with different numbers of layers, namely $L = 1, 2, 4, 6, 8, 10$. First of all I chose to have the size of all hidden layers within one model equal. Next, I made sure that the number of trainable parameters, i.e. the total number of weights in the models, is equal for all the models I trained. The total number of

trainable parameters was 5.400.000, a little more than the model researched in Sutskever et al. (2011) (4.900.000 parameters). This leads to a hidden layer size of 2236 neurons for $L = 1$, down to 513 for $L = 10$.
Due to the considerable computational cost of training these models, I only investigated one instantiation of each network model. This has the downside that we may accidentally draw a very bad or very good initial parameter set, which would obscure the true average performance. However I suspect that this effect is somewhat diminished due to the very large size of the models. In the end, when it became clear which of the models is in fact the most powerful, I have conducted an additional test, in which I initialized such a model, and next only trained the output weights (in exactly the same manner as when training the full model), to see whether the hierarchical RNN is merely a good prior, and can be used in the common RC framework.

#### 5.4.6.1 Parameter initialization

The input weights $\mathbf{V}$ were drawn from a normal distribution. All recurrent weights $\mathbf{W}_i$ and interlayer weights $\mathbf{Z}_i$ were drawn from a normal distribution and next divided by the square root of the number of nodes in the layer. This is based on the fact that the spectral radius of large, random matrices is roughly equal to the square root of their number of columns, times the standard deviation of their elements. This means that each network is initially roughly on the edge of stability.

#### 5.4.6.2 Training details

I used the whole Wikipedia dataset to train on, except for the last 10 million characters, which served as a test set. Each training iteration I chose a sequence of 250 characters at a random point in the training set. The 50 first frames of each batch were not trained on, and were only used to eliminate transient initialization effects.
The learning speed $\eta$ started at $\eta = 2 \cdot 10^{-4}$, and was reduced with a factor 0.9999 each 100 iterations, such that it gradually diminishes during training. Also, I halved the learning speed after $10^5$, $7 \cdot 10^5$ and $2 \cdot 10^6$ iterations, as I empirically found it to be beneficial to have a fast initial training phase and progressively slower ones in the later stages.

#### 5.4.6.3 Measurements

In order to keep track of the performance of the models during training, every 10,000 iterations I recorded the BPC and CER of the model, tested on the last million characters of the test set. This in order to have an estimate

of the true test error during the training phase. After training for about a month, I looked at the final testing errors on the whole test set.
I am also interested in the usage of each layer. In the worst case scenario, the model only trains the bottom layer and its corresponding output weights, and makes the rest of the network redundant. In the most interesting case, all of them would be comparably important. I will measure the BPC on the test set, while cutting off the output connections of one layer at a time and see how much the BPC increases. This will give a crude measure of the importance of each layer.

### 5.4.7 Results

In Figure 5.10 I show how the different models evolve during training. I have shown the BPC and the CER, both as a function of the actual training time and a function of training iterations, which is proportional to the number of training sequences the model is trained on. What is quite clear is the difference in computational speed between the models. The model with a single layer is over 4 times slower than the model with 4 layers.
Two models have stopped training prematurely: the model with two layers had halted due to a computer error, and its training could not be continued due to limitations in the available number of computers, and the model with 6 layers underwent a very strong bifurcation (leading to a CER above 0.5), from which it did not significantly recover. The results shown for the 6-layered model are those for the network right before the bifurcation.
What is apparent in Figure 5.10 is that sudden rises in BPC and CER are quite common, but are not always catastrophic. In the case for $L = 4$, which in the end performed best, we can see several distinct peaks, from which the training algorithm seemed to recover rapidly. Conversely, the model with $L = 8$ undergoes a bifurcation at roughly 650,000 iterations, and it never seems to completely recover from it.
Consider again the bottom row panels of Figure 5.10. When we only consider the training episodes of before 700,000 iterations, we can still make a clear comparison between the different models (before significant bifurcations occur or the training of certain models ends). From this evolution it is clear that the model with 4 layers converges the fastest, even when not considering the time required for each iteration. This means that there is a clear advantage to using a layered hierarchy of RNNs on this text generation task.
The evolution of the BPC and the CER are very similar. Therefore, I will only consider BPC in what follows, since the results are not qualitatively different.

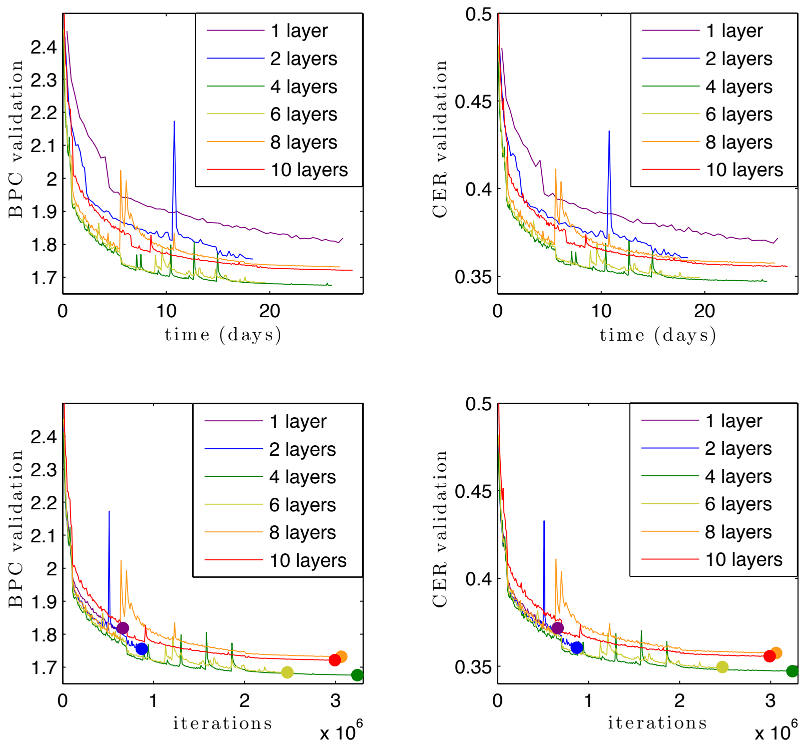

**Figure 5.10:** Convergence of LRNNs. The top row shows the evolution of BPC and CER of the small test set, as a function of time. The bottom row shows them as a function of the number of training iterations (corresponding to the amount of training data they have seen). In the bottom row I have indicated the end points of the lines with dots in their corresponding color. The results shown for the model with six layers is that what was saved last before it bifurcated. The sudden drops in performance correspond to the moments where the learning speed was halved.

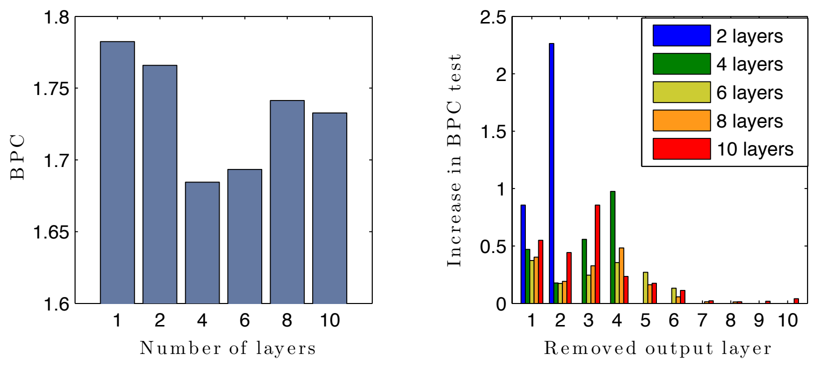

**Figure 5.11:** Left panel: BPC on the test set for the different final models. The case of six layers was tested on the model that was last saved before the bifurcation. Right panel: decrease in BPC of the test set when eliminating output of one layer. Results are shown for all models with more than one layer. The results shown for the model with six layers is that what was saved last before it bifurcated.

In the left panel of Figure 5.11 I show the final BPC on the test set for each of the models. It appears that the model with $L = 4$ performs best, followed closely by $L = 6$ (which had less training time in total). The model with 1 layer performs worst, most of all since its total number of training iterations is significantly smaller than the other models.

In the right panel of Figure 5.11 I show how much the BPC reduces if I remove the output from a single layer in each of the models, which I use as a rough indicator of importance. Both the models with $L = 2$ and $L = 4$ have the strongest reduction in performance when the highest layer is not included in the output. This gives an indication that the model indeed learns a hierarchical model, where lower layer perform a preprocessing step for the last layer, which contains the highest level features of the input data.

For models with more layers, especially $L = 8$ and $L = 10$, the layers in the top of the hierarchy seem to matter far less. This shows that, if the hierarchy is too deep, most of the processing occurs in the lower layers, and the higher layers never succeed in performing useful computations of the input data, which means that a large part of the potential computational power of the model is not employed. Likely this problem can be partially resolved by leaving out output weights for lower layers, forcing the model to use the full hierarchy.

It should be mentioned that the LRNN still performs significantly worse

than the state-of-the-art on this task. In Sutskever et al. (2011), the final test BPC was 1.56, much better than my optimal result[5] of 1.68.
The model that was trained in the common RC setup had 4 layers and was initialized in the same manner as I described before. The performance, measured during training, never went under a CER of about 50 %, far worse than any of the trained LRNNs. Even when gradually lowering the learning speed, performance no longer improved. Even when we do not make the comparison to the trained LRNNs (which would not be justified, see next paragraph), this performance is rather poor. If I let it generate text (see next section), it mostly generated non-existing words, and some short and common ones such as "and", "the","of", and "with". There are two potential reasons for this poor performance. First of all, it may just be the case that RC is unsuited for a task like next-character prediction, simply because the problem is of such a strongly non-linear nature. Secondly, it may also be the case that the RC setup does not work well with untrained LRNNs. Perhaps a randomly initialized LRNN will produce features that are too non-linear and complicated to be put to use, and RC might score better with a single, large reservoir.
An honest comparison between RC and fully trained LRNNs would require that the number of trainable parameters is the same in both cases. Whereas the trained LRNNs have over 5 million optimized parameters, the model that only trains the output layers only has about 325,000: the readout weights. This means that an honest comparison would require a reservoir where the number of output weights is equal to 5 million, which would require over 56,000 neurons. Such a large network is not completely infeasible, as is shown in the work of Triefenbach et al. (2010) and Jalalvand et al. (2012), and it would be interesting to find out if in those cases one could still benefit from a layered hierarchy like that of the LRNN.

### 5.4.8 Generating text

Since the model output are probabilities, it is possible to sample a character from the output, and to use this as a new input character. Repeating this process then allows the network to generate text. Especially, we can first feed the model a fixed sequence, and next let it fill in the remainder.
In what follows I will provide some examples of what kind of text the model can generate, using the 4-layered model. In this I will use a question mark (which is altogether rare in Wikipedia) to indicate the "unknown" character. Here is a sample of text, generated using *It was a dark and stormy night.* as

[5]The absolute best test BPC I ever attained, using a 5-layered LRNN where I hand-tuned the learning speed during training was 1.65.

initialization:

> It was a dark and stormy night. In 2005 the "Wat it to Cave The Pearl" played nine years in the U.S. Senator on its restaurant, while still noticed by the armed The Player of the Procumentary primitive efforts on November 3, . ??As a job, the box officially unusually started his legendary loss down to Atlas by Iraq. InGovering a buses the heart of the film and its place, scientific separations found in battle. Around the two tests and the Bacho Seadame American archaeological church is only entailed in the 1820s.??When the performance is published by which Hartford carefully bounded and actual ILD releases with the supporters of peace systems. Admiral of the governments then appeared on the Graduation that easoned the Western existing $1.5 million for replacement plans outside the postmodel's new producers between North

The model can generate a broad variety of words and phrases. Nevertheless, the text only appears coherent for a history of a few words, and grammatically correct sentences are rare. All in all, the model only produces a few non-existent words, and some of those are capitalized. Indeed, in actual Wikipedia texts, capitalized words are often names of places or persons, and they are far too diverse to be memorized by the model.
The next test I performed was to check whether the model is able to close quotation marks and brackets, for which I used *Known as the “* and *What followed next (* as initialization, two fairly generic phrases, meant to initialize the network. I noticed that in slightly over 80 % of the cases, the model finished the quotes in a satisfactory manner, meaning that the closing quotation mark appeared directly after a letter. If there was a space or a punctuation mark between the last word and the closing mark, I interpreted this as if the model opens a new quotation. It fills the quotes mostly with short, sometimes capitalized, phrases, which are interesting in their own right. Here are some examples:

> “British Nordware”, “Basilis”, “publisher James Days”, “Christian Eaglet”, “Super Bowl? 8th Press”, “Old War Detroit”, “Yuan Duchi”, “House Storm”, “Enterprise Centre”, “Sirsset Gots Christe”, “Northumberland”, “Riyal Observer and Fernando Royal School”, “senses back re-related women”, “Luxembourg”, “agricultung and to the Dita's Republic of Versailles”, “Rudish Park”, “Oregon of New Jersey”, “first particle murders”, “Club of Ed”, “Power Holton”

Brackets are more difficult, as in normal text they often contain longer phrases. If the intermediate text is too long, the model will forget the opening bracket during progression over the text, and not learn the fact that they always need to be closed. A similar test as before reveals that in about 70 % the model closes the brackets in a satisfactory manner. The phrases are somewhat similar to the phrases between quotation marks, but also include statements that can be interpreted as relative clauses (which are indeed often found between brackets). Here are some examples:

(courage Bear Pear XIII), (7), (streetcar), (ABRCA), (later = in Dunt), (500 m), (except), (1931), (the intensity of memory), (after work in New York Catholic), (according to nhooters), (Wellesley;), (in the Human Richtermann in Boat), (11 and 3,197 mD), (Williamstown), (PPC), (which provoked guilt as part of the activ), (diesel boas), (2002), (1st heaven), (200 ED0), (one of the masters' 6 is illustring), (played by a FT), (which also did the numerable recovered)

## 5.5 Conclusions

In this chapter I have bundled results of my research into backpropagation through time. I have investigated the effect of introducing a highly simplified variant of BPTT, where I truncate the the propagation to only a single step, which allows for an easy online learning rule. I researched three variations: the classic RC setup, where only output weights are trained, a setup where only the input and output weights are trained, and a setup that trains all parameters. The task I considered was a spoken digit recognition task.
I showed that training both in- and output weights performs significantly better than only training output weights. Training the full network performed still slightly better, but it included the chance of bifurcations. I also showed that training the input weights will provide a good clustering of the spoken digits, which means that even a simple learning rule to adapt the input weights can provide a great increase in performance. This suggests that for high-dimensional data reservoir computing can greatly benefit from learning the input weights.
Next, I designed a layered hierarchy of RNNs. It has been suggested in past research that feedforward neural networks with multiple layers may encode certain non-linear mappings more efficiently. Therefore I extended the concept of multilayered networks to recurrent networks. In order to train these architectures efficiently, I connected each layer to the output. This also provides some insight in the importance of each layer, as we can measure the decrease in performance when we disconnect a single layer.
I applied the LRNNs to next character prediction, trained on a large corpus of Wikipedia text, which is a very difficult and highly non-linear task, requiring a lot of representational power. I showed that the LRNN has as main advantage that it speeds up the training process more than 4 times compared to a normal RNN, allowing for significantly more training iterations. Furthermore it appears that, especially for the case of 4 layers, the model outperforms a common RNN, even when not considering the time required for training. Finally, I showed that applying too many layers in the

LRNN (models with 8 and 10 layers) is detrimental, as the higher layers are not efficiently used.

## 5.6 Future work

Here I discuss what the potential extensions of my research in BPTT are.

### 5.6.1 Advanced training methods

In Sutskever et al. (2011), the performance reached on the Wikipedia character prediction task far surpasses the best I ever obtained. They applied a specific kind of network which they claim is particularly unsuited for common gradient descent techniques. In fact, this claim is hard to check, as it is particularly slow to perform stochastic gradient descent on their model[6] Still, they reach very impressive performance with only 160 weight updates using a Hessian-free training method. This raises the question whether it is their model or the training method that leads to such impressive performance.

For this reason it would be highly informative to train LRNNs using Hessian-free gradient descent. The computational power required for this is rather high, but as each weight update is based on the full dataset, it can be computed in parallel far more readily than stochastic gradient descent. Using, e.g., a cluster of GPUs, as the authors of Sutskever et al. (2011) did, would make this quite feasible.

### 5.6.2 Pre-training: deep learning for RNNs

Recurrent neural networks are a particular kind of deep network, where the depth is the recursion through time. Successfully applying deep-learning techniques on RNNs is currently a hot topic of research[7], but seems to remain challenging. There are many potential ways one can interpret deep-learning strategies for RNNs, and it is unclear which one, or which ones, are correct. The LRNN poses a new platform on which such deep-learning strategies can

[6]Roughly stated, this model has a different recurrent connection matrix for each input character, which is defined by a matrix-matrix multiplication. For a limited character set, all these matrices can be computed in advance, a process that needs to be repeated every time the weights are updated. For the Hessian-free technique this is no problem, as the number of training iterations is particularly low. For stochastic gradient descent, however, there are many millions of iterations, which would render this model prohibitively slow to train.

[7]Which I know from informal communications.

be applied. These models are not only deep in the temporal sense, they are also spatially deep, i.e., they represent a deep hierarchy at each single frame. It is known that - even when they are potentially very powerful - deep models cannot always be trained efficiently, due to problems such as a fading gradient. Can we find efficient ways to pre-train such models by extrapolating idea from the deep learning community? Can such a pre-training phase produce powerful spatio-temporal features of the input data? Suppose we would, e.g., apply the framework of denoising auto-encoders on an RNN, and consider the network, unfolded in time as a multilayered network. What would we denoise? The current input frame, the previous hidden state, or both? Do we then need to feed the network with the noisy data, or only apply the noisy data for the last frame (such that it can use ‘clean’ context from before), etc. The potential ways to incorporate this concept in RNNs are indeed quite diverse. For LRNNs, the problem becomes even more convoluted. If we unfold it in time, the hierarchy runs in two directions: in time and up the layered hierarchy. Finding good pre training schemes for this architecture seems like a tantalizing challenge.

### 5.6.3 Architectural priors

When we apply a sophisticated network model like an LRNN, we implicitly assume that the data can be efficiently model by such a system. One could say that an LRNN is an *architectural prior*, i.e., the prior assumption on the nature of the data is encoded in the specific architecture that we use.
The LRNN is but one of many potential temporal hierarchical architectures. A systematic investigation to how such architectures can influence training and modeling power would be highly informative. Especially for recurrent systems, a good architectural prior can impose very specific dynamical behavior on a model, which may greatly aid training algorithms to find a suitable solution for an application.
Obviously, searching through the space of possible structures for such architectures is a computationally demanding undertaking. Nevertheless, with recent advances in computing power, and mainly the application of GPUs and similarly massively parallel computers, such a search may be feasible. Recently, great success has been achieved in this domain by searching through a large parameter space for a visual recognition architecture (Pinto et al., 2009). They showed that applying a large-scale brute force search for good parameters can yield solutions that beat state-of-the-art by a wide margin. Such a search, applied not just on parameters, but also network architectures, like LRNNs, might yield very powerful solutions to difficult problems like next character prediction.

### 5.6.4 Reservoirs, hierarchies and training algorithms

The final research question that I pose here is the following: are hierarchies suitable to be incorporated in the framework of RC? As I mentioned before, an LRNN used as reservoir performs much worse than a fully trained network. In principle, for such a task we would need a far greater number of neurons to make a fair comparison. It would be for instance possible to make very large, very sparsely connected ESNs, and use gradient decent to train a softmax layer as output. It is also possible to apply the same framework as LRNNs to this problem: for instance by taking 5 layers of each about 11,000 neurons. Still, even though a layered hierarchy may produce more complex features of the input data stream, in the framework of RC these will still be random, and it is yet to be proven whether a hierarchical setup would have any benefits at all. Perhaps unsupervised pre-training may prove a powerful RC strategy to employ hierarchical reservoirs.
From a broader perspective, it remains an open question to pinpoint exactly what reservoirs are good at. Even though I have given some suggestions at the start of this chapter, we still don't know if untrained, random DSs are fundamentally limited in their application domain, or whether we could simply find better prior assumptions on parameter distributions, such that they could also be applied on difficult, highly non-linear tasks that require a lot of representational power.

# 6 Conclusions and Future Perspectives

In my final chapter I will give a brief overview of my field of study, reiterate the most important contributions of my work and provide a spectrum of potential follow-ups for my research. I will largely try to follow the structure of the dissertation, in the sense that I discuss its three main research topics: memory, recurrent kernels and training algorithms.

## 6.1 Summary

Reservoir Computing (RC) is a promising Machine Learning strategy for processing time series. It provides a method to employ large non-linear dynamical systems to perform useful computations on an input time series. The field of RC is an amalgamate of training strategies that have emerged from a diverse set of angles. The main idea is that we can use essentially random dynamical systems as models. All that needs to be explicitly optimized is the so-called *readout*-layer, which is typically a direct linear projection of the internal state of the model. This is particularly fast and easy to train.
The fact that random dynamical systems can be such powerful computational entities came as a surprise, and has sparked a whole new field of scientific exploration. This thesis addresses some of the more theoretical questions that followed from this endeavor. Two questions in particular have been explored in depth, namely: *how does a reservoir keep track of the history of its input* and *what happens if we make reservoirs infinitely large.*

## 6.2 Research conclusions

Here I will state the main conclusions that follow from my research, grouped by theme.

### 6.2.1 Memory

The first main theme of this dissertation is on how dynamical systems can store a fading history of their input signal. I have applied the framework of linear memory capacity on two important reservoir computing schemes: that where the input signal is high-dimensional, and that where the model operates in continuous time.

I have shown that many of the well-known conclusions for linear memory capacity hold true in the high-dimensional case, namely that the total memory capacity is no higher than the number of neurons in the network and that increased non-linearity will deteriorate it.

Most real-world high-dimensional signals have a broad power spectrum. When braking down the signal into uncorrelated components, it quickly becomes clear that most of the signal variance (which we call *power*) resides in a few channels, and the power of the other channels falls quickly to very low levels. Usually, the bulk of useful information is present in the principal components with the highest power, and the ones with lowest power carry only noise. When we insert such a signal into a reservoir, we would like the available memory capacity to be distributed over the components roughly in proportion to their power.

I have shown that, in random networks, this is not the case. The low-power channels will hog a disproportionally large portion of the available memory capacity, and as such they can impair the proper functioning of the reservoir. If we consider the low-power channels as noise, it turns out that random networks are quite sensitive to noise. When I applied orthogonal networks as reservoirs, this problem was solved, confirming the well-known result that they are superior w.r.t. noise robustness.

The message for researchers applying reservoir computing to high-dimensional data is that they may benefit from either using orthogonal networks, or using only a select set of principal components of their input data.

For the second part of my study of reservoir memory, I have extended the concept of memory function and memory capacity to the continuous time domain. Continuous time models are an important subdomain of RC, particularly since one of the most common implementations involves leaky integration, a discrete-time approximation of a continuous time dynamical system. I have investigated three distinct network setups, and see how well they per-

formed in terms of noise robustness.
The first conclusion I drew was that random networks score particularly poorly. The set of filters that a random network provides in discrete time is quite unsuitable in the continuous time domain. Therefore I transformed this set of filters in a mathematically correct way, using an inverse $z$-transform. These networks have a strongly improved robustness against noise.
Finally I designed a continuous time counterpart of orthogonal networks. Essentially they consist of a set of damped oscillators with different frequencies but identical decay rates. I show that the way in which they store information on input history is similar to encoding a signal by its Fourier components, a particularly efficient way to store information on a signal. As a consequence, these networks are very robust against noise, and keep a high memory capacity even when the signal-to-noise ratio approaches one.
Finally, I have discussed the limitations of linear memory capacity, and briefly presented work (not from myself) that provides a non-linear extension of memory capacity. It models higher-order dependencies and is far less constrained than linear memory. I suggest another alternative way of interpreting memory in reservoirs, using the Jacobian. This method allows us to visualize memory through time. I also show that we can view the degree of non-linearity as the variability of this Jacobian, i.e., the more non-linear, the more a reservoirs sensitivity to past input changes over time.

### 6.2.2 Recurrent kernels

The second theme of my dissertation is the unification of kernel machines with Reservoir Computing. I started from the existing framework of feed-forward networks with an infinite number of hidden nodes. Even though it is not possible to explicitly define such a construct, it is possible to compute the inner product of two hidden states associated with two different input instances, given a prior on the distribution of the input weights. This inner product is a kernel function. It allows to perform linear regression of the infinite hidden state, given that there is a finite number of training examples. The training data is then directly incorporated into the end solution.
I show that it is possible to extend this framework to recurrent neural networks. In fact, for any kernel function that is a function of the inner product and norms of the input instances, it is possible to define a recurrent version. The input of these recurrent kernels is no longer a static data point but a time series; the recurrent kernel takes two finite time series and provides a series of kernel functions, each corresponding to the current inner product of the hidden states. When we use such a kernel in a standard kernel machine we shall typically use the final kernel value as an output value.

I provide a broad set of examples for recurrent kernels. For the most common Echo State Network implementation, with hyperbolic tangent nodes, the solution is analytically intractable, but for networks that use an error function as nonlinearity, which is highly similar, the kernel has an explicit solution. I also provide recurrent versions of the most popular kernels in literature: the linear, polynomial and Gaussian RBF kernel. Finally I derive a kernel function that is associated with infinite sparse threshold unit networks, an important academic model for a broad number of natural and manmade phenomena.
After having provided the expressions for recurrent kernels, I study how they behave dynamically as a function of their parameters. I show that, in the case of the kernel associated with error function nodes, it is possible to associate the concept of spectral radius with the standard deviation of the weight distribution of the infinite recurrent weight matrix. In this sense, the well-known result from Reservoir Computing that asymptotic stability requires a spectral radius smaller than or equal to one, remains true for infinite networks.
For other types of kernels a linearization around the origin is not practical. I have found a way for certain types of recurrent kernels to calculate their Lyapunov exponent, which also takes into account the influence of the input signal. Using this, it is possible to compute the Lyapunov exponent of infinite sparse threshold unit networks and recurrent Gaussian RBF kernels.
Finally, I study the practical applicability of recurrent kernels. I consider two tasks: firstly an academical example which I use to illustrate how kernel dynamics relate to real Echo State Networks, and secondly a real-world speech recognition task. For the first task I compare the performance of recurrent kernel machines with their finite counterparts. I show that, for a given amount of training data, the average performance on the task will increase for increasingly larger networks, and finally it converges to the performance of the associated recurrent kernels. In the second task I compare recurrent kernels with the more classic windowed approach. I show that not only recurrent kernels perform better, they also select a smaller number of support vectors, which illustrates that fading memory is a better prior for speech data than a fixed window.

### 6.2.3 Training recurrent networks

The final part of my research considers training algorithms. Most of the research in this chapter is quite preliminary, and due to the nature of training algorithms for recurrent networks, it is difficult to draw strong conclusions. Nevertheless I have tried out some strategies which are worth to mention in my dissertation.

First of all I consider a highly simplified variant of the typical backpropagation-through-time algorithm. Here I adapt the network and input weights by only propagating the error back one step. I then apply this strategy to a simple spoken digit classification task. I compare three situations: one where I only train the output weights, conform classic Reservoir Computing, one where I train input and output weights, and one where I train all the weights. As is to be expected, the full training works best. Nevertheless, the second strategy is almost as good, and it has the inherent advantage that the network dynamics cannot suddenly become chaotic. All that changes is the input projection. If the network is stable by construction it will remain so.

Next, I consider the situation in which the task is highly nonlinear, *and* requires a long history of the input. I argue that common RNNs may not be very efficient in modeling an instantaneous strong non-linearity of the current input frame, which may in some cases be required.

I propose a partial solution to this problem by stacking recurrent neural networks. Each layer is a recurrent network, and their input consists of the hidden state of the previous layer (except for the first one, which receives the input data). I argue that the activations of each layer do not need to be as highly nonlinear, due to the fact that the nonlinearities are stacked. As such, the network as a whole may find an easier balance between memory and nonlinearity. This model also makes use of the known fact that certain non-linear functions can be represented more efficiently using multiple layers. Each new input frame can instantaneously undergo a strong non-linear transformation, which in classic RNNs is more difficult.

I test this architecture on a good real-world example of a highly non-linear task that requires a relatively long input history: next character prediction in text. I use a text corpus which is assembled from English Wikipedia articles. I test the performance of the task using models with different numbers of layers.

The results indicate that networks with four layers perform better than the common RNN setup. Not only can they be trained significantly faster in terms of computational costs, but they also perform better, even when not considering the time required for training. Models with larger numbers of layers perform less well, and I show this is due to the fact that the highest layers in the hierarchy cannot be trained efficiently.

## 6.3 Future directions

Throughout my four years of study, many questions have been left unanswered, and many doors to new research paths have been brought into view. Here I will provide an overview of them, again grouped by the three main themes, put in the form of questions.

### 6.3.1 Memory

As far as Linear Memory Capacity is concerned, I believe research in this field has been more or less exhausted, and this part will deal with questions on non-linear behavior. I will list my main unsolved research questions here.

- **Is it possible to find a useful measure of history dependency?**
  What is most interesting at the moment is a better understanding of non-linear memory, i.e. we do not look at how well input can be linearly reconstructed from the hidden state, but we ask how much the current hidden state *depends* on the input history. The term 'dependency' is still rather vague. I have proposed a possible measure of this, being the Frobenius norm of the Jacobian, which allows an instantaneous view on the 'sensitivity' of the current hidden state on past input. Though illustrative, all that is visualized is the derivative of the hidden state to past input. It does not give insight into the nature of the non-linear map, and a derivative does not necessarily give an adequate view on what we would understand under 'dependency'.
- **Can history dependency be shaped by well-chosen parameter priors?**
  If we would define a good dependency measure, we would in principle be able to search for network parameters that provide specific dynamical behavior. One of the conclusions we could draw from the Jacobian sensitivity measure is that the dependency history of a highly nonlinear network seems to be variable. Would we be able to find a network setup in which we would have control over this variability; giving the network e.g. a long memory at one time and a short one in others.
- **Can history dependency be used to visualize the underlying nature of a task?**
  Suppose we have trained a recurrent network for a specific task. If a good measure would exist, it would in principle allow a researcher to visualize the historic dependency for a specific task. In data of which little additional knowledge exists, this could be used to show relations between input and output which were previously hidden.

### 6.3.2 Recurrent kernels

Lots of open questions and interesting challenges remain in the field of recurrent kernels. Here I provide a short overview of the ones I have considered in the end of chapter 4.

- **Can we accommodate recurrent kernels better than the standard kernel machine setup?**
  Computing a recurrent kernel is expensive. All intermediate kernel values have to be determined, and only the last value is used in the end solution. Current kernel machines assume that each kernel evaluation provides one output number. The representational power of recurrent kernels could potentially be greatly improved, if we allow for intermediate outputs to be integrated into the end solution, possibly greatly boosting the efficiency of the model. Multiple Kernel Learning may provide a potential solution here, as it is a field of study that tries to find task-specific kernels by linearly combining several different kernel functions.
- **Can we approximate the dynamics of the infinite hidden state via a simple recurrent model?**
  Each time a recurrent kernel machine needs to produce an output, it needs to pick a time window with the current input as last frame, and run the recursion. Compared to a recurrent model this is highly inefficient. A recurrent neural network only needs to do one evaluation for each frame of the input series, whereas a recurrent kernel machine with the same expressive power needs to do as many as there are frames in the time window.
  If we consider the output values of the kernel functions of each evaluation as a feature vector, we can essentially treat this feature vector as a hidden state. Would it be possible to find a recurrent model that mimics the dynamics of this hidden state? If it can be found we essentially find a very efficient way to make an approximation of the dynamics of the hidden state, and we can construct data-determined models that are far more efficient to compute than the recurrent kernels themselves.
- **Is the computational power of recurrent models due to fading memory or the recurrence?**
  Fading memory only implies the shape of historic dependency. It does not require a model to be recurrent. One of the most intriguing questions this thesis has raised is whether or not it is *only* the fading memory that is a good model for time series processing, or whether the recurrence (and hence the causality of the model) is the key to

their power. Presumably the answer will be highly task-specific. Nevertheless, a thorough comparative study to this question would be very informative.

- **Can the framework of recurrent kernels be extended to models of natural phenomena, continuous media and continuous time?**
  Recurrent kernels have to ability to abstract large dynamical systems that have a random element by assuming it is infinitely large, and we can average out the random element. This makes it suitable not only as a potential ML solution, but also as a way to study large-scale phenomena in dynamical systems. Can we use the recurrent kernel setup for, e.g., modeling the behavior of scale-free network phenomena?
  Related to this, a captivating research thread has been considering the non-linear dynamics of continuous media. Recurrent kernels have one thing in common with this scenario: namely that they also treat the neurons in the hidden layer as a continuum. Hence the question: can we find recurrent kernels that model the dynamics of such systems? The biggest challenge at the moment seems to be finding a method to account for continuous time. The math that is relatively straightforward in discrete time cannot be trivially transmuted.

### 6.3.3 Training algorithms

Even after a few decades of research, training algorithms for NNs are still a highly active research domain. RNNs pose a specific challenge in this field, due to special problems that appear from its recurrent nature. I will here list some of the most intriguing and promising lines of research that would directly follow from my research.

- **How much can advanced training strategies improve performance of the LRNN?**
  Common gradient descent techniques for RNNs are severely hampered by problematic curvature. This problem appears in any structure that needs to propagate an error through several non-linear layers, and in the case of LRNNs this problem appears twice: the errors need to be propagated both through time and through the layered hierarchy of the architecture.
  In my research I've only used stochastic gradient descent to train LRNNs, and quite possibly their potential is far greater than can be achieved this way. Therefore, the application of training strategies that are known to be able to deal with problematic curvature could be very informative to gauge LRNNs true computational power.

The choice of algorithm would be Hessian-free gradient descent. This training strategy is highly parallelizable, which makes it well-suited to run on massively parallel architectures such as GPUs.

- **How can deep learning strategies be applied on recurrent networks?**
  Other strategies to train deep neural architectures rely on a *pre-training* phase, where each layer is trained in an unsupervised fashion before actually training the whole system on a task. RNNs are a special case of deep networks, and it remains a challenging quest to introduce such deep learning strategies for recurrent models. In particular, the LRNN is an even better candidate for deep learning strategies, due to it being layered both in space and time.

- **What is the role of architectural choices and how can we automatically find them**
  Just like the parameters that determine a reservoir imposes a prior assumption on the nature of the task you wish to solve, so does a structured network such as an LRNN impose an *architectural prior*. A layered hierarchy can potentially work well if the input data is hierarchically structured, which is the case for data types such as text and images, which have challenging applications.
  An LRNN is one possible architecture from a large set of recurrent structures. We need to gather more insight into how an architecture can help to model certain types of data, and how their structure influences the ease with which they can be trained. Can we easily find architectures that are well suited for particular tasks? Can we significantly increase performance by performing a brute-force search through the space of possible architectures and parameters? These are questions that give rise to a riveting field of investigation.

- **Are there fundamental limitations to Reservoir Computing?**
  The final question I ask deals with the contrast between directed training algorithms, that optimize the whole model, versus Reservoir Computing approaches. I have argued that for some applications, basic RC setups such as ESNs may be fundamentally limited, as the odds of finding the correct kind of functionality in a random dynamical system may prove to be vanishingly small. Is it possible to resolve this issue by using different network types, different priors on weight distributions, or indeed, by imposing a well-chosen reservoir architecture, such that more useful features naturally appear within the reservoir's dynamics?

# A Appendix

## A.1 Memory for multidimensional input

### A.1.1 Compensating overestimation of the memory function

Here we will explain that the measured MF will approximately have a positive offset of $\frac{N}{T}$. We start by considering the principal components of the reservoir activation $\mathbf{a}$. If we write the eigendecomposition of the covariance matrix $\mathbf{C} = \mathbf{O}\mathbf{\Xi}\mathbf{O}^{\mathsf{T}}$, the principal components are given by $\tilde{\mathbf{a}} = \mathbf{O}^{\mathsf{T}}\mathbf{a}$. Using this expression and the fact that $\mathbf{C}^{-1} = \mathbf{O}\mathbf{\Xi}^{-1}\mathbf{O}^{\mathsf{T}}$, we can rewrite equation 3.5 as

$$m_n(k) = \frac{\left\langle s_n(i-k)\tilde{\mathbf{a}}^{\mathsf{T}}(i)\right\rangle_i \mathbf{\Xi}^{-1} \left\langle s_n(i-k)\tilde{\mathbf{a}}(i)\right\rangle_i}{\psi_n} \tag{A.1}$$

$$= \frac{1}{\psi_n}\sum_{j=1}^{N} \xi_j^{-1} \left\langle s_n(i-k)\tilde{a}_j(i)\right\rangle_i^2, \tag{A.2}$$

with $\xi_j$ the variance of the $j$-th principal component of the reservoir state. If the number of samples by which the covariances $\left\langle s_n(i-k)\tilde{a}_j(i)\right\rangle_i$ are estimated are finite, there will always be a positive overestimation of the memory function; We start by splitting the principal components of the reservoir states: $\tilde{a}_j(l) = \bar{a}_j(l) + \hat{a}_j(l)$, where $\bar{a}_j(l) = \left\langle s_n(i-k)\tilde{a}_j(i)\right\rangle_i s_n(l-k)$, i.e. $\bar{a}_j(l)$ is the part of $\tilde{a}_j(l)$ that is fully correlated with the delayed signal, and $\hat{a}_j(l)$ is completely uncorrelated with the signal. Generally $\left\langle s_n(i-k)\tilde{a}_j(i)\right\rangle_i$ will be very small for each individual delay and principal component, especially if $N_{in}$ is high. Therefore we approximate that $\sigma^2(\hat{a}_j(l)) \approx \sigma^2(\tilde{a}_j(l))$.

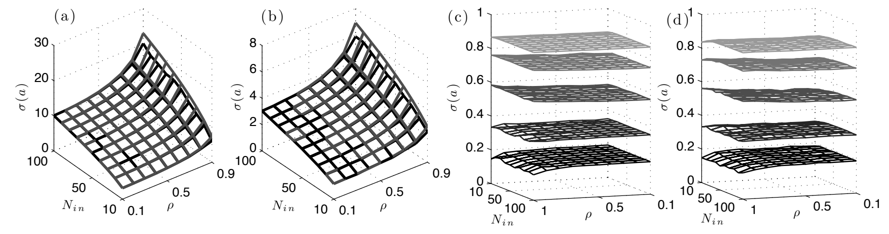

**Figure A.1:** (a,b): Comparison between the mean standard deviation of reservoir states measured empirically (black) and the estimated value (grey) for principal components with the same energy (a) and an energy spectrum as described in the text (b). (c,d): Empirically measured standard deviations in hyperbolic tangent reservoirs for different corrected input scaling factors: from light grey to black, $\phi = \{0.2,\ 0.4,\ 0.8,\ 1.6,\ 3.2\}$, for principal components with the same energy (c) and an energy spectrum as described in the text (d)

Equation A.2 then reduces to

$$\begin{aligned} m_n(k) = \frac{1}{\psi_n} \sum_{j=1}^{N} \xi_j^{-1} ( & \langle s_n(i-k)\bar{a}_j(i) \rangle_i^2 \\ & + \langle s_n(i-k)\hat{a}_j(i) \rangle_i^2 + 2 \langle s_n(i-k)\bar{a}_j(i)\hat{a}_j(i) \rangle_i ). \end{aligned}$$

The third term in this equation will be very small and therefore we shall neglect it (this is especially valid since its expected value is equal to zero). The first term is the actual measurement of the memory function, the second term is always positive and leads to the systematic overestimation, since it will always be positive with a finite sample size. The central limit theorem states that the expected value of the second term will be $\frac{1}{T}\sigma^2(s_n(i-k)\hat{a}_j(i))$, with $T$ the number of samples. When we assume that $s_n(i-k)$ and $\hat{a}_j(i)$ are statistically independent, the variance of their product is equal to the product of their variances. This, and the assumption that $\sigma^2(\hat{a}_j(l)) \approx \sigma^2(\tilde{a}_j(l))$ allows us to work out the second term to be approximately equal to $\frac{N}{T}$.

## A.1.2 Input scaling

To quantify the amount of non-linearity in a reservoir, we will form an estimation of the standard deviation of a linear reservoir. It is possible to exactly calculate the standard deviations of the states for any given linear network, but the resulting formula is highly convoluted and does not give insight into its dependence on general parameters such as spectral radius

and number of input channels. Therefore we shall have to make some broad assumptions to simplify this formula.
In a linear reservoir, the states have an analytical expression as a function of the input signal White et al. (2004):

$$\mathbf{a}(i) = \sum_{k=0}^{\infty} \mathbf{W}^k \mathbf{V}\mathbf{s}(i-1-k). \tag{A.3}$$

Using this, we can calculate the average variance of the reservoir states:

$$\left\langle \mathbf{a}^\mathsf{T}(i)\mathbf{a}(i) \right\rangle_i = \left\langle \sum_{k=0}^{\infty}\sum_{l=0}^{\infty} \mathbf{s}^\mathsf{T}(i-1-k)\mathbf{P}^{kl}\mathbf{s}(i-1-l) \right\rangle_i ,$$

with

$$\mathbf{P}^{kl} = \mathbf{V}^\mathsf{T}(\mathbf{W}^\mathsf{T})^k \mathbf{W}^l \mathbf{V}.$$

Writing this out as an explicit sum this becomes

$$\left\langle \mathbf{a}^\mathsf{T}(i)\mathbf{a}(i) \right\rangle_i = \sum_{k,l=0}^{\infty} \sum_{m,n=1}^{N_{in}} P_{mn}^{kl} \left\langle s_n(i-1-k)s_m(i-1-l) \right\rangle_i .$$

If we assume that $\langle s_n(i-1-k)s_m(i-1-l)\rangle_i = \psi_m \delta_{nm}\delta_{lk}$, i.e. no spatial or temporal correlation exists in the input signal, this expression simplifies to

$$\left\langle \mathbf{a}^\mathsf{T}(i)\mathbf{a}(i) \right\rangle_i = \sum_{m=1}^{N_{in}} \psi_m \sum_{k=0}^{\infty} P_{mm}^{kk}.$$

The term $\sum_{k=0}^{\infty} P_{mm}^{kk}$ can be calculated analytically for individual reservoirs, but it doesn't give a general idea of how it relates to $N_{in}$ and the spectral radius of the reservoir. For this reason we make the following strong simplification: we only consider orthogonal connection matrices, such that $\mathbf{W}\mathbf{W}^\mathsf{T} = \rho^2\mathbf{1}$. With these assumptions we can then calculate that

$$\begin{aligned}\sum_{k=0}^{\infty} P_{mm}^{kk} &= \sum_{k=0}^{\infty} \rho^{2k} \left[\mathbf{V}^\mathsf{T}\mathbf{V}\right]_{mm} \\ &= \frac{1}{1-\rho^2} \left|\mathbf{V}_m\right|^2 ,\end{aligned}$$

where $\mathbf{V}_m$ signifies the $m$-th column of $\mathbf{V}$. To come to a useful final result, we can make further assumptions. If the elements of $\mathbf{V}_m$ are chosen from a normal distribution with unit standard deviation and zero mean, then its square norm has a Chi-square distribution and its mean is equal to the number of elements (i.e. $N$). Using this, we can finally write down the mean variance of the reservoir states:

$$\frac{1}{N}\left\langle \mathbf{a}^{\mathsf{T}}(i)\mathbf{a}(i)\right\rangle_i = \frac{\sum_{m=1}^{N_{in}} \psi_m}{1-\rho^2}. \tag{A.4}$$

We now have a formula that gives an estimation of the variance of the network states with a very clear relation to the signal energy and $\rho$: the state variance is proportional to the total signal energy, and increases when $\rho$ increases. In the special case that all energies $\psi_m$ are equal to one (whitened data), this gives us

$$\frac{1}{N}\left\langle \mathbf{a}^{\mathsf{T}}(i)\mathbf{a}(i)\right\rangle_i = \frac{N_{in}}{1-\rho^2}. \tag{A.5}$$

We will check whether the approximations we made still can be generalized for regular (i.e. non-orthogonal $\mathbf{W}$). Therefore, we simulate linear networks with the conditions described above (i.e the elements of $\mathbf{V}$ have unit standard deviation and zero mean). We study this for two types of input signals: one with $\psi_i = 1$ (using Equation A.5), and one where $\psi_i = 10^{-5\frac{i}{N_{in}}}$ (using Equation A.4), which is comparable to the signal used in Section 3.2.3. We measure the average standard deviation in relation to $\rho$ and $N_{in}$ and compare this with our estimation. We used 10000 input samples and averaged over 10 reservoir initializations with 100 neurons. Figure A.1a and A.1b shows the comparison between our estimation and the measured value of the mean standard deviation of the network activations. Clearly there is a good correspondence between our formula and the actual result. When $\rho$ gets closer to one, it appears we systematically overestimate the standard deviation. This is likely due to the fact that the eigenvalues of orthogonal matrices all have the same absolute value, which corresponds to the same amplification for all its oscillatory modes. Random matrices have their eigenvalues more or less uniformly spread over the area of the disk with radius $\rho$, such that many of its oscillatory modes will decay more quickly.

It is now easy to define an input scaling $\phi$ which signifies the amount of nonlinearity of the network states. We choose the initial input weights from a normal distribution with unit variance and zero mean, and next we multiply this with

$$\phi\sqrt{\frac{1-\rho^2}{\sum_{m=1}^{N_{in}} \psi_m}}, \tag{A.6}$$

where $\phi$ is the desired standard deviation of the reservoir states. One major drawback of the above formula is the fact that it goes to zero when $\rho$ approaches one (and becomes even imaginary when $\rho > 1$). Obviously there is no real issue using a reservoir with spectral radius equal to or slightly higher than one, exactly because the effective spectral radius will always drop under one when the states are pushed far enough in the nonlinear part of the hyperbolic tangent. We will replace $\rho$ with an estimate of the spectral radius of the Jacobian $\mathbf{J}$ of the system, which is defined as

$$J_{ij}(k) = \frac{\partial a_i(k)}{\partial a_j(k-1)} = (1 - a_i^2(k-1))W_{ij}.$$

Disregarding individual differences between the reservoir states, we estimate the average spectral radius of $\mathbf{J}$ as

$$\rho_J = (1 - \frac{1}{N}\left\langle \mathbf{a}(k)^\mathsf{T}\mathbf{a}(k)\right\rangle_k)\rho.$$

We now have to estimate the variance of the states for a nonlinear reservoir, which is very hard indeed. However, since our goal is to give a rough input scaling factor which also works for $\rho \approx 1$, we found an ad hoc solution which works quite well. Since we 'impose' the standard deviation $\phi$ upon the linearized reservoirs, we expect that the standard deviation in the nonlinear reservoir will be roughly equal to the hyperbolic tangent of $\phi$. This gives us the above correction for equation A.6:

$$\phi\sqrt{\frac{1-\rho^2(1-\tanh(\phi)^2)^2}{\sum_{m=1}^{N_{in}}\psi_m}}. \tag{A.7}$$

We tested these assertions by measuring the standard deviation of reservoir states in hyperbolic tangent reservoirs, and see how much the actual mean standard deviation of the network states still depends on $\rho$ and $N_{in}$ after rescaling. Figure A.1 shows the result for mildly to very non-linear reservoirs (i.e. with mean standard deviation close to one). It appears that our result generally holds quite well, and the mean standard deviation only depends little on $\rho$ and $N_{in}$ in the tested ranges.

## A.2 Memory in continuous time

### A.2.1 Transformed matrices

#### A.2.1.1 Constructing connection matrices from a set of eigenvalues

Here we explain the process for building connection matrices with a given eigenvalue distribution. Though this is common knowledge, we include it here for completeness. To build a connection matrix $\mathbf{W}$ for a given eigenvalue distribution we start of with a diagonal matrix $\mathbf{D}$ with the eigenvalues ordered by absolute value on the diagonal. Since our resulting connection matrix has to have real elements, all eigenvalues will be either real, or complex conjugated pairs. In the next step, we perform an orthogonal transformation to make this a real block-diagonal matrix. To do this, we construct a matrix **O**. When $D_{ii}$ is real, $O_{ii} = 1$. When $D_{ii}$ and $D_{i+1,i+1}$ are a complex conjugated pair, we take the elements of the $2 \times 2$ block on the diagonal at the $i$-th and $i+1$-th row and column of **O** as

$$[\mathbf{O}]_{\{i,i+1\}} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{\jmath}{\sqrt{2}} & -\frac{\jmath}{\sqrt{2}} \end{pmatrix}.$$

All other elements of **O** are zero. Finally, we can transform **D** to a real block-diagonal form: $\mathbf{D_b} = \mathbf{O}\mathbf{D}\mathbf{O}^\dagger$. The resulting matrix is a block-diagonal matrix with the same structure as **O**. All real eigenvalues remain in the same place as in **D**, complex pairs of eigenvalues are replaced by a block with elements

$$[\mathbf{D_b}]_{\{i,i+1\}} = \begin{pmatrix} \Re(\lambda_i) & \Im(\lambda_i) \\ -\Im(\lambda_i) & \Re(\lambda_i) \end{pmatrix},$$

Which means one can also directly construct $\mathbf{D_b}$ from the real and imaginary parts of the eigenvalues.
One can already use $\mathbf{D_b}$ as a connection matrix, where it is clear that all single diagonal entries simply act as disconnected low-pass filters, and $2 \times 2$ blocks are associated with two interconnected neurons which together act as a damped resonator. A more general connection topology can be constructed by a similarity transform:

$$\mathbf{W} = \mathbf{C}\mathbf{D_b}\mathbf{C}^{-1},$$

where **C** can be any nonsingular matrix with the eigenvectors of **W** as its columns. For the connection matrices used for empirical testing throughout this paper we will choose **C** with random Gaussian elements.

#### A.2.1.2 $z$-transform of eigenvalue spectrum

To transform a dynamical system in continuous time to discrete time, one has to use the $z$-transform (see for instance Jury (1964)). The transformation between the complex variable $z$ in the $z$-domain and $s$ in the Laplace domain is defined as

$$z = e^{sT_s},$$

where $T_s$ is the sampling period by which the input signal is sampled. Discrete-time reservoirs can in fact be considered as a system where the input signal is sampled from a continuous signal at each time step. As such, we can transform this system to a continuous time equivalent by an inverse $z$-transform. Since reservoir dynamics are predominantly determined by the eigenvalues, we can use the above equation to find the Laplace-domain equivalent eigenvalues. Obviously, the sample period has no well defined meaning in this reasoning. However, when applying the transformation in the two following examples, one can quickly see that it is fact proportional to the reservoir timescale.

### A.2.2 Laplace-eigenvalue distribution for a uniform distribution in the $z$-domain

Starting from a distribution in the $z$-domain, defined in Ozturk et al. (2006) as uniform over a disk with radius $\rho$, we can again use the $z$-transform to determine the distribution of eigenvalues in the Laplace domain. Using coordinates $\sigma$ and $\varpi$ which denote the real and imaginary part of $s$, we can define an infinitesimal patch of area $d\sigma d\varpi$. In the $z$-domain, we use coordinates $\eta$ and $\phi$ for the radius and angle. A patch of area is here defined as $\eta\, d\eta\, d\phi$. The expected number of eigenvalues in this patch is proportional to its surface because of the uniform distribution. Using the relation $\eta = e^{\sigma T_s}$ we find $d\eta = T_s e^{\sigma T_s} d\sigma$. Together with $d\phi = d\varpi T_s$ we can finally write that the expected number of eigenvalues in the patch $d\sigma d\varpi$ is proportional to $T_s^2 e^{2\sigma T_s} d\sigma d\varpi$. Since $\phi$ goes from $-\pi$ to $\pi$, the distribution as a function of $\varpi$ will be uniform between $\varpi \in \{-\pi/T_s \cdots \pi/T_s\}$, and zero outside this range. Equivalently, we find that for $\eta > \rho$, the distribution is zero, so in the Laplace-domain the distribution will be equal to zero for $\sigma < \ln(\eta)/T_s$. We can then finally write for the distribution of eigenvalues in the Laplace domain $D_\lambda$:

$$D_\lambda(\sigma, \varpi) \sim e^{2\sigma T_s} \mathcal{H}\left(\frac{\ln(\eta)}{T_s} - \sigma\right) \operatorname{rect}\left(\frac{T_s \varpi}{2\pi}\right),$$

where $\mathrm{u}(x)$ is the unit step function, and $\operatorname{rect}(x)$ is the rectangle function, equal to one when $x \in \{-1/2 \cdots 1/2\}$, and zero everywhere else. Since we defined $\tau_R$ as the inverse of the mean real part of the eigenvalues, we can do the same here and use the above formula to find that $\tau_R^{-1} = (2T_s)^{-1} + \ln(\rho)$. For simplicity, we assume $\rho = 1$. This finally yields for the distribution $D_\lambda$:

$$D_\lambda(\sigma, \varpi) \sim e^{\sigma \tau_R} \mathcal{H}(-\sigma) \operatorname{rect}\left(\frac{\tau_R \varpi}{4\pi}\right). \tag{A.8}$$

#### A.2.2.1 Distributing eigenvalues

One strategy to generate eigenvalues for an exponentially distributed spectrum, is to use a random number generator with an exponential distribution for the real part, and one with a uniform distribution for the imaginary part. The problem that one faces with this strategy is that the eigenvalues will not necessarily be evenly spread; some places will be crowded, others empty. This gives a very large variance of the MFs; some performing very poor, others very good. Therefore, we shall use a simple algorithm that avoids clustering of eigenvalues. We start from the $z$-domain where we spread eigenvalues more or less evenly, and later transform them to the Laplace domain using the $z$-transform. The method used in Ozturk et al. (2006) to generate even distributions is based on Erdogmus et al. (2003), which uses an iterative method with entropy maximization as its end goal. We used a much simpler approach starting from a geometric point of view. We first define the upper half of the unit disk. The first eigenvalue of the discrete system is chosen randomly from this circle segment. Next, we define a circle of a certain radius $\rho_h$ around this point, and make sure no other eigenvalues can be chosen within it. We repeat the process until we have defined $N/2$ points and then include their complex conjugates. We also make sure no eigenvalue is chosen with an imaginary part smaller than $\rho_h/2$, which avoids clustering with the complex conjugates. $\rho_h$ has to be chosen small enough so that there will be enough space to have $N/2$ eigenvalues within the given area, but large enough to avoid clustering. Since the area cut out by each circle is proportional to $\rho_h^2$, and the total area cut out by the circles is proportional to $N$, $\rho_h$ will have to be proportional to $N^{-1/2}$. Rather than meticulously working out the necessary conditions for $\rho_h$, we eventually settled after some trial and error to choose $\rho_h = (1.7N)^{-1/2}$.

### A.2.3 Resonator reservoirs

Here we base ourselves on a result found in White et al. (2004). In this paper it was found that optimal noise robustness for memory storage is found for reservoirs where the eigenvalues of the connection matrix all lie on a circle on the complex plain centred on the origin, with a radius smaller than 1 (i.e. an orthogonal connection matrix). We assume that discrete-time eigenvalues, denoted by $\lambda'_i$ can be written as

$$\lambda'_i = \rho \exp\left(2\pi\jmath\theta_i\right),$$

where we use the symbol $\jmath$ as the imaginary unit to avoid confusion with indices $i$ or $j$. Transformation of this system to the Laplace domain yields

$$\lambda_i = \frac{\ln(\rho) + 2\pi\jmath\theta_i}{T_s},$$

which means all eigenvalues will lie on a line parallel to the imaginary axis which crosses the real axis at $\ln(\rho)/T_s = -\tau_R^{-1}$. We are free to choose the values $T_s$ and $\rho$, which means we have control over the imaginary as well as the real part of the eigenvalues. We will choose the eigenvalues to lie equidistantly on this line spread between values $\omega N/2$ and $-\omega N/2$. This way, when choosing $i$ as $i = -(N-1)/2 \cdots (N-1)/2$, we can write the eigenvalues as

$$\lambda_i = \jmath\omega i - \frac{1}{\tau_R}. \tag{A.9}$$

### A.2.4 Properties of resonator reservoirs

#### A.2.4.1 General shape of the memory function

We can draw conclusions concerning the shape of the MF of resonator reservoirs when we look at equations 3.13 and 3.18 which state that the MF consists of a set of cross-products of the elements $b_i(\tau)$. The timescale $\tau_R$ defines the exponential window as defined above and we assume that $\tau_R \gg \alpha^{-1}$, i.e. that the reservoir timescale is much longer than the signal fluctuations. This way, for $\tau \gg \alpha^{-1}$ we can neglect the term with $\exp(-\alpha\tau)$ in equation 3.18. The cross-products of the elements of $\mathbf{b}(\tau)$ can then be written as

$$b_i(\tau)b_j^*(\tau) \approx \exp\left(-2\frac{\tau}{\tau_R}\right)\frac{\exp\left(\jmath\omega(i-j)\tau\right)}{(\alpha^2-\lambda_i^2)(\alpha^2-\lambda_j^{*2})},$$

which means that the MF consists of a factor which is periodic with maximum period $T_R$, and a factor which decays exponentially with decay period $\tau_R/2$. In Figure 3.12a is a depiction of the MF for a resonator reservoir which confirms this.

#### A.2.4.2 Approximation of the memory quality

Here we will derive the approximation for the memory quality $M_q(T_R)$ for resonator reservoirs. We will split the calculations up in two main parts. First, we investigate interference from the signal beyond the reservoir period, next we will account for the finite number of Fourier coefficients.

We will redefine the windowed signal $s_W(t,t')$ as being equal to $s(t-t')\exp(-t'/\tau_R)$ for $0 < t' < T_R$ and zero elsewhere. It is useful to state this as its full Fourier series:

$$s_W(t,t') = \sum_{i=-\infty}^{\infty} e^{\jmath\omega i t'} a_i(t),$$

where

$$a_i(t) = \frac{1}{T_R}\int_0^{T_R} e^{\jmath\omega i t'} e^{-\frac{t'}{\tau_R}} s(t-t')dt'.$$

Next we define $\tilde{s}_W(t-t')$ as the truncated Fourier series:

$$\tilde{s}_W(t,t') = \sum_{i=-(N-1)/2}^{(N-1)/2} e^{\jmath\omega i t'} a_i(t).$$

Thirdly, we define the actually reconstructed windowed signal $\widehat{s}_W(t,t')$, which is also a truncated Fourier series, but has coefficients which are defined by equation 3.25, i.e:

$$\widehat{s}_W(t,t') = \sum_{i=-(N-1)/2}^{(N-1)/2} e^{\jmath\omega i t'} a'_i(t),$$

with (adding the scaling factor $T_R^{-1}$)

$$a'_i(t) = \frac{1}{T_R}\int_0^{\infty} e^{\jmath\omega i t'} e^{-\frac{t'}{\tau_R}} s(t-t')dt'. \tag{A.10}$$

Similar to $s_W(t,t')$ we define $\tilde{s}_W(t,t')$ and $\widehat{s}_W(t,t')$ to be zero outside the interval $0 < t' < T_R$. Looking at equation (A.10), we can divide the integration in equal intervals, i.e., we define

$$\int_0^{\infty} f(t)dt = \sum_{j=0}^{\infty}\int_0^{T_R} f(t+jT_R)dt.$$

Since $\frac{2\pi}{\omega} = T_R$, this yields

$$\begin{aligned} a'_i(t) &= \int_0^{\infty} e^{\jmath\omega i t'} e^{-\frac{t'}{\tau_R}} s(t-t')dt' \\ &= \sum_{j=0}^{\infty} e^{-j\frac{T_R}{\tau_R}} \int_0^{T_R} e^{\jmath\omega i t'} e^{-\frac{t'}{\tau_R}} s(t-t'-jT_R)dt' \\ &= \sum_{j=0}^{\infty} a_i(t-jT_R)e^{-j\frac{T_R}{\tau_R}}. \end{aligned}$$

We can then redefine $\widehat{s}_W(t,t')$ as

$$\begin{aligned}\widehat{s}_W(t,t') &= \sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}} e^{\jmath\omega i t'} \sum_{j=0}^{\infty} a_i(t-jT_R) e^{-j\frac{T_R}{\tau_R}} \\ &= \sum_{j=0}^{\infty} e^{-j\frac{T_R}{\tau_R}} \sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}} e^{\jmath\omega i t'} a_i(t-jT_R) \\ &= \sum_{j=0}^{\infty} e^{-j\frac{T_R}{\tau_R}} \tilde{s}_W(t-jT_R,t').\end{aligned}$$

Finally, we can use this expression in the first step to finding the memory quality. Since the MF does not depend on the scaling of the reconstructed signal, we can write (replacing $t'$ with $\tau$)

$$m(\tau)_{[\tau\in\{0\cdots T_R\}]} = \frac{\langle s_W(t,\tau)\widehat{s}_W(t,\tau)\rangle_t^2}{\sigma^2(s_W(t,\tau))\sigma^2(\widehat{s}_W(t,\tau))}.$$

The numerator is given by

$$\begin{aligned}\langle s_W(t,\tau)\widehat{s}_W(t,\tau)\rangle_t^2 &= \sum_{j=0}^{\infty} e^{-j\frac{T_R}{\tau_R}} \langle s_W(t,\tau)\tilde{s}_W(t-jT_R,\tau)\rangle_t^2 \\ &\approx \langle s_W(t,\tau)\tilde{s}_W(t,\tau)\rangle_t^2,\end{aligned}$$

since we can safely assume that the present signal will be virtually uncorrelated with the signal from a multiple of $T_R$ in the past. The denominator can be worked out in a similar manner. Calculating the variance for $\widehat{s}_W(t,t')$, we find

$$\begin{aligned}\sigma^2(\widehat{s}_w(t,\tau)) &= \lim_{P\to\infty}\frac{1}{2P}\int_{-P}^{P} \widehat{s}_W^2(t,\tau)dt \\ &= \lim_{P\to\infty}\frac{1}{2P}\int_{-P}^{P}\left[\sum_{j=0}^{\infty} e^{-j\frac{T_R}{\tau_R}}\tilde{s}_W(t-jT_R,\tau)\right]^2 dt\end{aligned}$$

Again, we neglect correlation between the present signal and the signals which extend multiple times $T_R$ in the past. As such we can rewrite this as

$$\begin{aligned}\sigma^2(\widehat{s}_w(t,\tau)) &\approx \sum_{j=0}^{\infty} e^{-2j\frac{T_R}{\tau_R}} \lim_{P\to\infty}\frac{1}{2P}\int_{-P}^{P}\tilde{s}_W^2(t-jT_R,\tau)dt\\ &= \sigma^2(\tilde{s}_W(t,\tau))\sum_{j=0}^{\infty} e^{-2j\frac{T_R}{\tau_R}}\\ &= \frac{\sigma^2(\tilde{s}_W(t,\tau))}{1-e^{-2\frac{T_R}{\tau_R}}},\end{aligned}$$

which finally leads us to the MF up to $T_R$ in the past:

$$m(\tau)_{[\tau\in\{0\cdots T_R\}]} = \left[1-e^{-2\frac{T_R}{\tau_R}}\right]\frac{\langle s_W(t,\tau)\tilde{s}_W(t,\tau)\rangle^2}{\sigma^2(s_W(t,\tau))\sigma^2(\tilde{s}_W(t,\tau))}.$$

To calculate the memory quality, we will have to make further approximations:

1. We assume that $m(\tau)$ is nearly constant in the range $\tau \in \{0\cdots T_R\}$. This constant is equal to the memory quality $M_q(T_R)$. The assumption can be validated by looking at Figure 3.11a.

2. We assume that $\langle s_W(t,\tau)\tilde{s}_W(t,\tau)\rangle$, $\sigma^2(s_W(t,\tau))$, and $\sigma^2(\tilde{s}_W(t,\tau))$ all evolve with $\tau$ as $\exp(-2\tau/\tau_R)$, multiplied by some constant value. This assumption is exactly true for $\sigma^2(s_W(t,\tau))$, and approximately for the others as long as $\tau_R$ is not too small relative to $T_R$.

Applying this approximation, we only need to find the relative proportions of $\langle s_W(t,\tau)\tilde{s}_W(t,\tau)\rangle$, $\sigma^2(s_W(t,\tau))$, and $\sigma^2(\tilde{s}_W(t,\tau))$ to find the actual memory quality. In order to do this, we integrate these expressions over $\tau$ in the reservoir period. We find

$$\begin{aligned}\int_0^{T_R}\langle s_W(t,\tau)\tilde{s}_W(t,\tau)\rangle_t d\tau &= \int_0^{T_R} d\tau\left\langle\sum_{i=-\infty}^{\infty}\sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}} e^{\jmath\omega(i-j)\tau}a_i(t)a_j^*(t)\right\rangle_t\\ &= \sum_{i=-\infty}^{\infty}\sum_{j=-\frac{N-1}{2}}^{\frac{N-1}{2}}\left\langle a_i(t)a_j^*(t)\right\rangle_t\int_0^{T_R} e^{\jmath\omega(i-j)\tau}\,d\tau\\ &= \sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}}\left\langle |a_i(t)|^2\right\rangle_t.\end{aligned}$$

Similarly, we find

$$\begin{aligned}\int_0^{T_R} \sigma^2(s_W(t,\tau))d\tau &= \sum_{i=-\infty}^{\infty} \left\langle |a_i(t)|^2 \right\rangle \\ \int_0^{T_R} \sigma^2(\tilde{s}_W(t,\tau))d\tau &= \sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}} \left\langle |a_i(t)|^2 \right\rangle .\end{aligned}$$

This finally yields for the memory quality

$$M_q(T_R) \approx [1-\exp(-2T_R/\tau_R)] \frac{\sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}} \left\langle |a_i(t)|^2 \right\rangle}{\sum_{i=-\infty}^{\infty} \left\langle |a_i(t)|^2 \right\rangle}. \tag{A.11}$$

The terms $\left\langle |a_i(t)|^2 \right\rangle$ form the power spectrum of the windowed signal. We can use the Wiener-Khinchin theorem which states that the power spectrum of a signal is equal to the spectrum of its autocorrelation function, which we shall denote as $R_W(t)$. Since this is a discrete spectrum, we have to assume the windowed function is periodic and calculate the autocorrelation function accordingly. Since we take the mean power spectrum over $t$, we shall take the mean over $t$ for the autocorrelation function as well. This will allow us to incorporate the signal statistics $R(t) = \exp(-\alpha|t|)$. We can calculate

$$\begin{aligned}\langle R_W(t')\rangle_t &= \left\langle \int_0^{T_R-t'} s_W(t,\tau)s_W(t,\tau+\theta)d\tau \right\rangle_t \\ &+ \left\langle \int_0^{t'} s_W(t,T_R-t'+\tau)s_W(t,\tau)d\tau \right\rangle_t \\ &= \int_0^{T_R-t'} e^{-\frac{2\tau+t'}{\tau_R}} \underbrace{\langle s(t-\tau)s(t-\tau+t')\rangle_t}_{R(t')} d\tau \\ &+ \int_0^{t'} e^{-\frac{T_R-t'+2\tau}{\tau_R}} \underbrace{\langle s(t-\tau)s(t-\tau+t'-T_R)\rangle_t}_{R(t'-T_R)} d\tau \\ &= \frac{\tau_R}{2}\left(e^{-t'(\alpha+\tau_R^{-1})}(1-e^{(t'-T_R)\tau_R^{-1}})\right) \\ &+ \frac{\tau_R}{2}\left(e^{(t'-T_R)(\alpha+\tau_R^{-1})}(1-e^{-t'\tau_R^{-1}})\right) \\ &\approx \frac{\tau_R}{2}\left(e^{-t'(\alpha+\tau_R^{-1})} + e^{(t'-T_R)(\alpha+\tau_R^{-1})}\right).\end{aligned}$$

The discrete Fourier spectrum of this function can be calculated as

$$\left\langle |a_i(t)|^2 \right\rangle_t = \frac{1}{T_R} \int_0^{T_R} \exp(\jmath \omega i t') \left\langle R_W(t') \right\rangle_t dt',$$

which yields

$$\left\langle |a_i(t)|^2 \right\rangle_t \sim \frac{1}{\frac{T_R^2}{4\pi^2} \left(\alpha + \tau_R^{-1}\right)^2 + i^2}.$$

The sums in equation (A.11) can be approximated by integrals:

$$\sum_{i=-\frac{N-1}{2}}^{\frac{N-1}{2}} \left\langle |a_i(t)|^2 \right\rangle_t \sim \int_{-N/2}^{N/2} \frac{1}{\frac{T_R^2}{4\pi^2} \left(\alpha + \tau_R^{-1}\right)^2 + q^2} dq,$$

and similar for the denominator. This finally yields for the memory quality

$$M_q(T_R) = \frac{2}{\pi} \left[1 - e^{-2\frac{T_R}{\tau_R}}\right] \arctan\left(\frac{\pi N}{T_R(\alpha + \tau_R^{-1})}\right). \tag{A.12}$$

The validity for this approximation is pictured in Figure 3.12b and 3.12c. It appears this gives a good estimate for the memory quality as long as $\tau_R$ is not too small compared to $T_R$.

### A.2.4.3 Asymptotic memory capacity

The limit situation $\tau_R \to \infty$ can now also be worked out. Notice that the assumptions we made in Section 3.3.6 for the normalized $\hat{\lambda}_i$ implies that $\hat{\lambda}_i$ remains finite, and so the imaginary parts to have to be finite. This means that, since $\lambda_i = \hat{\lambda}_i/\tau_R$, the imaginary parts of the eigenvalues will go to zero as well, implying that $T_R$ has to go to infinity together with $\tau_R$ for the derivation to remain valid. When we assume that the MF is flat in intervals $\{iT_R \cdots (i+1)T_R\}$, and using the results from (A.2.4.1), we can write

$$M = T_R \sum_{j=0}^{\infty} M_q(T_R) e^{-2j\frac{T_R}{\tau_R}} = T_R \frac{M_q(T_R)}{1 - e^{-2j\frac{T_R}{\tau_R}}},$$

and together with equation (A.12) this becomes

$$M = \frac{2}{\pi} T_R \arctan\left(\frac{\pi N}{T_R(\alpha + \tau_R^{-1})}\right).$$

When $\tau_R$ and $T_R$ go to infinity we can l'Hopital's rule to find that:

$$\lim_{T_R,\tau_R\to\infty} M = \frac{2N}{\alpha},$$

confirming equation (3.19).

## A.3 Infinite Reservoirs: Recurrent Kernels

### A.3.1 Finite network equivalent for Gaussian RBFs

Here we shall try to find a finite network which - when made infinite - is the equivalent network of a Gaussian RBF kernel. We repeat equation 4.14:

$$k(\mathbf{s}_i,\mathbf{s}_j) = \int_{\Omega_\mathbf{v}} P(\mathbf{v}) f(\mathbf{v},\mathbf{s}_i) f(\mathbf{v},\mathbf{s}_j). \tag{A.13}$$

The first thing we shall attempt is to see what happens if we use Gaussian RBF nodes, where

$$f(\mathbf{v},\mathbf{s}_i) = \exp\left(\frac{-||\mathbf{v}-\mathbf{s}_i||^2}{2\sigma_0^2}\right).$$

The distribution $P(\mathbf{v})$ we shall choose to be a Gaussian distribution with covariance matrix $\sigma^2\mathbf{I}$ and mean at the origin, but for reasons that will become clear later, we choose not to normalize it. The integral then becomes:

$$k(\mathbf{s}_i,\mathbf{s}_j) = \int_{\Omega_\mathbf{v}} \exp\left(\frac{-||\mathbf{v}||^2}{2\sigma^2}\right) \exp\left(\frac{-||\mathbf{v}-\mathbf{s}_i||^2}{2\sigma_0^2}\right) \exp\left(\frac{-||\mathbf{v}-\mathbf{s}_j||^2}{2\sigma_0^2}\right), \tag{A.14}$$

which can be solved easily by integrating over each dimension separately. The solution is:

$$k(\mathbf{s}_i,\mathbf{s}_j) = \frac{\sigma\sigma_0\sqrt{2\pi}}{\sqrt{\sigma_0^2+2\sigma^2}} \exp\left(\frac{-\sigma^2||\mathbf{s}_i-\mathbf{s}_j||^2-\sigma_0^2(||\mathbf{s}_i||^2+||\mathbf{s}_j||^2)}{2\sigma_0^2(\sigma_0^2+2\sigma^2)}\right). \tag{A.15}$$

Can we reduce this equation to obtain the gaussian RBF kernel? If we take the limit for $\sigma\to\infty$, indeed we find that

$$\lim_{\sigma\to\infty} k(\mathbf{s}_i,\mathbf{s}_j) = \sigma_0\sqrt{\pi}\exp\left(\frac{-||\mathbf{s}_i-\mathbf{s}_j||^2}{4\sigma_0^2}\right), \tag{A.16}$$

which is (down to a factor two for the standard deviation) the RBF kernel. If we would have normalized the probability distribution we would still have

to divide this by infinity, which would make the kernel equal to zero.
We now unfortunately have the situation that, in order to make such a network finite, we have to pick our input weights from an infinitely wide distribution, which is of course impossible. As it turns out, no applicable form of Gaussian RBF node networks leads to the RBF kernel.
What if we work with exponential nodes instead of Gaussian RBF nodes, i.e., the activation function is

$$f(\mathbf{v}, \mathbf{s}_i) = \exp\left(\mathbf{v} \cdot \mathbf{s}_i\right).$$

If we again assume a Gaussian distribution for the weights, we can solve equation 4.14. We find

$$k'(\mathbf{s}_i, \mathbf{s}_j) = \exp\left(\frac{\sigma^2}{2}||\mathbf{s}_i + \mathbf{s}_j||^2\right).$$

This kernel, when made recurrent, will rapidly diverge to infinity, and is unsuited for true applications. Suppose that we apply an equivalent to the softmax function, i.e., we divide the activation by its sum. This means that we need to divide the kernel by $B(\mathbf{s}_i)$ and $B(\mathbf{s}_i)$, which would be computed in this case by an integral:

$$B(\mathbf{s}_i) = \int_{\Omega_\mathbf{v}} P(\mathbf{v}) \exp\left(\mathbf{v} \cdot \mathbf{s}_i\right) d\mathbf{v} = \exp\left(\frac{\sigma^2}{2}||\mathbf{s}_i||^2\right),$$

i.e., of the same form of the kernel function. The kernel function associated with the softmax function is finally given by

$$k(\mathbf{s}_i, \mathbf{s}_j) = \frac{k'(\mathbf{s}_i, \mathbf{s}_j)}{B(\mathbf{s}_i)B(\mathbf{s}_j)} = \exp\left(\sigma^2 \mathbf{s}_i \cdot \mathbf{s}_j\right).$$

This kernel is again unstable when made recurrent, so not of immediate interest. What if, instead of taking the sums of the activations, we use the norms? We call the norms $H(\mathbf{s}_i)$ and $H(\mathbf{s}_i)$, and they are given by

$$H(\mathbf{s}_i) = \sqrt{\int_{\Omega_\mathbf{v}} P(\mathbf{v}) \exp(2\mathbf{v} \cdot \mathbf{s}_i) d\mathbf{v}} = \exp\left(\sigma^2 ||\mathbf{s}_i||^2\right).$$

This time, the division leads to

$$k(\mathbf{s}_i, \mathbf{s}_j) = \frac{k'(\mathbf{s}_i, \mathbf{s}_j)}{H(\mathbf{s}_i)H(\mathbf{s}_j)} = \exp\left(-\frac{\sigma^2}{2}||\mathbf{s}_i - \mathbf{s}_j||^2\right).$$

This means that, except for the role of $\sigma$, which has been inverted, we actually do attain a Gaussian kernel function.

## A.3.2 Sparse threshold unit networks

### A.3.2.1 Deriving the equivalent kernel for non-sparse networks

We start from equation 4.14, with $P(\mathbf{v})$ once again a Gaussian distribution with covariance matrix $\sigma^2\mathbf{I}$ and mean at the origin. It is possible to calculate this integral formally, however one can find a fairly simple geometric argument to find a solution to this problem. First realize that we can choose the coordinates of $\mathbf{v}$ such that the two first dimensions lie in the 2D-plane spanned by $\mathbf{s}_i$ and $\mathbf{s}_j$, which reduces the integral to a two dimensional integral over this plane. Next, realize that the function $f(\mathbf{s}\cdot\mathbf{v})$ can be visualized easily as a function were half of the plane equals $\theta_p$ and the other $-\theta_n$. The edge between the two lies perpendicular to $\mathbf{s}$. If we then consider the product under the integral, we see that the plane is divided in four sections. If $\alpha$ is the angle between $\mathbf{s}_i$ and $\mathbf{s}_j$, we get two fractions of $\alpha-\pi$ equal to $\theta_p^2$ and $\theta_n^2$, and two fractions of $\alpha$ equal to $-\theta_n\theta_p$. As the Gaussian is spherically symmetric we need only to consider the normalizing constant, and the integral can be solved as

$$k(\mathbf{s}_i,\mathbf{s}_j) = A\left(1-\frac{\mu}{\pi}\alpha\right), \tag{A.17}$$

where

$$\mu = 1+2\frac{\theta_p\theta_n}{\theta_n^2+\theta_p^2}, \tag{A.18}$$

and

$$A = \frac{\theta_n^2+\theta_p^2}{2}. \tag{A.19}$$

The angle $\alpha$ is given by

$$\alpha = \arccos\left(\frac{\mathbf{s}_i\cdot\mathbf{s}_j}{||\mathbf{s}_i||||\mathbf{s}_j||}\right). \tag{A.20}$$

Applying equation 4.12 we can then write down the recurrent version.

$$\kappa_{t+1,t'+1}(\mathbf{s}_1,\mathbf{s}_2)$$
$$= A\left(1-\frac{\mu}{\pi}\arccos\left(\frac{\mathbf{s}_1(t+1)\cdot\mathbf{s}_2(t'+1)+\kappa_{t,t'}(\mathbf{s}_1,\mathbf{s}_2)}{\sqrt{(||\mathbf{s}_1||^2+\kappa_t(\mathbf{s}_1,\mathbf{s}_1))(||\mathbf{s}_2||^2+\kappa_{t'}(\mathbf{s}_2,\mathbf{s}_2))}}\right)\right).$$

It's easy to see that the term $\kappa_t(\mathbf{s}_1,\mathbf{s}_1) = \kappa_{t'}(\mathbf{s}_2,\mathbf{s}_2) = A$, because if we consider equation A.17, we have to take the angle between the two vectors and if they are equal this angle is always zero. This reduces the recurrent kernel to:

$$\kappa_{t+1,t'+1}(\mathbf{s}_1,\mathbf{s}_2) = A\left(1-\frac{\mu}{\pi}\arccos\left(\frac{\mathbf{s}_1(t+1)\cdot\mathbf{s}_2(t'+1)+\kappa_{t,t'}(\mathbf{s}_1,\mathbf{s}_2)}{\sqrt{(||\mathbf{s}_1||^2+A)(||\mathbf{s}_2||^2+A)}}\right)\right). \tag{A.21}$$

In A.3.3.1 I will show that this kernel has an infinitely high Lyapunov exponent. This means that it will *always* have unstable dynamics.

#### A.3.2.2 Deriving STUN kernels

Here we shall derive the expression for the STUN kernel. As equations in this section can become lengthy we shall abstain from writing out kernel dependencies on the time series.
We start by introducing a neuron index $z$. This index is of an unspecified nature or dimensionality, but we assume it to be continuous and to span all the neurons in the network. Each neuron in the network is then characterized by a set of $K$ indices which index the neurons this neuron receives input from, a weight vector $\mathbf{w}$ of dimensionality $K$, i.e. the recurrent weights for the network nodes, and input weights $\mathbf{v}$. We write the set of indices as $\mathbf{z} = [z_1, \cdots, z_K]$. If we assume an input signal $\mathbf{s}(t)$ of dimension $N_{in}$, we can write the hidden state of a neuron with index $z'$ as

$$a_{\mathbf{v},\mathbf{w},\mathbf{z}}(z', t+1) = f\left(\sum_{i=1}^{K} w_i a_(z_i, t) + \sum_{j=1}^{N_{in}} v_j s_j(t+1)\right). \tag{A.22}$$

Here, $f$ is the threshold function. Notice that we used $z'$ to index this particular neuron. In this sense, the index $z'$ contains all the information of $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{z}$.
If we wish to define a kernel function we shall need to solve the following integral:

$$\kappa_t = \int_{\Omega_\mathbf{z}} d\mathbf{z} \int_{\Omega_{\mathbf{v},\mathbf{w}}} d\mathbf{v} d\mathbf{w} P(\mathbf{v},\mathbf{w}) a^1_{\mathbf{v},\mathbf{w},\mathbf{z}}(z', t) a^2_{\mathbf{v},\mathbf{w},\mathbf{z}}(z', t), \tag{A.23}$$

where $\Omega_\mathbf{w}$ is $\mathbb{R}^K$, $\Omega_\mathbf{v}$ is $\mathbb{R}^{N_{in}}$, and $\int_{\Omega_\mathbf{z}} d\mathbf{z}$ is a shorthand notation for

$$\int_{\Omega_{z_1}} \int_{\Omega_{z_2}} \cdots \int_{\Omega_{z_K}} dz_1 dz_2 \cdots dz_K.$$

I have used the abbreviation $a^1$ and $a^2$ to indicate that these are hidden states resulting from two different input streams $\mathbf{s}_1(t)$ and $\mathbf{s}_2(t)$. The probability distribution $P(\mathbf{v}, \mathbf{w})$ is a a Gaussian distribution with covariance matrix $\mathbf{I}$ and mean at the origin. The integral over $\mathbf{v}$ and $\mathbf{w}$ can be solved in a highly similar fashion as in section A.3.2.1. We simply unite the two vectors as $\mathbf{u} = [\mathbf{v}; \mathbf{w}]$ such that we obtain

$$a^1_{\mathbf{v},\mathbf{w},\mathbf{z}}(z', t+1) a^2_{\mathbf{v},\mathbf{w},\mathbf{z}}(z', t+1) = f\left(\mathbf{u} \cdot [\mathbf{s}_1(t+1); \mathbf{a}^1_{\mathbf{z}}]\right) f\left(\mathbf{u} \cdot [\mathbf{s}_2(t+1); \mathbf{a}^1_{\mathbf{z}}]\right),$$

in which $\mathbf{a}^i_{\mathbf{z}} = [a^i(z_1, t); \cdots ; a^i(z_K, t)]$ and we left out the dependency of $t$ for brevity. This means that we can reuse equation A.17, where the angle $\alpha$ is given by

$$\alpha = \arccos\left(\frac{\mathbf{s}_1(t+1) \cdot \mathbf{s}_2(t+1) + \mathbf{a}^1_{\mathbf{z}} \cdot \mathbf{a}^2_{\mathbf{z}}}{\sqrt{(||\mathbf{s}_1(t+1)||^2 + ||\mathbf{a}^1_{\mathbf{z}}||^2)(||\mathbf{s}_2(t+1)||^2 + ||\mathbf{a}^2_{\mathbf{z}}||^2)}}\right). \tag{A.24}$$

We shall use the shorthand notation $\alpha_{\mathbf{z}}$ for this angle, where we again omit the explicit dependency on time. Equation A.23 reduces to

$$\kappa_{t+1} = \int_{\Omega_{\mathbf{z}}} d\mathbf{z} \underbrace{A\left(1 - \frac{\mu}{\pi}\alpha_{\mathbf{z}}\right)}_{k_{\mathbf{z}}} = \int_{\Omega_{\mathbf{z}}} d\mathbf{z} k_{\mathbf{z}}. \tag{A.25}$$

This integral can be solved when we realize that the variables $a^i(z_j, t)$ are discrete, taking on either $-\theta_n$ or $\theta_p$. Suppose we will start by integrating over $z_1$:

$$\begin{aligned}\int_{\Omega_{z_1}} dz_1 k_{\mathbf{z}} = {} & h_{nn}[k_{\mathbf{z}}]_{a^1(z_1)=a^2(z_1)=-\theta_n} + h_{np}[k_{\mathbf{z}}]_{a^1(z_1)=-\theta_n, a^2(z_1)=\theta_p} \\ & + h_{pn}[k_{\mathbf{z}}]_{a^1(z_1)=\theta_p, a^2(z_1)=-\theta_n} + h_{pp}[k_{\mathbf{z}}]_{a^1(z_1)=a^2(z_1)=\theta_p}.\end{aligned}$$

Here, the numbers $h_{nn}$, $h_{np}$, $h_{pn}$, and $h_{pp}$ correspond to the fractions of the hidden state, i.e. the fraction of the total number of neurons, where the hidden state at time $t$ for time series 1 is the value corresponding to the first index, and the hidden state for time series 2 is the second. We can solve the integration recursively over all $z_i$, since they are all the same.
It is possible to find an expression for the fractions in terms of the previous kernel value $\kappa_t$, which will be the key to defining a recursive formula. Let us first put forward that the sum of all of them equals one. Next, we shall use the fact that when the kernel function acts on two identical time series, the kernel function is always equal to $A$, since $\alpha = 0$ and we assume that

$\int_{\Omega_{z_i}} dz_i = 1$. The kernel values can also be written in terms of the fractions:

$$\begin{aligned}
\kappa_t &= \theta_p^2 h_{pp} + \theta_n^2 h_{nn} - \theta_n \theta_p (h_{np} + h_{pn}) \\
A &= \theta_p^2 (h_{pp} + h_{pn}) + \theta_n^2 (h_{np} + h_{nn}) \\
A &= \theta_p^2 (h_{pp} + h_{np}) + \theta_n^2 (h_{pn} + h_{nn}) \\
1 &= h_{nn} + h_{pp} + h_{np} + h_{pn}.
\end{aligned}$$

Solving this set of equations gives:

$$\begin{aligned}
h_{nn} = h_{pp} &= \frac{\kappa_t + \theta_n \theta_p}{(\theta_n + \theta_p)^2} \\
h_{np} = h_{pn} &= \frac{A - \kappa_t}{(\theta_n + \theta_p)^2}
\end{aligned}$$

We shall now consider two specific cases. One where $\theta_p = \theta_n = 1$, and where $\theta_p = 1, \theta_n = 0$; the sign unit and the binary unit. The first case is the simplest. Consider the terms $||\mathbf{a}_\mathbf{z}^1||^2$ in the denominator. Since all elements in this vector are either $-1$ or 1, this quadratic norm is equal to $M$. For each integration over an index $z_i$, the numerator can then be split into two situations; when the two hidden states are equal, or when they are different. This means that, each time we integrate, we can split all terms the kernel has into two new terms, each with respective fraction $h_{pp}$ and $h_{np}$ where we can fill in one term in the inner product $\mathbf{a}_\mathbf{z}^1 \cdot \mathbf{a}_\mathbf{z}^2$. In the end we will get $2^K$ terms. If we regroup these and count the number of times each of them occurs we will finally reach equation 4.22.
The situation is slightly more complicated for binary nodes. Here, each time we perform an integration we will split each term into three new terms. Still we can regroup them in the end and obtain equation 4.25.
If we do not have a single in-degree, but rather a probability distribution $p_K$ for each in-degree, we simply need to add up the contributions of each in-degree, weighted with their respective probability. After all, we could say that of all neurons, a fraction $p_1$ has one incoming connection, a fraction $p_2$ has two incoming connections, etc. Equation A.23 can be exchanged by a weighted sum over all in-degrees. Furthermore, nothing changes about the line of thought that defines the fractions $h_{nn}$, $h_{np}$, $h_{pn}$, and $h_{pp}$, as these only depend on the threshold values $\theta_n$ and $\theta_p$. Therefore, each term in the weighted sum is simply the corresponding kernel for that particular in-degree, and the recurrent kernel for a STUN with a distribution over in-degrees is the weighted sum of the kernels with fixed in-degrees.

### A.3.3 Lyapunov exponent

The definition of the Lyapunov exponent is given by:

$$\lambda = \lim_{t\to\infty} \lim_{D(0)\to 0} \frac{1}{t} \ln\left(\frac{D(t)}{D(0)}\right),$$

where $D(t)$ is the distance between two state vectors at time $t$, $D(0)$ is the initial infinitesimally small distance between two initial conditions for the system. In the case of infinite networks, this distance is given by

$$D(t) = \sqrt{\kappa_t(\mathbf{s}_1,\mathbf{s}_1) + \kappa_t(\mathbf{s}_2,\mathbf{s}_2) - 2\kappa_t(\mathbf{s}_1,\mathbf{s}_2)}.$$

When $\kappa_t(\mathbf{s}_1,\mathbf{s}_1) = \kappa_t(\mathbf{s}_2,\mathbf{s}_2) = c$ for all $t$, this becomes

$$D(t) = \sqrt{2(c - \kappa_t(\mathbf{s_1},\mathbf{s}_2))}.$$

From now on we shall omit the dependency on the time series $\mathbf{s}$ to lighten notation. First of all we write the recurrent kernel as an iterative function: $\kappa_t = q(\kappa_{t-1})$. As $\kappa_{t+1}$ is infinitesimally close to $c$, we can write $q$ with a first order approximation:

$$\kappa_t(\mathbf{s}_1,\mathbf{s}_2) = q(c) + (\kappa_{t-1} - c)\left[\frac{\partial q(\kappa_{t-1})}{\partial \kappa_{t-1}}\right]_{\kappa_{t-1}=c}.$$

If we then eliminate $\kappa_{t-1}$ by writing it as a first order approximation of $\kappa_{t-2}$, and repeat the process until we reach $\kappa_0$, together with the fact that $q(c) = c$ we can write that:

$$\kappa_t = c + (\kappa_0 - c)\prod_{i=1}^{t}\left(\frac{\partial q(\kappa_{i-1})}{\partial \kappa_{i-1}}\right)_{\kappa_{i-1}=c}. \tag{A.26}$$

We can then find that

$$\frac{D(t)}{D(0)} = \sqrt{\prod_{i=1}^{t}\left(\frac{\partial q(\kappa_{i-1})}{\partial \kappa_{i-1}}\right)_{\kappa_{i-1}=c}}, \tag{A.27}$$

which leads to the following formula for the Lyapunov exponent:

$$\ell = \frac{1}{2}\left\langle \ln\left[\frac{\partial \kappa_i}{\partial \kappa_{i-1}}\right]_{\kappa_{i-1}=c}\right\rangle_i. \tag{A.28}$$

### A.3.3.1 Lyapunov exponent for sparse threshold unit kernels

If we take the derivative of equation A.21, assuming $\mathbf{s}_1 = \mathbf{s}_2$, we find:

$$\frac{\partial \kappa_i}{\partial \kappa_{i-1}} = \frac{A\mu}{\pi} \frac{1}{\sqrt{A - \kappa_{i-1}}}.$$

When the two input signals are equal, $\kappa_{i-1} = A$, and this expression is infinitely large.
STUN kernels do not suffer from this problem. Calculating the derivative of equation 4.22 yields

$$\frac{\partial \kappa_{t+1}}{\partial \kappa_t} = \frac{A}{(\theta_n + \theta_p)^2} \sum_{i=0}^{K} \sum_{j=0}^{K-i} \left[ (i+j) f_{pp}^{i+j-1} f_{np}^{K-i-j} - (K-i-j) f_{pp}^{i+j} f_{np}^{K-i-j-1} \right] Q_{ij},$$

where

$$Q_{ij} = \sum_{k=0}^{K-i-j} \frac{K!}{i!j!k!(K-i-j-k)!} C^0_{ijk(K-i-j-k)}. \tag{A.29}$$

If we then insert the condition that $\kappa_t = A$, we find that the only terms in the summation above are those where the power of $f_{np}$ is equal to zero (as $f_{np} = 0$ when $\kappa_t = A$), we find:

$$\begin{aligned}
\left( \frac{\partial \kappa_{t+1}}{\partial \kappa_t} \right)_{\kappa_t = A} &= \frac{A}{2^{M-1}(\theta_n + \theta_p)^2} \left[ M \sum_{i=0}^{M} Q_{i(M-i)} - \sum_{i=0}^{M-1} Q_{i(M-i-1))} \right] \\
&= \frac{M}{2^{M-1}\zeta} \sum_{i=0}^{M} \binom{M}{i} C^0_{i(M-i)00} \\
&- \frac{M}{2^{M-1}\zeta} \sum_{i=0}^{M-1} \binom{M-1}{i} \left( C^0_{i(M-i-1)01} + C^0_{i(M-i-1)10} \right) \\
&= \frac{2M}{\zeta} \left[ 1 - \frac{1}{2^M} \sum_{i=0}^{M-1} \binom{M-1}{i} \left( C^0_{i(M-i-1)01} + C^0_{i(M-i-1)10} \right) \right] \\
&= \frac{M}{\pi 2^{M-1}} \sum_{i=0}^{M-1} \binom{M-1}{i} \left[ \arccos(\Phi^0_{i(M-i-1)01}) \right].
\end{aligned}$$

Filling in $\theta_p = 1, \theta_n = 0$ then yields the expression given by equation 4.32. The case for $\theta_p = \theta_n = 1$ is again completely analogous where we start from equation 4.25 instead of 4.22.

### A.3.4 Memory capacity

Here we will derive the memory capacity for recurrent kernel machines. We shall only consider the linear recurrent kernel, as given by equation 4.15. We assume a set of $N$ support vectors $s_k(t)$, which are defined for $t \in \{-\infty, \cdots, 0\}$. We now need to find weights $\boldsymbol{\Upsilon}_\tau$ which optimally recreate the input signal from $\tau$ frames in the past. Importantly, when deriving the memory capacities we have to assume that we have an infinite amount of training data, but a limited number of support vectors.
Suppose we have readout weights $v_k^{(\tau)}$ that try to recreate the input signal $s(t-\tau)$. I denote this output signal $\tilde{s}_\tau(t)$, and it is given by

$$\tilde{s}_\tau(t) = \alpha \sum_{k=1}^{N} v_k^{(\tau)} \sum_{i=0}^{\infty} \alpha^i s_k(-i) s(t-i),$$

Or, if we introduce $\mathbf{z}_i$, a column vector with elements $s_k(-i)$ we can rewrite this compactly as

$$\tilde{s}_\tau(t) = \alpha \boldsymbol{\Upsilon}_\tau^\mathsf{T} \sum_{i=0}^{\infty} \alpha^i \mathbf{z}_i s(t-i).$$

We will need to minimize the MSE between $\tilde{s}_\tau(t)$ and $s(t-\tau)$, which is given by

$$\begin{aligned} \text{MSE} &= \left\langle (\tilde{s}_\tau(t) - s(t-\tau))^2 \right\rangle_t . \\ &= \left\langle \tilde{s}_\tau^2(t) \right\rangle_t + \left\langle s(t-\tau)^2 \right\rangle_t - 2 \left\langle s(t-\tau)\tilde{s}_\tau(t) \right\rangle_t . \end{aligned}$$

The first term we can write as

$$\left\langle \tilde{s}_\tau^2(t) \right\rangle_t = \alpha^2 \boldsymbol{\Upsilon}_\tau^\mathsf{T} \sum_{i,j=0}^{\infty} \alpha^{i+j} \mathbf{z}_i \mathbf{z}_j^\mathsf{T} \left\langle s(t-i)s(t-j) \right\rangle_t \boldsymbol{\Upsilon}_\tau.$$

We make the same assumption as in chapter 3, namely that the frames of $s(t)$ are i.i.d with unit standard deviation and zero mean, such that $\left\langle s(t-i)s(t-j) \right\rangle_t = \delta_{ij}$. This leads to

$$\begin{aligned} \left\langle \tilde{s}_\tau^2(t) \right\rangle_t &= \alpha^2 \boldsymbol{\Upsilon}_\tau^\mathsf{T} \underbrace{\sum_{i=0}^{\infty} \alpha^{2i} \mathbf{z}_i \mathbf{z}_j^\mathsf{T}}_{\mathbf{Z}} \boldsymbol{\Upsilon}_\tau \\ &= \alpha^2 \boldsymbol{\Upsilon}_\tau^\mathsf{T} \mathbf{Z} \boldsymbol{\Upsilon}_\tau. \end{aligned}$$

The third term of the MSE we can similarly calculate to be

$$\left\langle s(t-\tau)\tilde{s}_\tau(t) \right\rangle_t = \alpha^{\tau+1} \boldsymbol{\Upsilon}_\tau^\mathsf{T} \mathbf{z}_\tau.$$

In order to find optimal weights we need to derive the MSE to $\boldsymbol{\Upsilon}_\tau$ and equate it to zero. Finally this yields

$$\boldsymbol{\Upsilon}_\tau = \alpha^{\tau-1}\mathbf{Z}^{-1}\mathbf{z}_\tau.$$

Now we can explicitly solve equation 3.1. Notice that we already solved the variance of $\tilde{s}_\tau(t)$ and the covariance between $\tilde{s}_\tau(t)$ and $s(t-\tau)$ as respectively the first and third term in the MSE. When we insert the expression for $\boldsymbol{\Upsilon}_\tau$, we find that

$$\mathrm{cov}(s(t-\tau), \tilde{s}_\tau(t)) = \mathrm{var}(\tilde{s}_\tau(t)) = \alpha^{2\tau}\mathbf{z}_\tau^\mathsf{T}\mathbf{Z}^{-1}\mathbf{z}_\tau,$$

which yields a memory function

$$m(\tau) = \alpha^{2\tau}\mathbf{z}_\tau^\mathsf{T}\mathbf{Z}^{-1}\mathbf{z}_\tau.$$

Finally, we can now solve the memory capacity. We take the sum over all $\tau$

$$\begin{aligned} M &= \sum_{\tau=0}^{\infty} m(\tau) \\ &= \sum_{\tau=0}^{\infty} \alpha^{2\tau}\mathbf{z}_\tau^\mathsf{T}\mathbf{Z}^{-1}\mathbf{z}_\tau \\ &= \mathrm{tr}\left(\mathbf{Z}^{-1}\sum_{\tau=0}^{\infty}\alpha^{2\tau}\mathbf{z}_\tau\mathbf{z}_\tau^\mathsf{T}\right) \\ &= \mathrm{tr}\left(\mathbf{Z}^{-1}\mathbf{Z}\right) = N, \end{aligned}$$

where tr stands for the trace of the matrix.

# Bibliography

Adrian, E. (1928). *The Basis of Sensation: The Action of the Sense Organs.* W.W. Norton & Co, New York.

Aggarwal, C. C., Hinneburg, A., and Keim, D. A. (2001). On the surprising behavior of distance metrics in high dimensional space. In *Lecture Notes in Computer Science*, pages 420–434. Springer.

Antonelo, E. and Schrauwen, B. (2012). Learning slow features with reservoir computing for biologically-inspired robot localization. *Neural Networks*, 25(0):178 – 190.

Ataullakhanov, F., Lobanova, E., Morozova, O., Shnol, E., Ermakova, E., Butylin, A., and Zaikin, A. (2007). Intricate regimes of propagation of an excitation and self-organization in the blood clotting model. *Physics-Uspekhi*, 50:79.

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

Bach, F., Lanckriet, G., and Jordan, M. (2004). Multiple kernel learning, conic duality, and the smo algorithm. In *Proceedings of the 21 International Conference on Machine learning*, pages 41–48.

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fund. Math.*, 3:133–181.

Barabasi, A. L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286:509–512.

Belousov, B. (1959). A periodic reaction and its mechanism. *Compilation of Abstracts on Radiation Medicine*, 147:145.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer.

Boedecker, J., Obst, O., Mayer, N., and Asada, M. (2009). Studies on reservoir initialization and dynamics shaping in echo state networks. In *Proceedings of the European Symposium on Neural Networks*, pages 227–232.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152.

Botton, L., Chappelle, O., DeCoste, D., and Weston, J. (2007). *Large Scale Kernel Machines.* The MIT Press, Cambridge, Masschusetts.

Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168.

Boyd, S. and Chua, L. O. (1985). Fading memory and the problem of approximating nonlinear operators with volterra series. *IEEE Transactions on Circuits and Systems*, 32(11):1150–1171.

Broyden, C. (1970). The convergence of a class of double-rank minimization algorithms. *IMA Journal of Applied Mathematics*, 6(1):76–90.

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

Büsing, L., Schrauwen, B., and Legenstein, R. (2010). Connectivity, dynamics, and memory in reservoir computing with binary and analog neurons. *Neural Computation*, 22(5):1272–1311.

Buteneers, P., Verstraeten, D., Nieuwenhuyse, B. V., Stroobandt, D., Raedt, R., Vonck, K., Boon, P., and Schrauwen, B. (2012). Real-time detection of epileptic seizures in animal models using reservoir computing. in press.

Buteneers, P., Verstraeten, D., van Mierlo, P., Wyckhuys, T., Stroobandt, D., Raedt, R., Hallez, H., and Schrauwen, B. (2011). Automatic detection of epileptic seizures on the intra-cranial electroencephalogram of rats using reservoir computing. *Artificial Intelligence in Medicine*, 53(3):215–223.

Caluwaerts, K. and Schrauwen, B. (2011). The body as a reservoir : locomotion and sensing with linear feedback. In *Proceedings of the 2nd International Conference on Morphological Computation.*

Čerňanskỳ, M., Makula, M., and Beňušková, L. (2009). Improving the state space organization of untrained recurrent networks. *Advances in Neuro-Information Processing*, pages 671–678.

Chapelle, O. (2007). *Large-scale Kernel Machines*, chapter 2: Training a Support Vector Machine in the Primal, pages 29–51. MIT Press.

Chen, L. (1982). Topological structure in visual perception. *Science*, 19(2):371–403.

Cheng, C.-C., Sha, F., and Saul, L. (2009). A fast online algorithm for large margin training of online continuous density hidden markov models. In *Interspeech 2009*, pages 668–671.

Cho, Y. and Saul, L. K. (2010). Large margin classification in infinite neural networks. *Neural Computation*, 22(10):2678–2697.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Crammer, K. (2010). Efficient online learning with individual learning-rates for phoneme sequence recognition. In *the 2010 IEEE International Conference on Acoustics Speech and Signal Processing*, pages 4878–4881.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314.

Dale, K. and Husbands, P. (2010). The evolution of reaction-diffusion controllers for minimally cognitive agents. *Artificial Life*, 16(1):1–19.

Dambre, J., Verstraeten, D., Schrauwen, B., and Massar, S. (2012). Information processing capacity of dynamical systems. *Scientific Reports*. in press.

Dominey, P. F. (1995). Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning. *Biological cybernetics*, 73(3):265–274.

Domingos, P. and Pazzani, M. (1997). On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2):103–130.

Doya, K. (1992). Bifurcations in the learning of recurrent neural networks. In *1992 IEEE International Symposium on Circuits and Systems*, volume 6, pages 2777–2780.

Drachman, D. (2005). Do we have brain to spare? *Neurology*, 64(12):2004–2005.

Drossel, B. (2008). Random boolean networks. *Reviews of nonlinear dynamics and complexity*, 1:69–110.

Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification - Second Edition*. John Wiley and Sons, Inc.

Elman, J. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Erdogmus, D., Hild, K.E., I., and Principe, J. (2003). Online entropy manipulation: stochastic information gradient. *Signal Processing Letters*, 10(8):242– 245.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660.

Espinoza, M., Suykens, J., and Moor, B. (2006). Fixed-size least squares support vector machines: A large scale application in electrical load forecasting. *Computational Management Science*, 3(2):113–129.

Fernando, C. and Sojakka, S. (2003). Pattern recognition in a bucket. In *Proceedings of the 7th European Conference on Artificial Life*, pages 588–597.

FitzHugh, R. (1955). Mathematical models of threshold phenomena in the nerve membrane. *Bulletin of Mathematical Biology*, 17(4):257–278.

Fletcher, R. (1987). *Practical methods of optimization, Volume 1*. Wiley.

Furuya, S. and Itoh, T. (2009). A streamline selection technique for integrated scalar and vector visualization. *The Journal of The Society for Art and Science*, 8:120–129.

Ganguli, S., Huh, D., and Sompolinsky, H. (2008). Memory traces in dynamical systems. *Proceedings of the National Academy of Sciences*, 105(48):18970–18975.

Garofolo, J., of Standards, N. I., (US, T., Consortium, L. D., Science, I., Office, T., States, U., and Agency, D. A. R. P. (1993). *TIMIT Acoustic-phonetic Continuous Speech Corpus*. Linguistic Data Consortium.

Gelbard-Sagiv, H., Mukamel, R., Harel, M., Malach, R., and Fried, I. (2008). Internally generated reactivation of single neurons in human hippocampus during free recall. *Science*, 322(5898):96–101.

Geman, S. (1986). The spectral radius of large random matrices. *The Annals of Probability*, 14(4):1318–1328.

Gerstner, W. (2001). *A Framework for Spiking Neuron Models: The Spike Response Model*, volume 4, chapter 12, pages 469–516. Elsevier Science.

Gerstner, W. and Kistler, W. (2002). *Spiking Neuron Models*. Cambridge University Press.

Goldfarb, D. (1970). A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24:23–26.

Grant, I. (1987). Recursive least squares. *Teaching Statistics*, 9(1):15–18.

Hand, D. and Yu, K. (2001). Idiot's bayes - not so stupid after all? *International Statistical Review*, 69(3):385–398.

Hauser, H., Ijspeert, A., Füchslin, R., Pfeifer, R., and Maass, W. (2012). Towards a theoretical foundation for morphological computation with compliant bodies. *Biological Cybernetics*. in press.

Hawkins, J. and Blakeslee, S. (2004). *On Intelligence*. Times Books.

Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3(31).

Hermans, M. and Schrauwen, B. (2010a). Memory in linear recurrent neural networks in continuous time. *Neural Networks*, 23(3):341–355.

Hermans, M. and Schrauwen, B. (2010b). Memory in reservoirs for high dimensional input. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–7.

Hermans, M. and Schrauwen, B. (2010c). One step backpropagation through time for learning input mapping in reservoir computing applied to speech recognition. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 521–524.

Hermans, M. and Schrauwen, B. (2011). Recurrent kernel machines: Computing with infinite echo state networks. *Neural Computation*, 24(1):104–133.

Hermans, M. and Schrauwen, B. (2012). Infinite sparse threshold unit networks. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 612–619.

Hinton, G., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

Hinton, G. E. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313:504–507.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of ion currents and its applications to conduction and excitation in nerve membranes. *Journal of Physiology*, 117:500–544.

Hosmer, D. and Lemeshow, S. (2000). *Applied logistic regression*, volume 354. Wiley-Interscience.

Hsu, C. and Lin, C. (2002). A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425.

Huang, G., Zhu, Q., and Siew, C. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501.

Izhikevich, E. (2004). Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15:1063–1070.

Jaeger, H. (2001a). The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology.

Jaeger, H. (2001b). Short term memory in echo state networks. Technical Report GMD Report 152, German National Research Center for Information Technology.

Jaeger, H. (2003). Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems*, pages 593–600.

Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunication. *Science*, 308:78–80.

Jaeger, H., Lukosevicius, M., and Popovici, D. (2007). Optimization and applications of echo state networks with leaky integrator neurons. *Neural Networks*, 20:335–352.

Jalalvand, A., Triefenbach, F., and Martens, J. P. (2012). Continuous digit recognition in noise: Reservoirs can do an excellent job! In *Proceedings of the 13th annual conference of the International Speech Communication Association*.

Jalalvand, A., Triefenbach, F., Verstraeten, D., and Martens, J. (2011). Connected digit recognition by means of reservoir computing. In *Proceedings of the 12th annual conference of the International Speech Communication Association*, pages 1725–1728.

Jolliffe, I. T. (2002). *Principal Component Analysis.* Springer.

Jones, B., Stekel, D., Rowe, J., and Fernando, C. (2007). Is there a liquid state machine in the bacterium Escherichia Coli? In *IEEE Symposium on Artificial Life*, pages 187–191.

Jury, E. I. (1964). *Theory and Application of the Z-Transform Method.* Wiley, New York.

Kauffman, S. A. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467.

Keshet, J., McAllester, D., and Hazan, T. (2011). Pac-bayesian approach for minimization of phoneme error rate. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2224–2227.

Kuffler, S. et al. (1953). Discharge patterns and functional organization of mammalian retina. *Journal of Neurophysiology*, 16(1):37–68.

Lanckriet, G., De Bie, T., Cristianini, N., Jordan, M., and Noble, W. (2004). A statistical framework for genomic data fusion. *Bioinformatics*, 20(16):2626–2635.

LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361.

Lee, K. F. and Hon, H. W. (1989). Speaker-independent phone recognition using hidden markov models. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(11):1641 –1648.

Legenstein, R. and Maass, W. (2005). *New Directions in Statistical Signal Processing: From Systems to Brain*, chapter What makes a dynamical system computationally powerful? MIT Press.

Leslie, C., Eskin, E., Cohen, A., Weston, J., and Noble, W. (2004). Mismatch string kernels for discriminative protein classification. *Bioinformatics*, 20(4):467–476.

Lorenz, E. (1963). Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141.

Lukosevicius, M. and Jäger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149.

Lyapunov, A. (1992). The general problem of the stability of motion. *International Journal of Control*, 55(3):531–534.

Lyon, R. (1982). A computational model of filtering, detection and compression in the cochlea. In *Proceedings of the IEEE ICASSP*, pages 1282–1285.

Maass, W., Joshi, P., and Sontag, E. D. (2007). Computational aspects of feedback in neural circuits. *PLOS Computational Biology*, 3(1):e165, 1–20.

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

Mackey, M. and Glass, L. (1977). Oscillation and chaos in physiological control systems. *Science New Series*, 197(4300):287–289.

Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, 7:153–160.

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, pages 735–742.

Masuda, N. and Aihara, K. (2003). Duality of rate coding and temporal coding in multilayered feedforward networks. *Neural Computation*, 15:103–125.

McLachlan, G. and Wiley, J. (1992). *Discriminant analysis and statistical pattern recognition*, volume 5. Wiley Online Library.

Mercer, J. (1909). Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 209:415–446.

Micheli, A., Sperduti, A., and Starita, A. (2007). An introduction to recursive neural networks and kernel methods for cheminformatics. *Current pharmaceutical design*, 13(14):1469–1495.

Mountcastle, V. (1997). The columnar organization of the neocortex. *Brain*, 120(4):701–722.

Mozer, M. C. (1995). A focused backpropagation algorithm for temporal pattern recognition. In Chauvin, Y. and Rumelhart, D. E., editors, *Backpropagation: Theory, Architectures and Applications*, pages 137–169. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10):2061–2070.

Natschläger, T. and Maass, W. (2004). Information dynamics and emergent computation in recurrent circuits of spiking neurons. In *Advances in Neural Information Processing Systems*, volume 16, pages 1255–1262.

Ozturk, M. C., Xu, D., and Principe, J. C. (2006). Analysis and design of echo state networks. *Neural Computation*, 19:111–138.

Paquot, Y., Duport, F., Smerieri, A., Dambre, J., Schrauwen, B., Haelterman, M., and Massar, S. (2012). Optoelectronic reservoir computing. *Scientific Reports*, 2:1–6.

Pastor-Satorras, R. and Vespignani, A. (2001). Epidemic spreading in scale-free networks. *Physical Review Letters*, 86(14):3200–3203.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572.

Pfeifer, R., Bongard, J., and Grand, S. (2007). *How the body shapes the way we think: a new view of intelligence.* The MIT Press.

Pinto, N., Doukhan, D., DiCarlo, J., and Cox, D. (2009). A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS computational biology*, 5(11):e1000579.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for machine learning.* MIT Press.

Rényi, A. (1961). On measures of entropy and information. In *Fourth Berkeley Symposium on Mathematical Statistics and Probability*, pages 547–561.

Robinson, A. J. and Fallside, F. (1987). The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, Cambridge.

Rumelhart, D., Hinton, G., and Williams, R. (1986). *Learning internal representations by error propagation.* MIT Press, Cambridge, MA.

Rutishauser, U., Slotine, J.-J., and Douglas, R. J. (2012). Competition through selective inhibitory synchrony. *Neural Computation*, 24(8):2033–2052.

Schäfer, A. M. and Zimmerman, H. G. (2006). Recurrent neural networks are universal approximators. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 632–640.

Schmidhuber, J., Gers, F., and Eck, D. (2002). Learning nonregular languages: A comparison of simple recurrent neural networks and lstm. *Neural Computation*, 14:2039–2041.

Schrauwen, B., Defour, J., Verstraeten, D., and Van Campenhout, J. (2007). The introduction of time-scales in reservoir computing, applied to isolated digits recognition. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 471–479.

S.Davis and Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech & Signal Processing*, 28:357–366.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis.* Cambridge University Press.

Sivic, J. and Zisserman, A. (2003). Video google: A text retrieval approach to object matching in videos. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, pages 1470–1477.

Skowronski, M. D. and Harris, J. G. (2007). Automatic speech recognition using a predictive echo state network classifier. *Neural Networks*, 20(3):414–423.

Smola, A. and Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems*, 9:155–161.

Sonnenburg, S., Rätsch, G., Schäfer, C., and Schölkopf, B. (2006). Large scale multiple kernel learning. *The Journal of Machine Learning Research*, 7:1531–1565.

Sontag, E. (1998). *Mathematical Control Theory: Deterministic Finite Dimensional Systems.* Springer, New York.

Steil, J. J. (2004). Backpropagation-Decorrelation: Online recurrent learning with O(N) complexity. In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 843–848.

Steil, J. J. (2005). Stability of backpropagation-decorrelation efficient O(N) recurrent learning. In *Proceedings of the European Symposium on Artificial Neural Networks*, pages 43–48.

Sutskever, I., Hinton, G., and Taylor, G. (2008). The recurrent temporal restricted boltzmann machine. In *Advances in Neural Information Processing Systems*, volume 21, pages 1601–1608.

Sutskever, I., Martens, J., and Hinton, G. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning*, pages 1017–1024.

Suykens, J., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J. (2002). *Least Squares Support Vector Machines.* World Scientific Publishing.

Suykens, J. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural Processing Letters*, 9(3):293–300.

Suykens, J. and Vandewalle, J. (2000). Recurrent least squares support vector machines. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 47(7):1109–1114.

Theunissen, F. E. and Miller, J. P. (1995). Temporal encoding in nervous systems: A rigorous definition. *Journal of Computational Neuroscience*, 2:149–162.

Tibshirani, R. (1994). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288.

Tikhonov, A. N. and Arsenin, V. Y. (1977). *Solutions of Ill-Posed Problems.* Winston and Sons.

Triefenbach, F., Jalalvand, A., Schrauwen, B., and Martens, J.-P. (2010). Phoneme recognition with large hierarchical reservoirs. In *Advances in Neural Information Processing Systems 23*, pages 2307–2315.

Van Essen, D. and Maunsell, J. (1983). Hierarchical organization and functional streams in the visual cortex. *Trends in Neurosciences*, 6:370–375.

Vandoorne, K., Dierckx, W., Schrauwen, B., Verstraeten, D., Baets, R., Bienstman, P., and van Campenhout, J. (2008). Toward optical signal processing using photonic reservoir computing. *Optics Express*, 16(15):11182–11192.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory.* Springer-Verlag, New York.

Verstraeten, D. and Schrauwen, B. (2009). On the quantification of dynamics in reservoir computing. In *Proceedings of the International Conference on Artificial Neural Network*, pages 985–994.

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

Verstraeten, D., Schrauwen, B., and Stroobandt, D. (2006). Reservoir-based techniques for speech recognition. In *Proceedings of the World Conference on Computational Intelligence*, pages 1050–1053.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine learning*, pages 1096–1103.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408.

Walley, R. E. and Weiden, T. D. (1973). Lateral inhibition and cognitive masking: A neuropsychological theory of attention. *Psychological Review*, 80(4):284.

Werbos, P. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Applied Mathematics, Harvard University, Boston, MA.

White, O. L., Lee, D. D., and Sompolinsky, H. (2004). Short-term memory in orthogonal neural networks. *Physical Review Letters*, 92:148102.

Wiener, N. (1958). *Nonlinear Problems in Random Theory.* John Wiley.

Williams, C. and Seeger, M. (2001). Using the nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems*, volume 13.

Williams, C. K. A. (1998). Computation with infinite neural networks. *Neural Computation*, 10:1203–1216.

Wiskott, L. and Sejnowski, T. J. (2002). Slow feature analysis: Unsupervised learning of invariances. *Neural Computation*, 14(4):715–770.

wyffels, F. and Schrauwen, B. (2010). A comparative study of reservoir computing strategies for monthly time series prediction. *Neurocomputing*, 73:1958–1964.

wyffels, F., Schrauwen, B., and Stroobandt, D. (2008a). Stable output feedback in reservoir computing using ridge regression. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 808–817.

wyffels, F., Schrauwen, B., Verstraeten, D., and Stroobandt, D. (2008b). Band-pass reservoir computing. In *Proceedings of the International Joint Conference on Neural Networks*, pages 3203–3208, Hong Kong.

Xu, L., Lee, T., and Shen, H. (2010). An information-theoretic framework for flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1216–1224.

Zhabotinsky, A. and Zaikin, A. (1973). Autowave processes in a distributed chemical system. *Journal of theoretical biology*, 40(1):45–61.