Towards Efficient State Representations for Sequential Modelling with State Space Models

Tianyu Li, School of Computer Science

McGill University, Montreal

03, 2023

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

Sequential data refers to information that is ordered into sequences, such as time series, natural language, and DNA sequences [Sammut and Webb, 2010]. This type of data is prevalent in many real-world machine learning applications, such as predicting the temperature of the next hour, classifying different text sources, or navigating a robot.

One approach to modeling sequential data is through the use of State Space Models (SSMs). These models handle sequential data that involve latent variables or parameters that describe the system's evolving state. The concept of *state space* was first introduced in [Kalman, 1960] and refers to the set of all possible configurations of a system.

Efficient state representations are crucial in SSMs, as they impact various aspects of the performance of the models. Sample efficiency is of particular importance in modern machine learning, especially when data is scarce. Model efficiency, in other words, the expressiveness of the model, can also play a role, as a more expressive model allows for more efficient learning and inference. Representation efficiency, or the informativeness of the state representations for the final task, is also important in supervised learning, as incorporating the target signal into the state representation can avoid redundancy and improve learning efficiency. In addition, many machine learning applications must process data streams in real time, making it critical for SSMs to adapt efficiently to the change of dynamics of the data over time.

In this thesis, we will focus on constructing and learning SSMs that consider efficient state representations, taking into account the aforementioned factors.

Abrégé

Les *données séquentielles* font référence à des informations ordonnées en séquences, telles que des séries temporelles, du langage naturel et des séquences d'ADN [Sammut and Webb, 2010]. Ce type de données est courant dans de nombreuses applications d'apprentissage automatique du monde réel, telles que la prédiction de la température de la prochaine heure, la classification de différentes sources de texte ou le guidage d'un robot dans la navigation.

Une approche pour modéliser les données séquentielles est l'utilisation de Modèles d'Espace d'État (MEE). Ces modèles gèrent les données séquentielles qui impliquent des variables ou des paramètres latents qui décrivent l'état évolutif du système. Le concept d'*espace d'état* a été introduit pour la première fois dans [Kalman, 1960] et fait référence à l'ensemble de toutes les configurations possibles d'un système.

Les représentations d'état efficaces sont cruciales dans les MEE, car elles ont un impact sur différents aspects des performances du modèle. L'efficacité de l'échantillonnage est particulièrement importante dans l'apprentissage automatique moderne, surtout lorsque les données sont rares. L'efficacité du modèle, c'est-à-dire l'expressivité du modèle, peut également jouer un rôle, car un modèle plus expressif peut nécessiter moins de paramètres et permettre un apprentissage et une inférence plus efficaces. L'efficacité de la représentation, ou l'informativité de la représentation de l'état pour la tâche finale, est également importante dans l'apprentissage supervisé, car l'incorporation du signal cible dans la représentation de l'état peut éviter la redondance et améliorer l'efficacité de l'apprentissage. De plus, de nombreuses applications d'apprentissage automatique doivent traiter des flux de données

en temps réel, ce qui rend critique que les MEE s'adaptent efficacement à la dynamique des données au fil du temps.

Dans cette thèse, nous nous concentrerons sur la construction et l'apprentissage de MEE qui considèrent des représentations d'état efficaces, en tenant compte des facteurs mentionnés précédemment.

Acknowledgements

I would like to express my deepest gratitude to my Ph.D. supervisors, Guillaume Rabusseau and Doina Precup, for their unwavering support, guidance, and encouragement throughout this journey. Their insightful feedback, constructive criticism, and intellectual enthusiasm have helped shape my research and personal growth. In addition, I would also like to thank my supervisory committee for taking the time to provide me with valuable feedback: Geoff Gordon and Prakash Panangaden.

I am incredibly grateful to my boyfriend, Sebastien Filion, for his love, support, and encouragement, both in my personal life and academic pursuits. I do not think that I can accomplish this without him.

To my friends, Isabelle Mérineau, Min Liang, Ling Zhang, Mingfei Zhao, Clara Lacrocec, Harsh Satija, Beheshteh Tolouei, and Di Wu, thank you for your friendship, laughter, and endless support during the ups and downs of graduate school.

I would also like to thank Bogdan Mazoure for his collaboration and contribution to my research. Moreover, I want to express my appreciation to my lab mates, Jacob Miller, and Maude Lizaire for their invaluable insights, collaboration, and stimulating discussions. I would also like to thank Lucas Caccia, Jad Kabbara, and Emmanuel Bengio for their help with the final chaotic stage of the thesis submission.

Special thanks to the administrative staff, Caroline Lebrun, Ann Jack, and Diti Anastasopoulos for their help and support throughout my Ph.D. journey.

Last but not least, I would like to thank my parents, Daqing Li and You Wang, for their constant encouragement, wisdom, and love throughout my life. Their sacrifices and dedication have always been a source of inspiration and motivation for me.

Contents

List of Figures

List of Tables

List of Abbreviations

SSM . . . . . State Space Model

FSM . . . . . Finite State Machine

FSA . . . . . Finite State Automaton

RNN . . . . . Recurrent Neural Network

DFA . . . . . Deterministic Finite Automaton

NFA . . . . . Nondeterministic Finite Automaton

WFA . . . . . Weighted Finite Automaton

SFA . . . . . Stochastic Finite Automaton

PFA . . . . . Probabilistic Finite Automaton

HMM . . . . . Hidden Markov Model

OOM . . . . . Observable Operator Model

MDP . . . . . Markov Decision Process

IO-OOM . . . Input-output Observable Operator Model

RL . . . . . Reinforcement Learning

POMDP . . . Partially Observable Markov Decision Process

PSR . . . . . Predictive State Representation

TPSR . . . . . Transformed Predictive State Representation

CPSR . . . . . Compressed Predictive State Representation

JL . . . . . Johnson-Lindenstrauss

TT . . . . . Tensor Train (Decomposition)

SVD . . . . . Singular Value Decomposition

2-RNN . . . . Second-order Recurrent Neural Network

LSTM . . . . . Long Short Term Memory (Network)

GRU . . . . . Gated Recurrent Unit (Network)

UQF . . . . . Unnormalized Q Function

PBVI . . . . . Point-Based Value Iteration

CWFA . . . . . Continuous Weighted Finite Automaton

vv-WFA . . . . Vector-valued Weighted Finite Automaton

IHT . . . . . Iterative Hard Thresholding

ALS . . . . . Alternating Least Squares

OLS . . . . . Ordinary Least Squares

TIHT . . . . . Tensor Iterative Hard Thresholding

SGD . . . . . Stochastic Gradient Descent

NL-WFA . . . Nonlinear Weighted Finite Automaton

NADE . . . . Neural Autoregressive Density Estimator

RNADE . . . Real-valued Neural Autoregressive Density Estimator

NCWFA . . . Nonlinear Continuous Weighted Finite Automata

NLP . . . . . Natural Language Processing

EM . . . . . Expectation Maximization

RRNADE . . . Recurrent Real-valued Neural Autoregressive Density Estimator

KDE . . . . . Kernel Density Estimation

Chapter 1

Introduction

Many modern data science and machine learning applications, such as reinforcement learning, language modeling, and time series predictions, involve the process of modeling sequences. State space methods, leveraging models like weighted finite automata, recurrent neural networks, and Markov decision processes, are one of the most classic methods for such tasks. These methods often learn a model that maintains a state representation through time, which stores sufficient knowledge for predicting the future and outputs the final prediction via a function of the representation. Crucially, how to obtain these state representations efficiently and effectively becomes the most important problem in terms of sequential modeling with state space models. In this thesis, we investigate different approaches to represent the state in an efficient way that suits modern machine learning scenarios.

Contribution to original knowledge Efficiency comes in various forms. One of these forms is the representation efficiency w.r.t. the task. For example, in reinforcement learning, many state space models, such as predictive state representations (PSRs), learn the state representations in an unsupervised fashion. Such a learning paradigm disconnects the reward information from the learning of the environment model and can consequently lead to representations that are sample inefficient and time-consuming for the final plan-

ning task. One question one might ask is whether is it possible to connect these previously separated phases and to what extent would this connection bring any benefit.

In Chapter 3, we show that it is possible to make such a connection and propose a novel model that unifies the learning and planning phases of partially observable Markov decision processes and leverage the spectral learning algorithm to recover the model. In doing so, we empirically show on two domains that our approach is more sample and time efficient compared to classical methods as well as deep reinforcement learning methods. Our algorithm is closely related to the spectral learning algorithm for PSRs and offers appealing theoretical guarantees and time complexity. This work was published in Artificial Intelligence and Statistics Conference (AISTATs) 2020 [Li et al., 2020a].

Sample efficiency plays an important role in machine learning tasks. It is crucial especially when learning under small sample size restrictions or noisy datasets. Weighted finite automata (WFAs) can expressively model functions defined over sequences of discrete symbols. WFAs are often learned using the classic spectral learning algorithm [Hsu et al., 2009, Bailly et al., 2010, Balle and Mohri, 2012], which has proved to be sample efficient and enjoys various other properties such as consistency, and minimality. However, WFAs can only work with discrete input variables, while many real-world applications involve modeling sequences of continuous vectors.

In Chapter 4, we extend the classic WFAs models and the spectral learning algorithm to work with a continuous input space (CWFAs). Moreover, we present connections between WFAs, recurrent neural networks, and tensor networks, which are often used in quantum physics and numerical analysis. This work was first published at AISTATs 2019 [Rabusseau et al., 2019]. We then extended this conference version by including the connection with tensor networks and published the journal version of this work in the Machine Learning journal 2022 [Li et al., 2022b].

Another type of efficiency that one cares about is model efficiency, namely a more compact and expressive state representation for state space models. Although spectral learning and WFAs offer appealing learning properties, their expressivity is still limited

as they are inherently linear models, which often results in large, sometimes infinite, state representation sizes. Given the recent successes of nonlinear models in machine learning, it is natural to wonder whether extending WFAs to the nonlinear setting would be beneficial.

In Chapter 5 and Chapter 6, we present nonlinear extensions of WFAs and CWFAs, as well as the corresponding spectral learning algorithms. In these two works, we improve the expressivity of the WFAs and CWFAs and give rise to a more compact and expressive state representation. Chapter 5 was published in AISTATs 2018 [Li et al., 2018] while Chapter 6 was published in the LearnAut workshop in 2022 [Li et al., 2022a] at the International Colloquium on Automata, Languages and Programming (ICALP).

Many contemporary applications have a preference for a form of modeling in which the model updates and makes predictions while receiving new data inputs. This approach is frequently referred to as "online learning from data streams". The effectiveness of such learning tasks, particularly with regard to state space models, depends on the ability of the state representations to efficiently adapt to shifts in the distribution that frequently occur in the sequential modeling of data streams.

In Chapter 7, with a more application-oriented flavor, we present the final contribution of this thesis: the Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE), a flexible density-based model for online classification and density estimation that automatically adapts to distribution shifts. We show that RRNADE is strictly more expressive, in terms of density estimation, than Gaussian HMMs both theoretically and empirically. We also show the advantages of using RRNADE for online classification problems of streaming data through various experiments. This paper is currently in the process of being submitted to Transactions on Machine Learning Research (TMLR)

Contribution of Authors

- Chapters 1 and 2 are written specifically for this thesis.

- Chapter 3 is based on conference paper (AISTATs) [Li et al., 2020a] coauthored by me and Bogdan Mazoure, and supervised by Guillaume Rabusseau and Doina Precup. The idea of unnormalized Q function was developed by me and Bogdan. Bogdan conducted experiments on the S-Pocman environment and wrote the corresponding section of the paper. The rest of the experiments and writing were done by me. Feedback was provided by Guillaume and Doina.

- Chapter 4 is based on a journal paper [Li et al., 2022b] (Machine Learning) supervised by Guillaume Rabusseau and Doina Precup. The concept of the continuous WFA model and the spectral learning algorithm were developed by me and Guillaume. I was in charge of all the experiments while Guillaume developed most of the theories.

- Chapter 5 is based on a conference paper [Li et al., 2018] (AISTATs) supervised by Guillaume Rabusseau and Doina Precup. All experiments and writing were done by me, with feedbacks from Guillaume Rabusseau and Doina Precup.

- Chapter 6 is based on a workshop paper [Li et al., 2022a] (LearnAut in ICALP) coauthored by me and Bogdan Mazoure, and supervised by Guillaume Rabusseau. All experiments and writing were done by me, with feedback from my coauthors.

- Chapter 7 is currently in the process to be submitted to TMLR, coauthored by me and Bogdan Mazoure, under the supervision of Guillaume Rabusseau. I came up with the idea of approximating density functions of sequences using RNADE [Uria et al., 2013] like model while Bogdan proposed the direction of extending it to the online streaming setting. All experiments and writings were done by me, with feedback from Bogdan, and Guillaume.

- Chapters 8 is written specifically for this thesis.

Chapter 2

State Space Models

One classic approach of solving problems that involve modeling system dynamics is state space models (SSMs). The core idea of SSMs is to describe the system as an evolution of a series of unobservable variables or parameters, the so-called *states*. These models are widely applied in areas like time series prediction and classification, natural language processing, reinforcement learning, etc. The basic approach to state space modeling assumes that the development over a certain ordering of a system is determined by an unobserved sequence of vectors $\boldsymbol{h}_1, \cdots, \boldsymbol{h}_n \in \mathbb{R}^k$, that are associated with a sequence of observed vectors $x_1, \cdots, x_n \in \mathscr{X}$, with some output series $y_1, \cdots, y_n \in \mathscr{Y}$, where $\mathscr{X}$ denotes the input space and $\mathscr{Y}$ denotes the output space. The dynamics of the state vectors are governed by a *transition function* $g : \mathbb{R}^k \times \mathscr{X} \to \mathbb{R}^k$. Generally speaking, a state space model takes the following form of computation:

$$\boldsymbol{h}_t = g(\boldsymbol{h}_{t-1}, x_t) + \epsilon_t \tag{2.1}$$

$$y_t = \psi(\boldsymbol{h}_t) + \eta_t \tag{2.2}$$

where ϵ_t and η_t are random noises and $\psi : \mathbb{R}^k \to \mathscr{Y}$ is the output function. A *linear SSM* is an SSM where g and ψ are linear functions and $\mathscr{X}$, $\mathscr{Y}$ are vector spaces. Typically, an

SSM starts with an initial state $\boldsymbol{h}_0$ and computes the hidden states and outputs recursively based on equations 2.1 and 2.2.

There have been many developments in various types of state space models. In classic computer science, SSMs like *Finite State Machines* (FSMs), or *Finite State Automata* (FSAs) are widely used in compiling theory and grammatical inference. In modern machine learning, FSMs are also used for sequential density estimation and inference, as well as extended for problems in reinforcement learning. *Recurrent neural networks* (RNNs) and their variants have been popular choices for natural language processing problems. In this chapter, we will mainly be reviewing these two types of SSMs, i.e. finite state automata and recurrent neural networks and their variants.

2.1 Finite State Automata (FSAs)

An automaton is a simple machine used to recognize patterns with input taken from some character set (or *alphabet*). Given an alphabet Σ, automata are used to model functions over strings, i.e.

$$f : \Sigma^* \to \mathcal{Y},$$

where $\mathcal{Y}$ is the set of outputs. To achieve this, automata maintain a set of *states*, which can be described as instantaneous descriptions of the evolving system. If we assume that a state gives all the relevant information to determine how the system can evolve from that point on, then modeling the dynamics can be reduced to modeling the changes of states, which is defined as *transitions* in automata. Based on different properties of the states set, transitions, and output space, we have various types of automata. If the set of states is finite, we call the corresponding automaton a *finite automaton*. Examples of deterministic finite automata, nondeterministic finite automata and weighted finite automata can be found in Figure 2.1. In this section, we will review a series of finite automata.

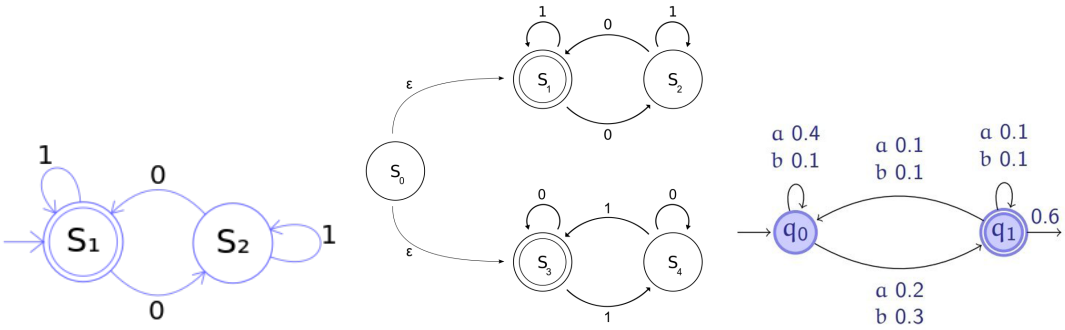


Figure 2.1: Examples of finite state automata. From left to right: DFA[*], NFA[†], and WFA[‡].

2.1.1 Deterministic Finite Automata (DFAs)

A deterministic finite automaton (DFA) is a finite state machine that accepts or rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. It computes a function that maps strings to Boolean values. Formally speaking,

Definition 1. *A DFA of size k is a structure:*

$$M = \langle \mathcal{S}, \Sigma, \delta, s, F \rangle,$$

where:

- $\mathcal{S}$ *is a finite set of* states *and* $|\mathcal{S}| = k$
- $s \in \mathcal{S}$ *is the* initial state
- Σ *is the* alphabet
- $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$ *is the* transition function
- $F \subseteq \mathcal{S}$ *is a set of* accepting states *(*final states*)*

[*]DFA credit: https://en.wikipedia.org/wiki/Deterministic_finite_automaton
[†]NFA credit: https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton
[‡]WFA credit: https://borjaballe.github.io/emnlp14-tutorial/

Additionally, we can extend the transition function δ into its multi-step version (over Σ^*): $\delta^* : \mathcal{S} \times \Sigma^* \to \mathcal{S}$, such that for some state $q \in \mathcal{S}$, the *empty string* λ, some string $x \in \Sigma^*$ and some character $a \in \Sigma$, we have:

$$\delta^*(q, \lambda) = q$$

$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

Definition 2. *A string x is said to be* accepted *by an automata M if $\delta^*(s, x) \in F$ and* rejected *if $\delta^*(s, x) \notin F$.*

Let $\mathbb{B}$ be the set of Boolean values $\mathbb{B} = \{0, 1\}$. Denote the function $f_M : \Sigma^* \to \mathbb{B}$ as the function realized by the automaton M. Given a string $x \in \Sigma^*$, the function over strings *realized* by a DFA is defined as the following:

$$f_M(x) = \begin{cases} 1 & \delta^*(s, x) \in F \\ 0 & \delta^*(s, x) \notin F \end{cases} \tag{2.3}$$

In fact, every finite automaton defines a function over strings. However, the reverse case is not always true. For a function f_M, if it can be realized by some finite automaton M, we call f_M a *rational function*. In the following text, without specifying, we will always use f_M to denote a rational function realized by a finite automaton M.

2.1.2 Nondeterministic Finite Automata (NFAs)

A Nondeterministic Finite Automaton (NFA) is a mathematical model for computation and a crucial idea in the theory of computation and automata. It is a type of finite automaton that allows multiple possible computations for a single input symbol, making its computationally non-deterministic. NFAs offer multiple transition paths for a given input, enabling more flexible computations. This concept is widely utilized in computer science, serving as a foundation for the design and analysis of algorithms, including com-

pilers, lexical analyzers, and pattern recognition systems. As a building block for more intricate computational models, NFA is an important tool for comprehending the theoretical foundations of computer science. Formally speaking, NFAs have the following definition

Definition 3. *An NFA of size k is a five-tuple:*

$$N = \langle \mathcal{S}, \Sigma, \Delta, s, F \rangle$$

where:

- $\mathcal{S}$, Σ *and* F *follow the definitions in definition 1*
- Δ *is the transition function* $\Delta : \mathcal{S} \times \Sigma \to 2^{\mathcal{S}}$, *where* $2^{\mathcal{S}}$ *denotes the* power set *of* $\mathcal{S}$
- s *is the* initial state

The behavior of a NFA is determined by the input symbols and the transition function. For a given input string $x = x_1, \cdots, x_n$, the NFA computes a set of possible computations, each represented as a sequence of states from s to a state in F. The NFA accepts the input string $x_1, \cdots, x_n$ if there exists a computation such that the sequence of states ends in an accepting state. Define $\Delta^* : \mathcal{S} \times \Sigma^* \to 2^{\mathcal{S}}$ and let $\Delta^*(q, x)$ be the set of all states reachable from state q by consuming the string x. Then the function computed by an NFA is:

$$f_N(x) = \begin{cases} 1 & \Delta^*(s, x) \cap F \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where $\Delta^*(q, \lambda) = \{q\}$ and $\Delta^*(q, x\sigma) = \bigcup_{q' \in \Delta^*(q,x)} \Delta(q', \sigma)$, for all $x \in \Sigma^*, \sigma \in \Sigma$.

2.1.3 Weighted Finite Automata (WFAs)

Weighted finite automata (WFAs) are finite state machines that extend traditional finite automata with the ability to assign weights or costs to transitions between states. The

weights can represent a wide range of information, such as the likelihood of a transition occurring, the cost of processing a symbol, or the confidence of a prediction. WFAs can be used to model and solve a variety of computational problems, such as language recognition, string matching, and optimization. Formally speaking, a WFA is defined as the following:

Definition 4. *A WFA of size k is a tuple $A = \langle \Sigma, \boldsymbol{\alpha}, \boldsymbol{\omega}, \{\boldsymbol{A}_\sigma\}_{\sigma\in\Sigma}\rangle$, for the set of real numbers $\mathbb{R}$* [†]*:*

- *Σ is an alphabet*
- *$\boldsymbol{\alpha} \in \mathbb{R}^k$ is the initial states weights*
- *$\boldsymbol{\omega} \in \mathbb{R}^k$ is the final states weights*
- *$\mathbf{A}_\sigma \in \mathbb{R}^k \times \mathbb{R}^k$ is the transition weights for each $\sigma \in \Sigma$*
- *k is the number of states*

A WFA computes a function $f_A : \Sigma^* \to \mathbb{R}$ defined for each word $x = x_1 x_2 \cdots x_n \in \Sigma^*$ by

$$f_A(x) = \boldsymbol{\alpha}^\top \mathbf{A}_{x_1} \mathbf{A}_{x_2} \cdots \mathbf{A}_{x_n} \boldsymbol{\omega}$$

Let $\mathbf{A}_x = \mathbf{A}_{x_1} \mathbf{A}_{x_2} \cdots \mathbf{A}_{x_n}$, we have $f_A(x) = \boldsymbol{\alpha}^\top \mathbf{A}_x \boldsymbol{\omega}$ as its simplified form.

In fact, WFAs are closely related to many models that we are familiar with, such as stochastic finite automata (SFAs), observable operator models (OOMs), hidden Markov models (HMMs), as well as predictive state representations (PSRs). In the following sections, we will introduce these models.

[†]In the literature WFAs are usually defined over a semiring. In the context of machine learning, however, a field algebra structure provides more useful convenience. Normally we set this field to be the real numbers $\mathbb{R}$.

2.1.4 Stochastic Languages and Probabilistic Finite Automata

A *stochastic language* defines a distribution over all possible strings given an alphabet Σ. Formally we have the following definition.

Definition 5. *A function* $f : \Sigma^* \to \mathbb{R}$ *is a stochastic language if* $\sum_{x \in \Sigma^*} f(x) = 1$ *and* $\forall x \in \Sigma^*$, $0 \leq f(x) \leq 1$.

Stochastic language modeling has been applied to many fields. Apart from speech recognition, language models are also essential for optical character recognition [Mori et al., 1992] and language translation [Berger et al., 1994]. Statistical techniques related to those used in language modeling can also be applied to language understanding [Pieraccini et al., 1993].

One can find a subclass of WFA such that the realized functions are stochastic languages. This subclass of WFA is referred to as stochastic WFA (SFA). In other words, given a WFA A and its function f_A, if $\sum_{x \in \Sigma^*} f_A(x) = 1$ and $f_A(x) \geq 0$ for all $x \in \Sigma^*$, then A is an SFA. A stochastic language is *rational* if it can be realized by an SFA.

However, it turns out the learning algorithm we will discuss later does not always return a valid SFA. In fact, even checking if a WFA is stochastic is an undecidable problem [Denis and Esposito, 2008]. Although we can still approximate a stochastic language using the algorithm to construct a WFA, the learned WFA can produce some unexpected values for a stochastic language (negative values or values larger than one). Alternatively, one can consider methods that will produce a WFA that is stochastic by construction, for example, a probabilistic automaton (PFA).

Definition 6. *A PFA of size* k *is a tuple:*

$$M_p = \langle \Sigma, \boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma \in \Sigma}, \boldsymbol{\omega} \rangle$$

where:

- Σ *is an alphabet*

- $\boldsymbol{\alpha} \in \mathbb{R}_+^k$ [‡] *is the initial states probability*
- $\boldsymbol{\omega} \in \mathbb{R}_+^k$ *is the final states probability*
- $\mathbf{A}_\sigma \in \mathbb{R}_+^k \times \mathbb{R}_+^k$ *is the transition probability for each* $\sigma \in \Sigma$
- $\sum_i \boldsymbol{\alpha}_i = 1$
- $\sum_{\sigma \in \Sigma} \sum_{j=1}^k (\mathbf{A}_\sigma)_{i,j} + \boldsymbol{\omega}_i = 1$ *for* $i = 1, 2, \cdots, k$
- $\sum_j^k (\mathbf{A}_\sigma)_{i,j} \boldsymbol{\omega}_j > 0$ *for* $\sigma \in \Sigma$ *and* $i = 1, 2, \cdots, k$

Given a string $x = x_1 x_2 \cdots x_n \in \Sigma^*$, a PFA computes the function of

$$f_{M_p}(x) = \boldsymbol{\alpha}^\top \mathbf{A}_{x_1} \mathbf{A}_{x_2} \cdots \mathbf{A}_{x_n} \boldsymbol{\omega}.$$

One can immediately see that a PFA defines a distribution over Σ^*, thus a stochastic language. We can also find the interpretation of the model's parameters: $\boldsymbol{\alpha}$ can be interpreted as probabilities of starting in each state, $\{\mathbf{A}_\sigma\}_{\sigma \in \Sigma}$ define a collection of transition probabilities between states, while $\boldsymbol{\omega}$ defines the corresponding stopping probabilities for each state.

2.2 Stochastic Processes, HMMs and OOMs

In this subsection, we will discuss hidden Markov models (HMMs), stochastic processes, and observable operator models (OOMs). It turns out that HMMs are equivalent to PFAs with no final states and OOMs are a generalization of HMMs.

2.2.1 Hidden Markov Models (HMMs)

The theory of hidden Markov models (HMMs) were developed in the 60s [Baum and Petrie, 1966], and it soon became popular in many applications, including speech recog-

[‡] $\mathbb{R}_+$ denotes the set of non-negative real numbers

nition [Huang et al., 1990], financial mathematics [Elliott et al., 2002], astronomy [Berger, 1997], biological sequence analysis [Durbin et al., 1998, Li et al., 2013].

Definition 7. *Let h_t denote the state at time step t, and x_t denote the observation at time step t. An HMM with k states can be defined as a tuple:*

$$M_h = \langle \mathcal{S}, \Sigma, \mathbf{T}, \mathbf{O}, \pi \rangle$$

- *$\mathcal{S}$ is a set of possible states of size k.*
- *Σ is an alphabet containing all the possible observations.*
- *$\mathbf{T} \in \mathbb{R}^{k \times k}$ is the state transition probability matrix: $\mathbf{T}_{i,j} = \mathbb{P}(h_{t+1} = j | h_t = i)$.*
- *$\mathbf{O} \in \mathbb{R}^{|\Sigma| \times k}$ is the observation probability matrix: $\mathbf{O}_{i,j} = \mathbb{P}(x_t = i | h_t = j)$.*
- *$\boldsymbol{\mu} \in \mathbb{R}^k$ is the initial state distribution: $\pi_i = \mathbb{P}(h_0 = i)$.*

An HMM defines a probability distribution over sequences of hidden states (h_t) and observations (x_t). In an HMM, we assume that the underlying process follows the Markov property. That is, the current state h_t only depends on the previous state h_{t-1}. Formally speaking, we have $\mathbb{P}(h_t | h_{t-1}, \cdots, h_1) = \mathbb{P}(h_t | h_{t-1})$.

There are three basic problems we need to solve for any HMM:

- *Evaluation problem*: Given the observation sequence $x_1 x_2 \cdots x_n$ and the model M_h, how do we efficiently compute the probability $\mathbb{P}(x_1 x_2 \cdots x_n)$?
- *Inference problem*: Given the observation sequence $x_1 x_2 \cdots x_n$ and the model M_h, how do we choose a corresponding state sequence $h_1 h_2 \cdots h_n$ which best explains the observations?
- *Learning problem*: How do we adjust the model parameters, more precisely $\mathbf{T}, \mathbf{O}, \pi$ to maximize the likelihood of the observations.

There have been many solutions to these three problems. Among them, the classical solutions are the forward algorithm [Juang and Rabiner, 1991] for the evaluation problem, Viterbi algorithm [Viterbi, 1967] for the inference problem and the Baum-Welch algorithm [Baum and Eagon, 1967] for the learning problem.

Intuitively from their definitions, HMMs and PFAs share some similarities. In fact, for any $\sigma \in \Sigma$ let us define $\mathbf{O}_\sigma = diag(\mathbf{O}_{\sigma,1}, \mathbf{O}_{\sigma,2}, \cdots, \mathbf{O}_{\sigma,k})$, then define

$$\mathbf{B}_\sigma = \mathbf{T}\mathbf{O}_\sigma$$

then we have

$$\mathbb{P}(x_1 x_2 \cdots x_n) = \boldsymbol{\mu}^\top \mathbf{B}_{x_1} \mathbf{B}_{x_2} \cdots \mathbf{B}_{x_n} \mathbf{1}_k$$

where $\mathbf{1}_k$ is an all-ones vector of size k. One can observe that this form of an HMM is very similar to a WFA and a PFA. In fact, [Dupont et al., 2005] shows that for any HMM, there exists an equivalent PFA with no final probabilities. Note the terminology "no final probabilities" does not mean $\boldsymbol{\omega} = \mathbf{0}_k$, but means the automaton has no final states, i.e. they will continue evolving forever. To evaluate the probability of a sequence, however, we do need to sum up all the possible sequences of states weighted by their probabilities. Moreover, by leveraging the transformation from a PFA into an HMM, we have the following theorem:

Theorem 1 ([Dupont et al., 2005]). *HMMs are equivalent to probabilistic finite automata with no final probabilities.*

2.2.2 Stochastic Processes and Observable Operator Models

A stochastic process is a mathematical model that describes a time-evolving sequence of random variables. It can be thought of as a collection of random variables defined over a common *index set*, typically the set of natural numbers or real numbers. The value of a stochastic process at any given time point is a random variable and the values at different

time points are assumed to be dependent. The behavior of the process is determined by its probability distribution over the possible sequences of values over time. Stochastic processes are widely used in various fields, including finance, physics, and engineering, to model phenomena that exhibit random behavior over time. In this chapter, our focus will be solely on discrete-time and discrete-valued stochastic processes, meaning that the index set is the set of natural numbers and each random variable has a discrete value. Formally, we have the following definition.

Definition 8. *Given an alphabet Σ, a (discrete-time and discrete-valued) stochastic process is a function $f : \Sigma^* \to [0, 1]$ such that $f(\lambda) = 1$ and $f(x) = \sum_{\sigma\in\Sigma} f(x\sigma)$ for all $x \in \Sigma^*$ and $\sigma \in \Sigma$.*

Modeling an arbitrary stochastic process has been shown difficult for HMMs [Thon and Jaeger, 2015], as the state process might not be Markovian. To tackle this problem, the observable operator models (OOMs) [Jaeger, 1997] as a concise algebraic characterization of stochastic processes. Compared to HMMs, OOMs concentrate on the observations themselves, considering the model trajectory as a sequence of linear operators rather than of states.

Definition 9 ([Thon and Jaeger, 2015])**.** *An observable operator model (OOM) is a linear SSM M such that f_M is a stochastic process. The rank of an OOM is defined to be the size of the state vector of M.*

One can check that any HMM can be converted into an OOM and this conversion can yield an OOM of a smaller size than the HMM [Jaeger, 2000]. Moreover, there are examples of OOMs of finite rank that cannot be modeled by any HMM with a finite number of states [Thon and Jaeger, 2015] such as the "probability clock" problem [Jaeger, 1998]. Therefore, OOMs are strictly more expressive than HMMs and are indeed a generalization of HMMs.

2.3 Markov Decision Processes and Predictive State Representations

In this subsection, we will discuss WFAs constrained on another type of dynamical system: controlled processes. One of the most famous models of such a system is the Markov Decision Process (MDP), which is a popular model in reinforcement learning. MDPs often model fully observable environments, where the agent can directly observe the state. However, in many real-world examples, partial observability is often assumed, where the agent only obtains partial information about the environment. Partially Observable Markov Decision Processes (POMDPs) were developed for this case, as well as a more general model, Predictive State Representations (PSRs).

2.3.1 Controlled Process

Controlled processes are mathematical models that capture the behavior of systems that can be impacted by external control variables. These control variables are utilized to modify the system's behavior with the aim of achieving a specific objective, such as stability, optimization, or regulation. A discrete-valued controlled process S_c with actions from $\mathscr{A}$ and observations from $\mathscr{O}$ is a stochastic process governed by actions.

Definition 10. *A (discrete-valued discrete-time) controlled process with action set $\mathscr{A}$ and observation set $\mathscr{O}$ is a function $f_c : (\mathscr{A} \times \mathscr{O})^* \to [0, 1]$ such that*

- $f_c(\lambda) = 1$
- $\forall x \in (\mathscr{A} \times \mathscr{O})^*, \forall a \in \mathscr{A}$, *we have* $f_c(x) = \sum_{o \in \mathscr{O}} f_c(xao)$

For controlled processes, intuitively, we can consider there are two stochastic processes f^o and f^a, for observations and actions respectively. Then one can view f_c as the probabilities of a sequence of observations given a sequence of actions, that is,

$$\mathbb{P}(o_1 o_2 \cdots o_n | a_1 a_2 \cdots a_n).$$

Following the correlation between OOMs and stochastic processes, the so-called *input-output* OOMs (IO-OOMs) [Jaeger, 1998] are linear SSMs that computes controlled processes. In addition, predictive state representations (PSRs) [Littman and Sutton, 2002] are equivalent to IO-OOMs. In the next subsections, we will introduce MDPs, POMDPs, and PSRs.

2.3.2 Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) [Bellman, 1957b] are the core models for solving reinforcement learning (RL) tasks. Typically, a reinforcement learning task consists of an agent and an environment. At any point in time, the agent is in a given *state* of the environment. The agent then proceeds to interact with the environment via *actions* to obtain or infer information of the environment. The agent obtains a *reward* after accomplishing certain tasks in the environment. An MDP provides a mathematical tool to describe the environment and how the agent interacts with it. Formally speaking:

Definition 11. *An MDP of size k is a tuple $\langle \mathcal{S}, \mathscr{A}, \mathbf{P}, \mathbf{R}, \boldsymbol{\mu} \rangle$ where*

- *$\mathcal{S}$: set of states, $|\mathcal{S}| = k$;*
- *$\mathscr{A}$: set of actions;*
- *$\boldsymbol{\mathcal{P}}^{sas'} = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a) : \mathcal{S} \times \mathscr{A} \times \mathcal{S} \to [0, 1]$ is the transition tensor;*
- *$\mathbf{R}^{sa} = \mathrm{R}(s, a) : \mathcal{S} \times \mathscr{A} \to \mathbb{R}$ is the reward matrix;*
- *$\boldsymbol{\mu} = \mathbb{P}(s_0) : \mathcal{S} \to [0, 1]$ is the initial state distribution.*

The goal of an RL task is often to learn a policy that governs the actions of the agent to maximize the *accumulated discounted rewards* (return) in the future. A stochastic policy in an MDP environment is defined as $\boldsymbol{\Pi} \in [0, 1]^{\mathcal{S} \times \mathscr{A}}$. $\boldsymbol{\Pi}$ operates at the state level. At each timestep, the optimal action is selected probabilistically with respect to $\boldsymbol{\Pi}$ given the state of the current step. The agent then moves to the next state depending on the

corresponding transition matrix indexed by a and collects potential rewards from the state.

2.3.3 Partially Observable Markov Decision Processes (POMDPs)

In MDPs, we assume that the environment is *fully observable*. This means that at any time, the agent can directly obtain the state it is currently in. However, in many real-life applications, such as robot navigation and the game of poker, this is not possible. One generalization of MDPs is to assume partial observability instead, which gives rise to the model of Partially Observable Markov Decision Processes (POMDPs). Under POMDP, the underlying dynamics are still governed by MDPs, but the agent cannot directly observe the state. Instead, the agent obtains observations which are determined by the state (and the action) at the corresponding time step. Formally, we have:

Definition 12. *A POMDP of size k is a tuple $\langle \mathcal{S}, \mathscr{A}, \mathscr{O}, \mathbf{P}, \mathbf{R}, \mathbf{O}, \boldsymbol{\mu} \rangle$ where*

- *$\mathcal{S}$: set of states, $|\mathcal{S}| = k$;*
- *$\mathscr{A}$: set of actions;*
- *$\mathscr{O}$: set of observations;*
- *$\boldsymbol{\mathcal{P}}^{sas'} = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a) : \mathcal{S} \times \mathscr{A} \times \mathcal{S} \to [0, 1]$ is the transition tensor;*
- *$\boldsymbol{\mathcal{O}}^{sao} = \mathbb{P}(O_t = o | S_t = s, A_t = a) : \mathcal{S} \times \mathscr{A} \times \mathscr{O}$ is the emission tensor;*
- *$\mathbf{R}^{sa} = \mathrm{R}(s, a) : \mathcal{S} \times \mathscr{A} \to \mathbb{R}$ is the reward matrix;*
- *$\boldsymbol{\mu} = \mathbb{P}(s_0) : \mathcal{S} \to [0, 1]$ is the initial state distribution.*

As the agent cannot directly observe which state it is at, one classic problem in POMDP is to compute the *belief state* $\boldsymbol{b}(h) \in \mathbb{R}^k$ knowing the past trajectory h. Formally, given $h = a_1 o_1 \cdots a_n o_n \in (\mathscr{A} \times \mathscr{O})^*$, we want to compute $\boldsymbol{b}(h)^\top = [\mathbb{P}(s^1|h), \cdots, \mathbb{P}(s^k|h)]^\top$. This can be solved with a forward method similar to HMM [Juang and Rabiner, 1991].

Let $\tilde{\mathbf{O}}_{ao} = \mathrm{diag}(\boldsymbol{\mathcal{O}}_{s^1,a,o}, \boldsymbol{\mathcal{O}}_{s^2,a,o}, \cdots, \boldsymbol{\mathcal{O}}_{s^k,a,o})$, $\tilde{\mathbf{M}}_a = \mathrm{diag}(\boldsymbol{\Pi}_{s^1,a}, \boldsymbol{\Pi}_{s^2,a}, \cdots, \boldsymbol{\Pi}_{s^k,a})$ and denote $\mathbf{E}_{ao} = \tilde{\mathbf{M}}_a \boldsymbol{\mathcal{P}}_{:,a,:} \tilde{\mathbf{O}}_{ao}$, where $\boldsymbol{\mathcal{P}}_{:,a,:} \in \mathbb{R}^{\mathcal{S}\times\mathcal{S}}$ is a matrix where $(\boldsymbol{\mathcal{P}}_{:,a,:})_{ij} = \boldsymbol{\mathcal{P}}^{iaj}$, $\forall i, j \in \mathcal{S}$. It can be shown that $\boldsymbol{b}(h)^\top = \boldsymbol{\mu}^\top \mathbf{E}_{a_1 o_1} \cdots \mathbf{E}_{a_n o_n}$ and $\boldsymbol{b}(\lambda) = \boldsymbol{\mu}^\top$, where λ denotes the empty string.

Similarly to the MDP setting, the state-level policy for POMDPs is defined by $\boldsymbol{\Pi} \in [0,1]^{\mathcal{S}\times\mathcal{A}}$, where $\boldsymbol{\Pi}_{s,a} = \mathbb{P}(a|s)$. However, due to partial observability, the agent's true state cannot be directly observed. Nonetheless, any state-level policy implicitly induces a probabilistic policy over past trajectories, defined by $\Pi(a|h) = \sum_{s\in\mathcal{S}} \mathbb{P}(s|h)\boldsymbol{\Pi}_{s,a}$ for each $h \in (\mathcal{A}\times\mathcal{O})^*$. Similarly, every state-level policy induces a probabilistic distribution over trajectories. With a slight abuse of notation, denote the probability of a trajectory h under the policy Π by $\mathbb{P}^\Pi(h)$. Here, we assume Π is induced by a state-level policy $\boldsymbol{\Pi}$ and define $\mathbb{P}^\Pi(h) = \boldsymbol{b}(h)^\top \mathbf{1}$, where $\mathbf{1}$ is an all-one vector. To make clear the notations, we will use $\pi : \Sigma^* \to \mathcal{A}$ for deterministic policies in the later chapters.

2.3.4 Predictive State Representations (PSRs)

Predictive State Representations (PSRs) are a representation of POMDPs that uses a prediction of the future state of the system, based on past observations and actions, to make decisions. PSRs are compact representations of POMDPs that can be used to efficiently solve decision-making problems. One important notion for PSRs is the concept of *history* and *test*. This is similar to the notion of prefixes and suffixes in the context of WFAs, which we will introduce shortly. In the following definitions, we will define history and test, as well as the core test set. We will then give the definition of PSRs.

Definition 13. *Given a set of actions $\mathcal{A}$ and a set of observations $\mathcal{O}$, denote the set of sequences $(\mathcal{A}\times\mathcal{O})^*$ by Σ_c^*. Then the set of* history *is defined by $\mathcal{R} = \{s : s = x_1 \cdots x_i,\ \forall x \in \Sigma_c^* \text{ and } i = 1, \cdots, |x|\}$, where $|x|$ denotes the length of the sequence x. Similarly, the set of* test *is defined by $\mathcal{S} = \{s : s = x_i \cdots x_{|x|},\ \forall x \in \Sigma_c^* \text{ and } i = 1, \cdots, |x|\}$. For a history $r \in \mathcal{R}$, an* extended history *of r given some action $a \in \mathcal{A}$ and its corresponding observation $o \in \mathcal{O}$ is rao.*

Definition 14 ([Littman and Sutton, 2002]). *Given a subset of the tests set of size k: $\mathcal{q} \subset \mathcal{S}$, then we have the vector $f_{\mathcal{q}}(\cdot) = [\mathbb{P}(\mathcal{q}_1|\cdot), \mathbb{P}(\mathcal{q}_2|\cdot), \cdots \mathbb{P}(\mathcal{q}_k|\cdot)]^\top$. The set $\mathcal{q}$ is a* core test set *of size k if and only if for any $r \in \mathcal{R}$, it forms a sufficient statistic for r, i.e., there exists a collection of functions $\zeta_q : \mathbb{R}^k \to \mathbb{R}$ for any $q \in \mathcal{S}$ such that:*

$$\mathbb{P}(q|r) = \zeta_q(f_{\mathcal{q}}(r)) \quad \forall r \in \mathcal{R}$$

In this thesis, we will only consider the linear case, that is, the function ζ_q is a linear function. We call $\{\zeta_q | q \in \mathcal{S}\}$ prediction functions.

Definition 15. *A linear* predictive state representation *(PSR) of size k is a tuple $\langle \mathcal{A}, \mathcal{O}, \mathcal{q}, \mathbf{L}, \boldsymbol{l}_1 \rangle$:*

- *$\mathcal{A}$ is a set of actions.*
- *$\mathcal{O}$ is a set of observations.*
- *$\mathcal{q} \subseteq \mathcal{S}$ is a set of core tests of size k.*
- *$\mathbf{L} \in \mathbb{R}^{\mathcal{S} \times \mathcal{q}}$ is a set of linear prediction functions (matrix).*
- *$\boldsymbol{l}_1 = f_{\mathcal{q}}(\lambda)$ is the initial vector, where λ denotes the empty sequence. For $r \in \mathcal{R}$ and $q \in \mathcal{S}$*

It can be checked that given an arbitrary POMDP, one can construct a PSR from the POMDP that produces the same probability distribution over histories as the POMDP model. Moreover, the constructed PSR is no more complex than the POMDP in terms of its size [Littman and Sutton, 2002, Singh et al., 2004]. This means that PSRs are at least as expressive as POMDPs. In addition, the following theorem [Thon and Jaeger, 2015] shows that PSRs are a special case of WFAs.

Theorem 2 ([Thon and Jaeger, 2015]). *Let a linear PSR consisting of k core tests $q_i \in \mathcal{S}$, prediction functions $\mathbf{L}$ and an initial state $\boldsymbol{l}_1$. Then an equivalent WFA $A = \langle \alpha, \{\mathbf{A}_\sigma\}_{\sigma \in \Sigma}, \boldsymbol{\omega} \rangle$ is obtained by setting $\alpha = \boldsymbol{l}_1$, $\boldsymbol{\omega} = \sum_{o \in \mathcal{O}} \mathbf{L}_{ao,:}$ and $\mathbf{A}_\sigma = [\mathbf{L}^\top_{\sigma q_1,:}, \mathbf{L}^\top_{\sigma q_2,:}, \cdots, \mathbf{L}^\top_{\sigma q_k}, :]^\top$*

Learning a PSR can be divided into two stages. First, the discovery problem: choose a sufficient set of test $\mathcal{q}$; second, the learning problem: learn PSR parameters. For the discovery problem, one can simply solve it in a greedy fashion, that is grouping all the linearly independent $q \in \mathcal{S}$ to create the core test set $\mathcal{q}$.

Although this algorithm gives a good solution to the discovery problem of PSRs, when the size of the set $\mathcal{S}$ is too big, it can be time-consuming. Furthermore, in many real-world scenarios, noise in the data can pose additional challenges to this approach. It is easy to see that by using this greedy algorithm, one is essentially selecting the basis for the test space. Therefore, instead of explicitly finding the basis, *transformed predictive state representations* (TPSRs) [Rosencrantz et al., 2004, Boots et al., 2011] offer an alternative solution. TPSRs implicitly estimate a linear transformation of the PSR via subspace-based approaches. This approach drastically reduces the complexity of estimating a PSR model and has shown many benefits, such as consistency and sample efficiency in various RL domains [Boots et al., 2011, Singh et al., 2004].

Indeed, this approach is able to obtain a small transformed space of the original PSRs, however, it still faces scalability issues. Typically, one can obtain an estimate of TPSR by performing truncated SVD on the estimated *system-dynamics matrix* [Singh et al., 2004], which is indexed by histories and tests. The scalability issue arises in complex domains, which require a large number of histories and tests to form the system-dynamics matrix. As the time complexity of SVD is cubic in the number of histories and tests, the computation time explodes in these types of environments.

Compressed predictive state representations (CPSRs) [Hamilton et al., 2013] were introduced to circumvent this issue. The main idea of this approach is to project the high dimensional system-dynamics matrix onto a much smaller subspace spanned by randomly generated bases that satisfy the Johnson-Lindenstrauss (JL) lemma [Johnson and Lindenstrauss, 1984]. The projection matrices corresponding to these bases are referred to as JL matrices. Intuitively, JL matrices define a low-dimensional embedding which approximately preserves Euclidean distance between projected points. More formally, given a

matrix $\mathbf{H} \in \mathbb{R}^{m \times n}$ and JL random projection matrices $\mathbf{\Phi}_1 \in \mathbb{R}^{m \times d_1}$ and $\mathbf{\Phi}_2 \in \mathbb{R}^{n \times d_2}$, the compressed matrix $\mathbf{H}_c$ is computed by:

$$\mathbf{H}_c = \mathbf{\Phi}_1^\top \mathbf{H} \mathbf{\Phi_2}$$

where $\mathbf{H}_c$ is the compressed matrix. The choice of random projection matrix is rather empirical and often depends on the task. Gaussian matrices [Baraniuk and Wakin, 2009] and Rademacher matrices [Achlioptas, 2003] are common choices for the random projection matrices that satisfy JL lemma. Asides from these classic choices, hashed random projections, although do not satisfy the JL lemma, have also been shown to preserve certain kernel functions and perform extremely well in practice [Weinberger et al., 2009, Shi et al., 2009].

2.4 Spectral Learning Algorithm for WFAs

Spectral learning is a powerful and widely-used class of algorithms for learning models from data, particularly in the context of sequential data. This approach has been successful in various fields, including natural language processing, recommendation systems, and state space modeling. In state space modeling, spectral learning algorithms are particularly useful for learning the underlying structure of systems that evolve over time and involve latent variables or parameters, such as HMMs [Hsu et al., 2009] and PSRs [Singh et al., 2004, Boots et al., 2011]. The resulting models can then be used for tasks such as prediction, filtering, and smoothing of the system. The name "spectral" comes from the fact that these types of algorithms often leverage singular value decomposition (a type of spectral decomposition) on a given matrix that describes the system dynamics. One of the natural choices of such a matrix is the so-called *Hankel matrix* for WFAs. In fact, given a WFA A realizing a function f, we can always obtain its corresponding Hankel matrix $\mathbf{H}_f$. However, is the reversed case also true, i.e. given a Hankel matrix $\mathbf{H}_f$ of a function

f, can one always find a WFA realizing f? In this section, we will tackle this problem by introducing the duality result from [Fliess, 1974, Carlyle and Paz, 1971, Hsu et al., 2009, Bailly et al., 2010, Balle Pigem et al., 2013].

2.4.1 Functions over Strings and Hankel Matrices

Previously, we have introduced the notion of functions over strings, i.e. $f : \Sigma^* \to \mathcal{Y}$ to the output space $\mathcal{Y}$. In this section, we will constrain $\mathcal{Y}$ to be the set of real numbers, i.e. $f : \Sigma^* \to \mathbb{R}$. One useful definition for the discussion of the remaining thesis is the rank of the function.

Definition 16. *The* rank *of a function over strings: $f : \Sigma^* \to \mathbb{R}$ is defined as the minimal number of states of a WFA that computes the function f. If f is not rational, then* $\mathrm{rank}(f) = \infty$.

The *Hankel matrix* $\mathbf{H}_f \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$ associated with a function $f : \Sigma^* \to \mathbb{R}$ is the bi-infinite matrix with entries $(\mathbf{H}_f)_{u,v} = f(uv)$ for all words $u, v \in \Sigma^*$, where uv is the concatenation of the prefix u and the suffix v. We can see $\mathbf{H}_f$ as a matrix indexed by prefixes and suffixes. To see this, let us denote the one-hot encoding for the prefix u and suffix v by $\mathbf{u} \in \mathbb{B}^{\Sigma^*}$ and $\mathbf{v} \in \mathbb{B}^{\Sigma^*}$, respectively, then we have $(\mathbf{H}_f)_{u,v} = \mathbf{u}^\top \mathbf{H}_f \mathbf{v}$.

The Hankel matrix is a redundant way to represent the function. Observe that for any u and v, $(\mathbf{H}_f)_{u,v}$ has appeared $|uv| + 1$ times in the matrix. Despite that the Hankel matrix is a redundant way to represent the function, for a Hankel matrix $\mathbf{H}_f$, it fully characterizes its corresponding function f in the sense that for any $u, v \in \Sigma^*$, one can always find the value of $f(uv)$ in $\mathbf{H}_f$.

In practice, it is common to only consider finite sub-blocks of the bi-infinite Hankel matrix. As the Hankel matrix is indexed by prefixes and suffixes, we can therefore extract the rows indexed by a set of prefixes $\mathcal{U}$ and the columns indexed by a set of suffixes $\mathcal{V}$ to form up a sub-block $\mathbf{H}_\mathcal{B} \in \mathbb{R}^{\mathcal{U} \times \mathcal{V}}$, where $\mathcal{B}$ is called a basis $\mathcal{B} = (\mathcal{U}, \mathcal{V})$. By definition, we have $(\mathbf{H}_\mathcal{B})_{u,v} = (\mathbf{H}_f)_{u,v} = f(uv)$, for $u \in \mathcal{U}$ and $v \in \mathcal{V}$. In addition, since $\mathbf{H}_\mathcal{B}$ is a sub-block of $\mathbf{H}_f$, we have $rank(\mathbf{H}_\mathcal{B}) \leq rank(\mathbf{H}_f)$. We are especially interested in the full

rank case where $rank(\mathbf{H}_{\mathscr{B}}) = rank(\mathbf{H}_f)$. We call the basis of such sub-blocks *complete* for the function f, and the sub-block is a *complete sub-block* of $\mathbf{H}_f$.

2.4.2 Duality between minimal WFA and Hankel matrix

In this subsection, we will show that there exists a connection between the rank factorization of $\mathbf{H}_f$ and the minimal WFA A computing f, for a rational function over strings f, and its Hankel matrix $\mathbf{H}_f$. This relation is the core motivation of the spectral learning algorithm that we will be presenting later. To start off, we have the following theorem illustrating the connection between the size of A and the rank of the Hankel matrix $\mathbf{H}_f$.

Theorem 3 ([Carlyle and Paz, 1971, Fliess, 1974]). *A function $f : \Sigma^* \to \mathbb{R}$ can be computed by a WFA iff $rank(\mathbf{H}_f)$ is finite and in that case $rank(\mathbf{H}_f)$ is the minimal number of states of any WFA A such that $f = f_A$*

The above theorem shows a strong relationship between Hankel matrix and WFA. More specifically, for a basis $\mathscr{B}$ if $rank(\mathbf{H}_{\mathscr{B}}) = rank(\mathbf{H}_f)$, then we will be able to find the required number of states for a WFA by looking at the rank of the Hankel matrix. Moreover, one can leverage $\mathbf{H}_{\mathscr{B}}$ to recover the WFA realizing f, which is at the core of the spectral algorithm.

One initial observation is that a WFA A naturally induces a factorization of $\mathbf{H}_f$. Given a WFA $A = \langle \boldsymbol{\alpha}, \{\mathbf{A}\}_{\sigma\in\Sigma}, \boldsymbol{\omega}\rangle$ and the Hankel matrix $\mathbf{H}_f$ corresponding to the function f that A realizes, define $\mathbf{P} \in \mathbb{R}^{\Sigma^*\times k}, \mathbf{S} \in \mathbb{R}^{k\times\Sigma^*}$, where $\mathbf{P}_{u,:} = \boldsymbol{\alpha}^\top \mathbf{A}_u$ and $\mathbf{S}_{:,v} = \mathbf{A}_v\boldsymbol{\omega}$ for $u, v \in \Sigma^*$. One can easily check that $\mathbf{H}_f = \mathbf{PS}$. This factorization also holds for the sub-block of $\mathbf{H}_f$, i.e. given a basis $\mathscr{B} = (\mathscr{U}, \mathscr{V})$, we can construct $\mathbf{P}_{\mathscr{B}}, \mathbf{S}_{\mathscr{B}}$ in the above fashion so that $\mathbf{H}_{\mathscr{B}} = \mathbf{P}_{\mathscr{B}}\mathbf{S}_{\mathscr{B}}$. This rank k factorization $\mathbf{H}_{\mathscr{B}} = \mathbf{P}_{\mathscr{B}}\mathbf{S}_{\mathscr{B}}$ can also be seen as finding a low dimensional representation $(\mathbf{P}_{\mathscr{B}})_{u,:} \in \mathbb{R}^k$ for each prefix u, from which the original *Hankel representation* $(\mathbf{H}_{\mathscr{B}})_{u,:}$ can be recovered using the linear map $\mathbf{S}$ (indeed $(\mathbf{H}_{\mathscr{B}})_{u,:} = (\mathbf{P}_{\mathscr{B}})_{u,:}\mathbf{S}_{\mathscr{B}}$). In the following text, we will use the notation $\mathbf{H}$ as the complete sub-block matrix $\mathbf{H}_{\mathscr{B}}$ to simplify the notation. Besides of the sub-block matrix $\mathbf{H}$, we are

also interested in the series of matrices $\mathbf{H}_\sigma$, where $(\mathbf{H}_\sigma)_{u,v} = f(u\sigma v) = \mathbf{H}_{u\sigma,v}$, and $\sigma \in \Sigma$. By a similar construction, one can verify that $\mathbf{H}_\sigma = \mathbf{P}_\mathscr{B}\mathbf{A}_\sigma\mathbf{S}_\mathscr{B}$.

This observation naturally poses a question: is the converse also true? That is, given a rank factorization of a complete sub-block of $\mathbf{H}_f$, can we find a WFA that computes the function f? The answer is indeed positive and we have the following theorem.

Theorem 4 ([Balle Pigem et al., 2013]). *Let* $\mathbf{H}$ *be a sub-block of* $\mathbf{H}_f$ *and* $\operatorname{rank}(\mathbf{H}) = \operatorname{rank}(\mathbf{H}_f)$, *let* $\mathbf{H} = \mathbf{P}\mathbf{S}$ *be a rank factorization. Then a WFA* $A = \langle \boldsymbol{\alpha}, \{\boldsymbol{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega}\rangle$ *is a minimal WFA computing* f, *where* $\boldsymbol{\alpha}^\top = \mathbf{P}_{\lambda,:}$, $\boldsymbol{\omega} = \mathbf{S}_{:,\lambda}$ *and* $\mathbf{A}_\sigma = \mathbf{P}^\dagger\mathbf{H}_\sigma\mathbf{S}^\dagger$, *and* † *denotes Moore–Penrose pseudoinverse.*

This theorem shows the duality between rank factorization of complete sub-blocks of Hankel matrix and the corresponding minimal WFA of f. In addition, one can check that all minimal WFA for a function f can be transformed between each other via some change of basis [Balle et al., 2014a]. This theorem gives rise to the spectral learning algorithm of WFA which leverages a decomposition of the Hankel matrix.

2.4.3 Spectral Learning Algorithm for WFAs

The spectral learning algorithm, relying on the matrix factorization, can be derived from the proof of Theorem 4. Suppose $f : \Sigma^* \to \mathbb{R}$ is an unknown function of finite rank k and we want to compute a minimal WFA for this function. For a basis $\mathscr{B} = \{\mathscr{U}, \mathscr{V}\}$, assume that we are given an estimate $\hat{\mathbf{H}}$ of the hankel matrix $\mathbf{H} \in \mathbb{R}^{\mathscr{U}'\times\mathscr{V}'}$ and its sub-blocks $(\hat{\mathbf{H}}_\sigma)_{\sigma\in\Sigma}$, where $\mathscr{U}' = \{\lambda\} \cup \mathscr{U}$ and $\mathscr{V}' = \{\lambda\} \cup \mathscr{V}$. Then, we only need one rank factorization of $\hat{\mathbf{H}}$ to be able to apply the algorithm.

Recall that the rank k *compact SVD* of $\hat{\mathbf{H}} \in \mathbb{R}^{\mathscr{U}'\times\mathscr{V}'}$ is given by the expression $\hat{\mathbf{H}} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$, where $\mathbf{U} \in \mathbb{R}^{\mathscr{U}'\times k}$ and $\mathbf{V} \in \mathbb{R}^{\mathscr{V}'\times k}$ are left and right singular vectors, the diagonal matrix $\mathbf{D} \in \mathbb{R}^{k\times k}$ contains all the corresponding singular values.

Now that we have obtained the rank factorization of $\hat{\mathbf{H}}$, we can give the spectral learning algorithm for WFAs. The algorithm is illustrated in Algorithm 1. One side note is that

this spectral learning method is essentially the same as [Bailly et al., 2009] as well as [Hsu et al., 2012], where [Bailly et al., 2009] proposed a principal component analysis based method from a language learning perspective, while [Hsu et al., 2012] proposed the algorithm for HMMs.

The spectral learning algorithm enjoys various nice properties. First and foremost, the solution offered by the spectral learning algorithm is a closed-form one. With only the rank as the hyperparameter, the tuning process of the spectral algorithm is greatly simplified compared to iterative methods. Secondly, if we have the exact complete sub-block of the Hankel matrix $\mathbf{H}_f$ and the function f is rational, then the WFA returned by the spectral algorithm is guaranteed to be the minimal WFA that computes the function [Balle et al., 2014a]. In other words, one cannot find another WFA with a smaller state size that computes the same function. It has also been shown there is a strict convergence to the ground truth model with an increasing amount of samples (consistency) [Balle et al., 2014b]. Last but not least, various PAC bounds are derived, often polynomial w.r.t. the lengths of the sequences [Hsu et al., 2009, Balle and Mohri, 2012], showing the sample efficiency of the method.

Algorithm 1 Spectral Learning Algorithm for WFAs

Input:

A collection of the estimated sub-blocks of the Hankel matrix of the function f: $(\hat{\mathbf{H}}_\sigma)_{\sigma\in\Sigma}$ and $\hat{\mathbf{H}}$.

Output:

A WFA $A = \langle \boldsymbol{\alpha}, \{\mathbf{A}\}_{\sigma\in\Sigma}, \boldsymbol{\omega}\rangle$

1: Perform compact SVD on $\hat{\mathbf{H}}$:

$$\hat{\mathbf{H}} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$$

2: Recover the WFA realizing the function f:

$$\hat{\boldsymbol{\alpha}} = (\mathbf{U}\mathbf{D})_{\lambda,:}$$

$$\hat{\boldsymbol{\omega}} = \mathbf{V}^\top_{:,\lambda}$$

$$\hat{\mathbf{A}}_\sigma = (\mathbf{U}\mathbf{D})^\dagger \hat{\mathbf{H}}_\sigma \mathbf{V}^\top$$

3: **return** $\hat{A} = \langle \hat{\boldsymbol{\alpha}}, \{\hat{\mathbf{A}}\}_{\sigma\in\Sigma}, \hat{\boldsymbol{\omega}}\rangle$

2.4.4 Hankel Matrix Construction

By definition, one can fill the whole Hankel matrix with the corresponding function values. However, in practice, for an arbitrary function $f : \Sigma^* \to \mathbb{R}$, it is hard to obtain a good estimation of a complete sub-block $\mathbf{H}_{\mathscr{B}}$. This comes in two-folds: First, the basis $\mathscr{B}$ is not trivial to find; second, there might be missing values in the sub-block due to the lack of data.

Finding the basis is of great importance as one cannot recover the function through the spectral learning method if the basis is not complete. A common choice is the basis of the form $\mathscr{U} = \mathscr{V} = \Sigma^{\leq l}$ for some $l > 0$ [Hsu et al., 2009], where $\Sigma^{\leq l} = \{w | w \in \Sigma^*, |w| \leq l\}$. One other approach is to choose a basis that contains the most frequent elements observed in the samples [Balle et al., 2014a]. This can be either strings, prefixes, suffixes or substrings. Another way is to use the largest Hankel matrix possible given the input data by building a basis with every prefix and suffix seen in the sample [Bailly et al., 2009]. Note that this approach can be unfeasible due to the amount of prefixes and suffixes being too big.

For a stochastic WFA, we are indeed modeling a stochastic language, i.e. f defines a distribution over strings. Then under this scenario, the Hankel matrix construction and the missing values are easy to deal with. By using the empirical frequencies of the strings appearing in the dataset as estimates, we can approximate the true Hankel matrix. Given a set of strings $\mathscr{W} = \{w_1, w_2, \cdots, w_n\}$, we can estimate the Hankel matrix by:

$$(\hat{\mathbf{H}}_{\mathscr{B}})_{u,v} = \frac{\#(uv \; appears \; in \; \mathscr{W})}{n}$$

In this case, even when there are missing values in the estimated Hankel matrix, since the increasingly large samples yield uniformly convergent estimates for these probabilities, it can be safely assumed that the probability of any string from $\mathscr{B}$ not present in the samples is zero. In fact, it has been shown that this estimate is very close to the true Hankel matrix given enough samples. For $\hat{\mathbf{H}}_{\mathscr{B}}$, the following condition holds with high probability [Hsu

et al., 2009]:

$$\|\mathbf{H}_{\mathscr{B}} - \hat{\mathbf{H}}_{\mathscr{B}}\|_F \leq O(\frac{1}{\sqrt{n}}),$$

where n is the number of i.i.d. strings from some distribution over Σ^*.

For a WFA that computes an arbitrary function, however, it is difficult to apply this treatment. The values assigned by the WFA to an unseen example is unknown. In addition, one cannot expect that a sample set would contain values of the target function for all the strings in $\mathscr{B}$, especially under this setting we cannot simply assign zero to the missing values. To tackle this problem, [Balle and Mohri, 2012] proposed a method based on matrix completion. The core idea is to use a constrained matrix completion method to recover the Hankel matrix that has many missing values. Then decompose the recovered Hankle matrix and proceed with the spectral learning routine.

2.5 Recurrent Neural Networks

While linear state space models, such as weighted finite automata, have shown promise in modeling sequential data, their limitations become apparent as the complexity of the data increases. As a result, researchers have turned to more expressive models, such as *recurrent neural networks* (RNNs), which have the ability to capture complex temporal dependencies in data. These models have gained popularity in many machine learning applications, including natural language processing, speech recognition, and computer vision, due to their ability to effectively model sequential data. In fact, WFAs and RNNs can be seen as two ends of the spectrum in terms of their expressiveness and complexity. On the one hand, WFAs provide a compact and interpretable representation of sequential data that is linear and can be learned efficiently using spectral learning algorithms and comes with theoretical guarantees. On the other hand, RNNs are highly expressive and can capture complex nonlinear dependencies in sequential data, but can be difficult to train and interpret. In addition, due to the strong non-convexity, it is difficult to obtain theoretical results in RNNs learning. Understanding the connection between these

two types of models can lead to new insights into the nature of sequential data and the development of more effective learning algorithms. Later in Chapter 4, we will further showcase this connection.

Formally speaking, RNNs are a class of neural networks designed to handle sequential data. An RNN takes as input a sequence (of arbitrary length) of elements from an input space $\mathcal{X}$ and outputs an element in the output space $\mathcal{Y}$. Thus an RNN computes a function from $\mathcal{X}^*$, the set of all finite-length sequences of elements of $\mathcal{X}$, to $\mathcal{Y}$. In most applications, $\mathcal{X}$ is a vector space, typically $\mathbb{R}^d$. When the input of the problem is sequences of symbols from a finite alphabet Σ, then the so-called *one-hot* encoding is often used to embed Σ into $\mathbb{R}^{|\Sigma|}$ by representing each symbol in Σ by one of the canonical basis vectors.

Definition 17. *Let $\mathcal{X}$ and $\mathcal{Y}$ be the input and output space, respectively. A* recurrent model *with n states is given by a tuple $R = (g, \psi, \boldsymbol{h}_0)$ where $g : \mathcal{X} \times \mathbb{R}^n \to \mathbb{R}^n$ is the* recurrent function, *$\psi : \mathbb{R}^n \to \mathcal{Y}$ is the* output function *and $\boldsymbol{h}_0 \in \mathbb{R}^n$ is the* initial state. *A recurrent model R computes a function $f_R : \mathcal{X}^* \to \mathcal{Y}$ defined by the (recurrent) relation:*

$$f_R(x_1 x_2 \cdots x_k) = \psi(\boldsymbol{h}_k) \quad \textit{where } \boldsymbol{h}_t = g(x_t, \boldsymbol{h}_{t-1}) \textit{ for } 1 \leq t \leq k$$

for all $k \geq 0$ and $x_1, x_2, \ldots, x_k \in \mathcal{X}$.

Many architectures of recurrent neural networks have been proposed and used in practice. The difference of many kinds of recurrent models often resides in the transition functions. In this section, in addition to the first-order (vanilla) RNN, we will cover three variants of the recurrent model, namely the second-order recurrent neural networks (2-RNNs), the long short-term memory networks (LSTMs) as well as the gated recurrent unit networks (GRU), where each of these models has its own different transition function.

Definition 18. *A* first-order RNN *(or vanilla RNN) with n states (or, equivalently, n hidden neurons) is a recurrent model $R = (g, \psi, \boldsymbol{h}_0)$ with input space $\mathcal{X} = \mathbb{R}^d$ and output space $\mathcal{Y} = \mathbb{R}^p$.*

It computes a function $f_R : (\mathbb{R}^d)^* \to \mathbb{R}^p$ *defined by* $f_R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) = \psi(\boldsymbol{h}_k)$*, where the recurrent and output functions are defined by*

$$\boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = z_{rec}(\mathbf{U}\boldsymbol{x}_t + \mathbf{V}\boldsymbol{h}_{t-1}) \quad \textit{and} \quad \boldsymbol{y}_t = \psi(\boldsymbol{h}_t) = z_{out}(\mathbf{W}\boldsymbol{h}_t).$$

The parameters of a first-order RNN are:

- *the initial state* $\boldsymbol{h}_0 \in \mathbb{R}^n$*,*
- *the weight matrices* $\mathbf{U} \in \mathbb{R}^{n \times d}$*,* $\mathbf{V} \in \mathbb{R}^{n \times n}$ *and* $\mathbf{W} \in \mathbb{R}^{p \times n}$*,*
- *the activation functions* $z_{rec} : \mathbb{R}^n \to \mathbb{R}^n$ *and* $z_{out} : \mathbb{R}^p \to \mathbb{R}^p$*.*

For the sake of simplicity, we omitted the bias vectors usually included in the definition of first-order RNN. Note however that this is without loss of generality when z_{rec} is either a rectified linear unit or the identity (which will be the cases considered in this thesis). Indeed, for any recurrent model with n states $R = (g, \psi, \boldsymbol{h}_0)$ with input space $\mathcal{X} = \mathbb{R}^d$ and output space $\mathcal{Y} = \mathbb{R}^p$ defined by

$$\boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = z_{rec}(\mathbf{U}\boldsymbol{x}_t + \mathbf{V}\boldsymbol{h}_{t-1} + \boldsymbol{b}) \quad \text{and} \quad \psi(\boldsymbol{h}_t) = z_{out}(\mathbf{W}\boldsymbol{h}_t + \boldsymbol{c})$$

one can append a 1 to all input vectors, $\tilde{\boldsymbol{x}}_t = (\boldsymbol{x}_t \; 1)^\top$, and define a new recurrent model with $n + 1$ states $\tilde{R} = (\tilde{g}, \tilde{\psi}, \tilde{\boldsymbol{h}}_0)$ with input space $\mathcal{X} = \mathbb{R}^{d+1}$ and output space $\mathcal{Y} = \mathbb{R}^p$ defined by

$$\tilde{\boldsymbol{h}}_t = g(\tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{h}}_{t-1}) = z_{rec}(\tilde{\mathbf{U}}\tilde{\boldsymbol{x}}_t + \tilde{\mathbf{V}}\tilde{\boldsymbol{h}}_{t-1}), \;\; \psi(\tilde{\boldsymbol{h}}_t) = z_{out}(\tilde{\mathbf{W}}\tilde{\boldsymbol{h}}_t) \;\text{ and } \tilde{\boldsymbol{h}}_0 = (\boldsymbol{h}_0 \; 1)^\top$$

computing the same function.

In contrast to first-order RNNs, second-order RNNs (2-RNNs) [Giles et al., 1990, Pollack, 1991, Lee et al., 1986] leverage not only the additive relation between the state and input but also the multiplicative one as well. Second-order recurrent architectures have

been successfully applied more recently, such as in e.g. [Sutskever et al., 2011] and [Wu et al., 2016]. We now give the formal definition of 2-RNNs.

Definition 19. *A* second-order RNN *(2-RNN) with n states is a recurrent model $R = (g, \psi, \boldsymbol{h}_0)$ with input space $\mathcal{X} = (\mathbb{R}^d)^*$ and output space $\mathcal{Y} = \mathbb{R}^p$. It computes a function $f_R : (\mathbb{R}^d)^* \to \mathbb{R}^p$ defined by $f_R(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k) = \psi(\boldsymbol{h}_k)$, where the recurrent and output functions are defined by*

$$\boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = z_{rec}(\boldsymbol{h}_{t-1}^\top (\sum_i (\boldsymbol{x}_t)_i \boldsymbol{\mathcal{A}}_{:,i,:})) \quad \textit{and} \quad \boldsymbol{y}_t = \psi(\boldsymbol{h}_t) = z_{out}(\mathbf{W}\boldsymbol{h}_t).$$

The parameters of a second-order RNN are:

- *the initial state $\boldsymbol{h}_0 \in \mathbb{R}^n$,*
- *the weight tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n \times d \times n}$ and output matrix $\mathbf{W} \in \mathbb{R}^{p \times n}$,*
- *the activation functions $z_{rec} : \mathbb{R}^n \to \mathbb{R}^n$ and $z_{out} : \mathbb{R}^p \to \mathbb{R}^p$.*

A linear 2-RNN R with n states is called minimal *if its number of states is minimal (i.e. any linear 2-RNN computing f_R has at least n states).*

In the remaining of the thesis, we will define a second-order RNN using its parameters, i.e. $R = (\boldsymbol{h}_0, \boldsymbol{\mathcal{A}}, \mathbf{W}, z_{rec}, z_{out})$. In the particular case where the activation functions are linear (i.e. equal to the identity function), we will omit them from the definition, e.g. $R = (\boldsymbol{h}_0, \boldsymbol{\mathcal{A}}, \mathbf{W})$ defines a linear second-order RNN.

The recurrent activation function z_{rec} of a RNN is usually a componentwise non-linear function such as a hyperbolic tangent or rectified linear unit, while the output activation function often depends on the task (the softmax function being the most popular for classification and language modeling tasks).

One can see that the difference between first-order and second-order RNN only lies in the recurrent function. For first-order RNN, the pre-activation $\boldsymbol{a}_t = \mathbf{U}\boldsymbol{x}_t + \mathbf{V}\boldsymbol{h}_{t-1} + \boldsymbol{b}$ is a linear function of $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$, while for second-order RNN the pre-activation $\boldsymbol{a}_t = \boldsymbol{h}_{t-1}^\top (\sum_i (\boldsymbol{x}_t)_i \boldsymbol{\mathcal{A}}_{:,i,:})$ is a bilinear map applied to $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ (hence the *second-order* denomination).

It is worth mentioning that second-order RNN are often defined with additional parameters to account for first-order interactions and bias terms:

$$\boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = z_{rec}(\boldsymbol{h}_{t-1}^\top(\sum_i (\boldsymbol{x}_t)_i \boldsymbol{\mathcal{A}}_{:,i,:}) + \mathbf{U}\boldsymbol{x}_t + \mathbf{V}\boldsymbol{h}_{t-1} + \boldsymbol{b}).$$

The definition we use here is conceptually simpler and without loss of generality (similarly to the omission of the bias vectors in the definition of first-order RNN). Indeed, when z_{rec} is either the identity or a rectified linear unit, one can always append a 1 to all input vectors and augment the state space by one state to obtain a 2-RNN computing the same function. It follows from this discussion that 2-RNN are a strict generalization of vanilla RNN: any function that can be computed by a vanilla RNN can be computed by a 2-RNN (provided that all input vectors are appended a constant entry equal to one).

One of the most benefits of RNNs structure is to allow the use of contextual information when mapping between input and output sequences. However, for traditional RNNs, due to vanishing (exploding) gradient problems that often occur during training, the range of context that can be accessed by the model is often limited. Typically, the weights of the network receive an update proportional to the partial derivative of the error function with respect to the current weight. In some cases, the gradient update will be vanishingly small, effectively preventing the weight from changing its value, or explosively large, leading to out-of-bounds prediction (NaNs). Through introducing the concept of *gating mechanism*, Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] is able to alleviate this issue and show great success in numerous sequential prediction and classification tasks. In essence, the gating mechanism selectively passes or retains information during training through the use of the Hadamard product.

Definition 20. *A* long short-term memory networks *(LSTM) with n hidden states is a recurrent model $R = (g, \psi, \boldsymbol{h}_0)$ with input space $\mathcal{X} = (\mathbb{R}^d)^*$ and output space $\mathcal{Y} = \mathbb{R}^p$. It computes a function $f_R : (\mathbb{R}^d)^* \to \mathbb{R}^p$ defined by $f_R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) = \psi(\boldsymbol{h}_k)$, where the transition function*

$g : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$ *is defined by:*

$$\text{Forget Gate :} \quad \boldsymbol{f}_t = z_{rec}(\mathbf{W}_f \boldsymbol{x}_t + \mathbf{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{2.5}$$

$$\text{Input Gate :} \quad \boldsymbol{i}_t = z_{rec}(\mathbf{W}_i \boldsymbol{x}_t + \mathbf{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \tag{2.6}$$

$$\text{Output Gate :} \quad \boldsymbol{o}_t = z_{rec}(\mathbf{W}_o \boldsymbol{x}_t + \mathbf{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \tag{2.7}$$

$$\text{Cell Gate Input :} \quad \tilde{\boldsymbol{c}}_t = z_{cell}(\mathbf{W}_c \boldsymbol{x}_t + \mathbf{U}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \tag{2.8}$$

$$\text{Cell Gate :} \quad \boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \tag{2.9}$$

$$\text{Hidden State :} \quad \boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = \boldsymbol{o}_t \odot z_{out}(\boldsymbol{c}_t) \tag{2.10}$$

where $\odot$ *denotes Hadamard product,* $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_c \in \mathbb{R}^{n \times d}$, $\mathbf{U}_f, \mathbf{U}_i, \mathbf{U}_o, \mathbf{U}_c \in \mathbb{R}^{n \times n}$ *and* $\boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_o, \boldsymbol{b}_c \in \mathbb{R}^n$. *Typically,* z_{rec} *is chosen to be a sigmoid function while* z_{cell}, z_{out} *are chosen to be hyperbolic tangent functions.*

A gated recurrent unit (GRU) was proposed in [Cho et al., 2014] to make each recurrent unit adaptively capture dependencies of different time scales [Chung et al., 2014]. GRUs are very similar to LSTMs, with the difference that GRUs remove separate memory cells.

Definition 21. *A* gated recurrent unit network *(GRU) with* n *hidden states is a recurrent model* $R = (g, \psi, \boldsymbol{h}_0)$ *with input space* $\mathcal{X} = (\mathbb{R}^d)^*$ *and output space* $\mathcal{Y} = \mathbb{R}^p$. *It computes a function* $f_R : (\mathbb{R}^d)^* \to \mathbb{R}^p$ *defined by* $f_R(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) = \psi(\boldsymbol{h}_k)$, *where the transition function* $g : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$ *is defined by:*

$$\text{Update Gate :} \quad \boldsymbol{u}_t = z_{rec}(\mathbf{W}_u \boldsymbol{x}_t + \mathbf{U}_u \boldsymbol{h}_{t-1} + \boldsymbol{b}_u) \tag{2.11}$$

$$\text{Reset Gate :} \quad \boldsymbol{r}_t = z_{rec}(\mathbf{W}_r \boldsymbol{x}_t + \mathbf{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{2.12}$$

$$\text{Candidate Vector :} \quad \hat{\boldsymbol{h}}_t = z_{out}(\mathbf{W}_h \boldsymbol{x}_t + \mathbf{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{2.13}$$

$$\text{Output Vector :} \quad \boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = \boldsymbol{u}_t \odot \hat{\boldsymbol{h}}_t + (1 - \boldsymbol{u}_t) \odot \boldsymbol{h}_{t-1} \tag{2.14}$$

where $\mathbf{W}_u, \mathbf{W}_r, \mathbf{W}_h \in \mathbb{R}^{n \times d}$, $\mathbf{U}_u, \mathbf{U}_r, \mathbf{U}_h \in \mathbb{R}^{n \times n}$ *and* $\boldsymbol{b}_u, \boldsymbol{b}_r, \boldsymbol{b}_h \in \mathbb{R}^n$. *Typically,* z_{rec} *is chosen to be sigmoid function while* z_{cell}, z_{out} *are chosen to be hyperbolic tangent function.*

Chapter 3

Efficient Planning under Partial Observability with Unnormalized Q Functions and Spectral Learning

The general formulation of State Space Models (SSMs) involves two stages: the transition between states and the final output emission through an output function of the final state representation. A key objective in learning a good state representation in SSMs is to achieve a representation that captures sufficient information about the system dynamics, such as temporal dependencies and memories, in a manner that makes the final output function simple and easy to optimize. In model-based reinforcement learning, many SSMs, such as Predictive State Representations (PSRs), learn the state representations in an unsupervised manner and then use the learned model to plan and form a policy. However, this learning paradigm can lead to difficulties in obtaining an informative representation, as it disconnects the reward information from the learning of the environment model, resulting in an inefficient planning phase. In this chapter, we address this issue by connecting the learning and planning phases using the unnormalized Q function (UQF) and propose a spectral learning algorithm for the UQF. This chapter is based on my publication [Li et al., 2020a].

3.1 Introduction

A common assumption in reinforcement learning (RL) is that the agent has the knowledge of the entire dynamics of the environment, including the state space, transition probabilities, and a reward model. However, in many real world applications, this assumption may not always be valid. Instead, the environment is often *partially observable*, meaning that the true state of the system is not completely visible to the agent. This partial observability can result in numerous difficulties in terms of learning the dynamics of the environment and planning to maximize returns.

Partially observable Markov decision Processes (POMDPs) [Sondik, 1978, Cassandra et al., 1994] provide a formal framework for single-agent planning under a partially observable environment. In contrast with MDPs, agents in POMDPs do not have direct access to the state space. Instead of observing the states, agents only have access to observations and need to operate on the so-called *belief states*, which describe the distribution over the state space given some past trajectory. Therefore, POMDPs model the dynamics of an RL environment in a latent variable fashion and explicitly reason about uncertainty in both action effects and state observability [Boots et al., 2011]. Planning under a POMDP has long been considered a difficult problem [Kaelbling et al., 1998]. To perform *exact* planning under a POMDP, one common approach is to optimize the value function over all possible belief states. Value iteration for POMDPs [Sondik, 1978] is one particular example of this approach. However, due to the curse of dimensionality and the curse of history [Pineau et al., 2006], this method is often computationally intractable for most realistic POMDP planning problems [Boots et al., 2011].

As an alternative to exact planning, the family of predictive state representations (PSRs) has attracted many interests. In fact, PSRs are no weaker than POMDPs in terms of their representation power [Littman and Sutton, 2002], and there are many efficient algorithms to estimate PSRs and their variants relying on likelihood based algorithms [Singh et al., 2003, 2004] or spectral learning techniques [Boots et al., 2011, Hamilton et al., 2013]. How-

ever, to plan with PSRs is not straightforward. Typically, a two-stage process is applied to discover the optimal policy with PSRs: first, the model that describes the environment needs to be estimated. This includes learning a PSR model in an unsupervised fashion, and a reward function based on the learned model. After this stage, a planning method is used to discover the optimal policy. Several planning algorithms can be used for the second stage of this process. For example, in [Boots et al., 2011, Izadi and Precup, 2008], point based value iteration (PBVI) [Pineau et al., 2003] is used to obtain an approximation of the value function and hence the optimal policy; in [Hamilton et al., 2014], the authors use the fitted-Q method [Ernst et al., 2005] to iteratively regress Bellman updates on the learned state representations, thus approximating the action value function.

However, despite numerous successes, this two-stage process still suffers from significant drawbacks. To begin with, the PSRs parameters are learned independently from the reward information, resulting in a less efficient representation for planning. Secondly, planning with PSRs often involves multiple stages of regression, and these extra steps of approximation can be detrimental for obtaining the optimal policy. Last but not least, the planning methods for PSRs are often iterative methods that can be very time consuming.

In this work, we propose an alternative to the traditional paradigm of planning in partially observable environments. Inspired by PSRs, our method leverages the spectral learning algorithm for subspace identification, treating the environment as a latent variable model. However, instead of explicitly learning the dynamics of the environment, we learn a function that is proportional to the action value function, which we call *unnormalized Q function* (UQF). In doing so, we incorporate the reward information into the dynamics in a supervised learning fashion, which unifies the two stages of the classic learning-planning paradigm for POMDPs. To some extent, our approach effectively learns a goal-oriented representation of the environment and therefore is more sample efficient compared to the classic methods. Our algorithm relies on the spectral learning algorithm for *weighted finite automata* (WFAs), which are an extension of PSRs that can model not only probability distributions but arbitrary real-valued functions. Our method

inherits the benefits of spectral learning: it provides a consistent estimation of the UQF and is computationally more efficient than EM-based methods. Furthermore, planning with PSRs usually requires multiple steps and is often based on iterative methods, which can be time consuming. In contrast, our algorithm directly learns a policy in one step, offering a more time efficient method. In addition, we also adopt *matrix compressed sensing* techniques to extend this approach to complex domains. This technique has also been used in PSRs based methods to overcome similar problems [Hamilton et al., 2014].

We conduct experiments on partially observable grid world and S-PocMan environment [Hamilton et al., 2014] where we compare our approach with classical PSR based methods. In both domains, our approach is significantly more data-efficient than PSR based methods with considerably smaller running time.

3.2 Methodology

In this section, we will introduce our POMDP planning method. The main idea of our algorithm is to directly compute the optimal policy based on the estimation of *unnormalized Q function* that is proportional to the action value function. Moreover, the value of this function, given a past trajectory, can be computed via a WFA and it is then straightforward to use the classical spectral learning algorithm to recover this WFA. Unlike traditional PSR methods, our approach takes advantage of the reward information by integrating the reward into the learned representations. In contrast, classical PSRs based methods construct the representations solely with the environment dynamics, completely ignoring the reward information. Consequently, our method offers a more sample efficient representation of the environment for planning under POMDPs. In addition, our algorithm only needs to construct a WFA and there is no other iterative method involved. Therefore, compared to traditional methods to plan with PSRs, our algorithm is more time efficient. Finally, with the help of compressed sensing techniques, we are able to scale our algorithm to complex domains.

3.2.1 Unnormalized Q function

The estimation of the action value function is of great importance for planning under POMDP. Typically, given a probabilistic sampling policy $\Pi : \mathscr{A} \times \Sigma^* \to [0, 1]$, where $\Sigma = \mathscr{A} \times \mathscr{O}$, the action value function (Q function) of a given trajectory $h \in \Sigma^*$ is defined by:

$$Q^{\Pi}(h, a) = \mathbb{E}^{\Pi}(r_t + \gamma r_{t+1} + \cdots + \gamma^i r_{t+i} + \cdots | ha)$$

where $|h| = t$ and r_t is the immediate rewards collected at time step t.

Given a POMDP $\psi = \langle \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{O}}, \mathbf{r}, \mathscr{A}, \mathscr{O}, \mathcal{S}, \boldsymbol{\mu}, \gamma \rangle$, denote the expected immediate reward collected after h by $\tilde{R}(h)$, which is defined as:

$$\tilde{R}(h) = \mathbb{E}_{s \in \mathcal{S}}(\boldsymbol{r}_s | h) = \sum_{s \in \mathcal{S}} \boldsymbol{r}_s \mathbb{P}(s|h)$$

The action value function can then be expanded to:

$$\begin{aligned}
Q^{\Pi}(h, a) &= \mathbb{E}^{\Pi}(r_t + \gamma r_{t+1} + \cdots + \gamma^i r_{t+i} + \cdots | ha) \\
&= \sum_{z \in \Sigma^*} \sum_{o \in \mathscr{O}} \gamma^{|z|} \tilde{R}(haoz) \mathbb{P}^{\Pi}(haoz | ha) \\
&= \frac{\sum_{o \in \mathscr{O}} \sum_{z \in \Sigma^*} \gamma^{|z|} \tilde{R}(haoz) \mathbb{P}^{\Pi}(haoz)}{\mathbb{P}^{\Pi}(ha)} \\
&= \frac{\sum_{o \in \mathscr{O}} \sum_{z \in \Sigma^*} \gamma^{|z|} \tilde{R}(haoz) \mathbb{P}^{\Pi}(haoz) / \Pi(a|h)}{\mathbb{P}^{\Pi}(h)} \\
&:= \frac{\tilde{Q}^{\Pi}(h, a)}{\mathbb{P}^{\Pi}(h)}
\end{aligned}$$

where we will refer to the function $\tilde{Q}^{\Pi}(h, a)$ as the *unnormalized Q function*(UQF). It is trivial to show that given the same trajectory h:

$$\tilde{Q}^{\Pi}(h, \cdot) \propto Q^{\Pi}(h, \cdot)$$

Therefore, we have $\arg\max_{a\in\mathscr{A}} Q^{\Pi}(h,a) = \arg\max_{a\in\mathscr{A}} \tilde{Q}^{\Pi}(h,a)$ and we can then plan according to the UQF instead of Q^{Π}.

3.2.2 A spectral learning algorithm for UQF

In this section, we will present our spectral learning algorithm for UQF. First, we will show that the value of a UQF given a past trajectory can be computed via a WFA. Let us denote $\sum_{z\in\Sigma^*}\gamma^{|z|}\tilde{R}(hz)\mathbb{P}^{\Pi}(hz)$ by $\tilde{V}^{\Pi}(h)$, we have:

$$\tilde{Q}^{\Pi}(h,a) = \frac{\sum_{o\in\mathscr{O}}\tilde{V}^{\Pi}(hao)}{\Pi(a|h)}$$

Assume the probabilistic sampling policy Π is given, then we only need to compute the value of the function $\tilde{V}^{\Pi}(hao)$. As a special case, if Π is a random policy that uniformly select the actions, we can replace the term $\Pi(a|h)$ by 1 without affecting the learned policy.

It turns out that the function $\tilde{V}^{\Pi}$ can be computed by a WFA. To show that this is true, we first introduce the following lemma stating that the function $\tilde{R}(h)\mathbb{P}^{\Pi}(h)$ can be computed by a WFA B:

Lemma 1. *Given a POMDP $\psi = \langle \boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{O}}, \mathbf{r}, \mathscr{A}, \mathscr{O}, S, \boldsymbol{\mu}, \gamma\rangle$ of size k and a sampling policy Π induced by $\boldsymbol{\Pi} \in [0,1]^{\mathcal{S}\times\mathscr{A}}$, there exists a WFA $B = \langle \boldsymbol{\beta}, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\tau}\rangle$ with k states that realizes the function $g(h) = \tilde{R}(h)\mathbb{P}^{\Pi}(h)$, where $\Sigma = \mathscr{A}\times\mathscr{O}$ and $h\in\Sigma^*$.*

Proof. Let s^i denote the i^{th} state and let

$$\tilde{\mathbf{O}}_{ao} = \operatorname{diag}(\boldsymbol{\mathcal{O}}_{s^1,a,o}, \boldsymbol{\mathcal{O}}_{s^2,a,o}, \cdots, \boldsymbol{\mathcal{O}}_{s^k,a,o}),$$

$$\tilde{\mathbf{M}}_a = \operatorname{diag}(\boldsymbol{\Pi}_{s^1,a}, \boldsymbol{\Pi}_{s^2,a}, \cdots, \boldsymbol{\Pi}_{s^k,a}).$$

We can construct a WFA $B = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\tau}\rangle$ such that: $\boldsymbol{\beta}^\top = \boldsymbol{\mu}^\top$, $\mathbf{B}_\sigma = \mathbf{B}_{ao} = \tilde{\mathbf{M}}_a\boldsymbol{\mathcal{T}}_{:,a,:}\tilde{\mathbf{O}}_{ao}$, $\boldsymbol{\tau} = \mathbf{r}$. Then by construction, one can check that the WFA B computes the function g, which also shows that the rank of the function g is at most k. □

In fact, we can show that the function $\tilde{V}^{\Pi}$ can be computed by another WFA A, and one can easily convert B to A.

Theorem 5. *Given a POMDP ψ of size k, a sampling policy Π and a WFA $B = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\tau}\rangle$ realizing the function $g : h \mapsto \tilde{R}(h)\mathbb{P}^{\Pi}(h)$ such that the spectral radius $\rho(\gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma) < 1$, the WFA $A = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, (\mathbf{I} - \gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma)^{-1}\boldsymbol{\tau}\rangle$ of size k realizes the function $\tilde{V}^{\Pi}(h) = \sum_{z\in\Sigma^*}\gamma^{|z|}\tilde{R}(hz)\mathbb{P}^{\Pi}(hz)$.*

Proof. By definition of the function $\tilde{V}^{\Pi}$, we have:

$$\begin{aligned}
\tilde{V}^{\Pi}(h) &= \sum_{z\in\Sigma^*}\gamma^{|z|}\tilde{R}(hz)\mathbb{P}^{\Pi}(hz) \\
&= \sum_{z\in\Sigma^*}\gamma^{|z|}\boldsymbol{\beta}^\top\mathbf{B}_h\mathbf{B}_z\boldsymbol{\tau} \\
&= \boldsymbol{\beta}^\top\mathbf{B}_h(\sum_{z\in\Sigma^*}\gamma^{|z|}\mathbf{B}_z)\boldsymbol{\tau} \\
&= \boldsymbol{\beta}^\top\mathbf{B}_h(\sum_{i=0}^{\infty}(\gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma)^i)\boldsymbol{\tau} \\
&= \boldsymbol{\beta}^\top\mathbf{B}_h(\mathbf{I} - \gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma)^{-1}\boldsymbol{\tau}
\end{aligned}$$

Here we applied Neumann identity: $\sum_{i=0}^{\infty}\mathbf{T}^i = (\mathbf{I} - \mathbf{T})^{-1}$, which holds when $\rho(\mathbf{T}) < 1$. Therefore, the WFA $A = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, (\mathbf{I} - \gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma)^{-1}\boldsymbol{\tau}\rangle$ realizes the function $\tilde{V}^{\Pi}$. □

Note the condition on the spectral radius implies that the function $\tilde{V}$ needs be bounded. This condition is in fact very easy to satisfy as in RL we typically assume that the value function $V(h) = \sum_{z\in\Sigma^*}\gamma^{|z|}\tilde{R}(hz)\mathbb{P}(z|h)$ is bounded.

Therefore, in order to compute the function $\tilde{Q}^{\Pi}$, we only need to learn a WFA that computes the function g. Following the classical spectral learning algorithm, we present our learning algorithm of POMDP planning in Algorithm 2. In fact, it has been shown that the spectral learning algorithm of WFAs is statistically consistent [Balle et al., 2014a]. Therefore our approximation of the function $\tilde{Q}^{\Pi}$ is consistent with respect to sample sizes.

Algorithm 2 Spectral algorithm for UQF

Input: A set of actions $\mathscr{A}$, a set of observations $\mathscr{O}$, discount factor γ, a probabilistic sampling policy Π, training trajectories D and their immediate reward $\boldsymbol{y}$, rank of the truncated SVD k.

Output: A new deterministic policy function $\pi^{new} : \Sigma^* \to \mathscr{A}$.

1. For a prefix u and a suffix v, we estimate its value in the Hankel matrix as $\hat{\mathbf{H}}_{u,v} = \sum_{i=0}^{|D|} \mathbb{I}_{uv}(D_i)\boldsymbol{y}_i/|D|$, where $|D|$ is the cardinality of the training set D, $\Sigma = \mathscr{A} \times \mathscr{O}$.
2. Perform truncated SVD of rank k on the estimated Hankel matrix: $\hat{\mathbf{H}} \simeq \mathbf{U}\mathbf{D}\mathbf{V}^\top$
3. Recover the WFA $B = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\tau}\rangle$ realizing the function $g(h) = \tilde{R}(h)\mathbb{P}(h)$: $\boldsymbol{\beta}^\top = (\mathbf{U}\mathbf{D})_{\lambda,:}$, $\boldsymbol{\tau} = \mathbf{V}^\top_{:,\lambda}$, $\mathbf{B}_\sigma = (\mathbf{U}\mathbf{D})^+\hat{\mathbf{H}}_\sigma\mathbf{V}^\top$
4. Convert the WFA B to $A = \langle \boldsymbol{\alpha}^\top, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega}\rangle$, which realizes the function $\tilde{V}^\Pi$, following Theorem 5, we have $\boldsymbol{\alpha}^\top = \boldsymbol{\beta}^\top$, $\mathbf{A}_\sigma = \mathbf{B}_\sigma$, and $\boldsymbol{\omega} = (\mathbf{I} - \gamma\sum_{\sigma\in\Sigma}\mathbf{B}_\sigma)^{-1}\boldsymbol{\tau}$.
5. **Return** A new deterministic policy function π^{new}, such that given $h \in \Sigma^*$, $\pi^{new}(h) = \arg\max_{a\in\mathscr{A}} \frac{\sum_{o\in\mathscr{O}} \boldsymbol{\alpha}^\top \mathbf{A}_h \mathbf{A}_{ao} \boldsymbol{\omega}}{\Pi(a|h)}$

3.2.3 Scalable learning of UQF

Now we have established the spectral learning algorithm for UQF. However, similar to the spectral learning algorithm for TPSRs, one can immediately observe that both time and storage complexity are the bottleneck of this algorithm. For complex domains, in order to obtain a complete sub-block of the Hankel matrix, one will need large amount of prefixes and suffixes to form a basis and the classical spectral learning will become intractable.

By projecting matrices down to low-dimensional spaces via randomly generated bases, *matrix compressed sensing* has been widely applied in matrix compression field. In fact, previous work have successfully applied matrix sensing techniques to TPSRs [Hamilton et al., 2013] and developed an efficient online algorithm for learning TPSRs [Hamilton et al., 2014]. Here, we adopt a similar approach.

Assume that we are given a set of prefixes $\mathscr{U}$ and suffixes $\mathscr{V}$ and two independent random full-rank Johnson-Lindenstrauss (JL) projection matrices $\boldsymbol{\Phi}_\mathscr{U} \in \mathbb{R}^{\mathscr{U}\times d_\mathscr{U}}$, and $\boldsymbol{\Phi}_\mathscr{V} \in \mathbb{R}^{\mathscr{V}\times d_\mathscr{V}}$, where $d_\mathscr{U}$ and $d_\mathscr{V}$ are the projection dimension for the prefixes and suffixes. In this

Algorithm 3 Scalable spectral algorithm for UQF

Input: A set of actions $\mathscr{A}$, a set of observations $\mathscr{O}$, discount factor γ, a probabilistic sampling policy Π, training trajectories D and their immediate reward $\boldsymbol{y} \in \mathbb{R}^{|D|}$, the rank of the truncated SVD k, a set of prefixes $\mathscr{U}$, a set of suffixes $\mathscr{V}$, mapping functions for both prefixes and suffixes, ϕ_U, ϕ_V, and the corresponding projection matrix $\boldsymbol{\Phi}_{\mathscr{U}}$.

Output: A new deterministic policy function $\pi^{new} : \Sigma^* \to \mathscr{A}$.

1. Compute the compressed estimation of Hankel matrices:

$$\hat{\mathbf{c}}_{\mathscr{U}} = \sum_{i=0}^{|D|} \sum_{u \in \mathscr{U}} \mathbb{I}_v(D_i) \boldsymbol{y}_i \phi_U(u)$$

$$\hat{\mathbf{C}}_{\mathscr{U}\mathscr{V}} = \sum_{i=0}^{|D|} \sum_{u,v \in \mathscr{U}\times\mathscr{V}} \mathbb{I}_{uv}(D_i) \boldsymbol{y}_i (\phi_U(u) \otimes \phi_V(v))$$

2. Perform truncated SVD on the estimated Hankel matrix with rank k: $\hat{\mathbf{C}}_{\mathscr{U}\mathscr{V}} \simeq \mathbf{U}\mathbf{D}\mathbf{V}^\top$

3. Recover the WFA $B = \langle \boldsymbol{\beta}^\top, \{\mathbf{B}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\tau} \rangle$ realizing the function $g(h) = \tilde{R}(h)\mathbb{P}(h)$:

$$\boldsymbol{\beta}^\top = \boldsymbol{e}^\top \mathbf{U}\mathbf{D}, \qquad \boldsymbol{\tau} = \mathbf{D}^{-1}\mathbf{U}^\top \hat{\mathbf{c}}_{\mathscr{U}}$$

$$\mathbf{B}_\sigma = \sum_{i=0}^{|D|} \sum_{u,v \in \mathscr{U}\times\mathscr{V}} \mathbb{I}_{u\sigma v}(D_i) \boldsymbol{y}_i [\mathbf{D}^{-1}\mathbf{U}^\top \phi(u) \otimes \mathbf{V}^\top \phi(v)]$$

   where $\boldsymbol{e}$ is a vector s.t. $\boldsymbol{e}^\top \boldsymbol{\Phi}_{\mathscr{U}}^\top = (1, 0, \cdots, 0)^\top$

4. Following Theorem 5, convert the WFA B to $A = \langle \boldsymbol{\alpha}^\top, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega} \rangle$.

5. **Return** A new deterministic policy function π^{new} defined by $\pi^{new}(h) = \arg\max_{a\in\mathscr{A}} \frac{\sum_{o\in\mathscr{O}} \boldsymbol{\alpha}^\top \mathbf{A}_h \mathbf{A}_{ao} \boldsymbol{\omega}}{\Pi(a|h)}$

work, we use Gaussian projection matrices for $\boldsymbol{\Phi}_{\mathscr{U}}$ and $\boldsymbol{\Phi}_{\mathscr{V}}$, which contain i.i.d. entries from the distribution $\mathcal{N}(0, 1/d_{\mathscr{U}})$ and $\mathcal{N}(0, 1/d_{\mathscr{V}})$, respectively.

Let us now define two injective functions over prefixes and suffixes: $\phi_{\mathscr{U}} : \mathscr{U} \to \mathbb{R}^{d_{\mathscr{U}}}$ and $\phi_{\mathscr{V}} : \mathscr{V} \to \mathbb{R}^{d_{\mathscr{V}}}$, where for all $u \in \mathscr{U}$ and $v \in \mathscr{V}$, we have $\phi(u) = \boldsymbol{\Phi}_{u,:}$ and $\phi(v) = \boldsymbol{\Phi}_{v,:}$. The core step of our algorithm is to obtain the compressed estimation of the Hankel matrix, denoted by $\hat{\mathbf{C}}_{\mathscr{U},\mathscr{V}}$ associated with the function $\tilde{R}(h)\mathbb{P}(h)$ for all $h \in \Sigma^*$. Formally, we can

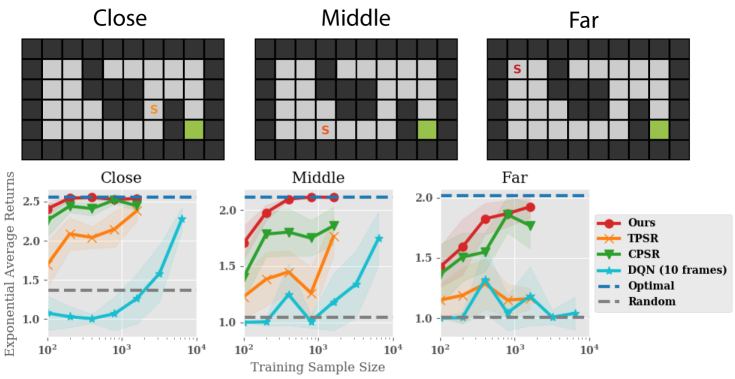


Figure 3.1: Experiments on three grid world tasks. The plots show the accumulated discounted rewards (returns) over 1,000 test episodes of length 100. The discount factor for computing returns is set to 0.99

obtain $\hat{\mathbf{C}}_{\mathcal{U},\mathcal{V}}$ by:

$$\begin{aligned}\hat{\mathbf{C}}_{\mathcal{U},\mathcal{V}} &= \mathbf{\Phi}_{\mathcal{U}}^{\top}\mathbf{H}\mathbf{\Phi}_{\mathcal{V}} \\ &= \sum_{i=0}^{|D|}\sum_{u,v\in\mathcal{U}\times\mathcal{V}} \mathbb{I}_{uv}(D_i)\boldsymbol{y}_i(\phi_U(u)\otimes\phi_V(v))\end{aligned}$$

where D is the training dataset, containing all sampled trajectories, $\boldsymbol{y}$ is the vector of immediate rewards. Then, after performing the truncated SVD of $\hat{\mathbf{C}}_{\mathcal{U},\mathcal{V}} \simeq \mathbf{U}\mathbf{D}\mathbf{V}^{\top}$ of rank k, we can compute the transition matrix for the WFA by:

$$\begin{aligned}\mathbf{B}_{\sigma} &= (\mathbf{U}\mathbf{D})^{+}\hat{\mathbf{C}}_{\mathcal{U}\sigma\mathcal{V}}\mathbf{V} \\ &= \sum_{i=0}^{|D|}\sum_{u,v\in\mathcal{U}\times\mathcal{V}} \mathbb{I}_{u\sigma v}(D_i)\boldsymbol{y}_i[(\mathbf{U}\mathbf{D})^{+}\phi(u)\otimes\mathbf{V}^{\top}\phi(v)]\end{aligned}$$

We present the complete method in Algorithm 3. Instead of iterative sweeping through datasets like most planning methods do, one can build an UQF in just two passes of data: one for building the compressed Hankel, one for recovering the parameters. More precisely, let L denote the maximum length of a trajectory in the dataset D, then the time complexity of our algorithm is $O(L|D|)$ [Hamilton et al., 2014], and there is no extra planning time needed. In contrast, fitted-Q algorithm alone requires $O(TL|D|log(L|D|))$ only

for the planning stage, where T is the expected number of the fitted-Q iterations. Therefore, in terms of time complexity, our algorithm is linear to the number of trajectories, leading to a very efficient algorithm.

3.2.4 Policy iteration

Policy iteration has been widely applied in both MDP and POMDP settings [Bellman, 1957a, Sutton et al., 1998], and have shown benefits from both empirical and theoretical perspectives [Bellman, 1957a]. It is very natural to apply policy iteration to our algorithm, since we directly learn a policy from data. The policy iteration algorithm is listed in Algorithm 4. Note that for re-sampling, we convert our learned deterministic policy to a probabilistic one in an ϵ-greedy fashion.

Algorithm 4 Policy iteration for UQF

Input: An initial deterministic policy π, ϵ-greedy factor ϵ, a decay rate for ϵ-greedy $\eta > 1$, number of policy iterations n, number of trajectories N.

Output The final policy function $\pi^{fin} : \Sigma^* \to \mathscr{A}$.

1. Convert the deterministic policy π to a probabilistic policy Π in an ϵ-greedy fashion: at each step, with probability $1 - \epsilon$ select the optimal action according to π, with probability ϵ select a random action.*
2. Sample N trajectories based on policy Π
3. Execute Algorithm 2 or Algorithm 3 and obtain the corresponding new policy π^{new}.
4. $\pi^{new} \to \pi$, $\epsilon/\eta \to \epsilon$
5. Repeat all the above for n times.
6. **Return** The final policy function $\pi^{fin} = \pi^{new}$

*Note for the first iteration, one can set $\epsilon = 1$, resulting in a pure random sampling policy.

3.3 Experiments

To assess the performance of our method, we conducted experiments on two domains: a toy grid world environment and the S-Pocman game [Hamilton et al., 2014]. We use TPSR/CPSR + fitted-Q as our baseline method. Our experiments have shown that indeed, in terms of sample complexity and time complexity, we outperm the classical two-stage algorithms.

3.3.1 Grid world experiment

The first benchmark for our method is the simple grid world environment shown in Fig. 3.1. The agent starts in the tile labeled S and must reach the green goal state. At each time step, the agent can only perceive the number of surrounding walls and proceeds to execute one of the four actions: go up, down, left or right. To make the environment stochastic, with probability 0.2, the execution of the action will fail, resulting instead in a random action at the current time step. The reward function in this navigation task is sparse: the agent receives no reward until it reaches the termination state. We ran three variants of the aforementioned grid world, each corresponding to a different starting state. As one can imagine, the further away the goal state is from the starting state, the harder the task becomes.

We used a random policy to generate training data, which consisted of trajectories of length up to 100. To evaluate the policy learned by the different algorithms, we let the agent execute the learned policy for 1,000 episodes and computed the average accumulated discounted rewards, with discount factor being 0.99. The maximum length for test episodes was also set to 100. Hyperparameters were selected using cross-validation (i.e. the number of rows and columns in the Hankel matrices, the rank for SVD and γ). As a baseline, we use the classical TPSRs and CPSRs as the learning method for the environment, and fitted-Q algorithm as the planning algorithm, as well as a deep Q-network (DQN). We use a three layers MLP of size 50*128*128 to approximate the Q function and

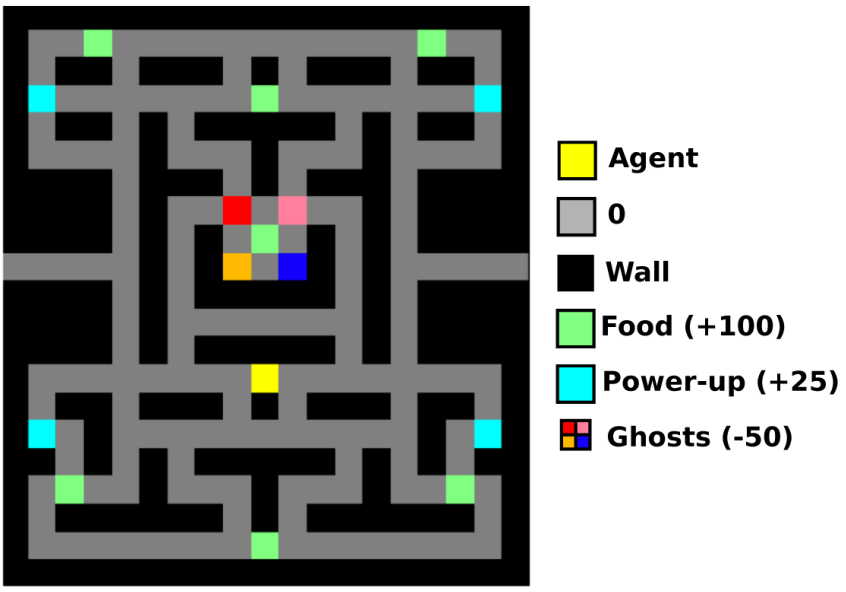


Figure 3.2: S-PocMan domain

Adam optimizer with 10^{-4} learning rate. The DQN is fitted on 10 consecutive observation vectors (number of walls around agent, same as all other methods) and tested with 1,000 test episodes of maximum length 100, as to reproduce the same experimental setup as the other methods. We performed a hyperparameter search for all baseline methods using cross validation. In addition, we report the rewards collected by a random policy as well as the optimal policy for comparison.

Results on this toy domain (see Figure 3.1) highlight the sample and time efficiency achieved by our method. Indeed, our algorithm outperforms the classical CPSR+fitted-Q method in all three domains, notably achieving better performance in small data regime, showing significant sample efficiency. Furthermore, it is clear that our algorithm reaches consistently to the optimal policy as sample size increases. In addition, our methods are much faster than other compared methods. For example, for the experiment with 800 samples, to achieve similar results, our method is approximately 100 times faster compared to CPSR+fitted-Q. As expected, DQN requires a lot more data to achieve competitive results on the partially observable grid world domain than the spectral approaches.

Table 3.1: Training time for one policy iteration and averaged accumulated discounted rewards on S-PocMan trained on 500 trajectories.

Method	Fitted-Q Iterations	Time (s)	Returns
UQF	-	**2**	**-92**
CPSR	400	489	-101
	100	116	-109
	50	60	-150
	10	15	-200

3.3.2 S-PocMan domain

For the second experiment, we show the results on the S-PocMan environment [Hamilton et al., 2014]. The partially observable version of the classical game Pacman was first introduced by Silver and Veness [Silver and Veness, 2010] and is referred to as PocMan. In this domain, the agent needs to navigate through a grid world to collect food and avoid being captured by the ghosts. It is an extremely large partially observable domain with 10^{56} states [Veness et al., 2011]. However, Hamilton et al. showed that if one were to treat the partially observable environment as if it was fully observable, a simple memoryless controller can perform extremely well under this set-up, due to extensive reward information [Hamilton et al., 2014]. Hence, they proposed a harder version of PocMan, called S-PocMan. In this new domain, they drop the parts of the observation vector that allow the agent to sense the direction of the food and greatly sparsify the amount of food in the grid world, therefore making the environment more partially observable. In this experiment, we only used the combination CPSR+fitted Q for our baseline algorithm, as TPSR can not scale to the large size of this environment. Similarly to the grid world experiment, we select the best hyperparameters through cross validation. The discount factor for computing returns was set to be 0.99 in all runs. Table 3.1 shows the run-time and average return for both our algorithm and the baseline method. One can see that UQF achieves better performance compared to CPSR+fitted-Q. Moreover, UQF exhibits a significant reduction in running time: about 200 times faster than CPSR+fitted-Q. Note that building CPSR takes similar amount of time to our method, however, the extra iterative fitted-Q

planning algorithm takes considerably more time to converge, as our analysis showed in section 3.3.

3.4 Conclusion

In this chapter, we propose a novel learning and planning algorithm for partially observable environments. The main idea of our algorithm relies on the estimation of the unnormalized Q function with the spectral learning algorithm. Theoretically, we show that in POMDP, UQF can be computed via a WFA and consequently can be provably learned from data using the spectral learning algorithm for WFAs. Moreover, UQF combines the learning and planning phases of reinforcement learning together, and learns the corresponding policy in one step. Therefore, our method is more sample efficient and time efficient compared to traditional POMDP planning algorithms. This is further shown in the experiments on the grid world and S-PocMan environments.

Future work includes exploring some theoretic properties of this planning approach. For example, a first step would be to obtain convergence guarantees for policy iteration based on the UQF spectral learning algorithm. In addition, our approach could be extended to the multitask setting by leveraging the multi-task learning framework for WFAs proposed in [Rabusseau et al., 2017]. Readily, since we combine the environment dynamics and reward information together, our approach should be able to deal with partially shared environment and reward structure, leading to a potentially flexible multi-task RL framework.

Chapter 4

Connecting Weighted Automata, Tensor Networks and Recurrent Neural Networks through Spectral Learning

The spectral learning algorithm of Weighted Finite Automata (WFAs) is known for its desirable properties, including minimality, consistency, and sample efficiency. However, it is limited to modeling functions over discrete input variables, or strings. Many real-world applications of sequential data require modeling functions of series of continuous vectors. To handle such tasks, Recurrent Neural Networks (RNNs) and gradient descent are often used. Nevertheless, unlike spectral learning for WFAs, gradient descent for RNN models can struggle with overfitting in the low data regime due to the highly nonlinear formulation of the RNN structure. Additionally, it is challenging to obtain theoretical results for RNN learning using gradient descent due to the non-convexity of the functions. The question arises whether it is possible to extend WFAs to handle continuous input variables and develop a spectral learning algorithm for them. Furthermore, it is of interest to explore any connections between this extended WFA and existing RNN models.

In this chapter, we present connections between three models used in different research fields: weighted finite automata (WFA) from formal languages and linguistics,

recurrent neural networks used in machine learning, and tensor networks which encompass a set of optimization techniques for high-order tensors used in quantum physics and numerical analysis. We first present an intrinsic relation between WFA and the tensor train decomposition, a particular form of tensor network. This relation allows us to exhibit a novel low-rank structure of the Hankel matrix of a function computed by a WFA and to design an efficient spectral learning algorithm leveraging this structure to scale the algorithm up to very large Hankel matrices. We then unravel a fundamental connection between WFA and second-order recurrent neural networks (2-RNN): in the case of sequences of discrete symbols, WFA and 2-RNN with linear activation functions are expressively equivalent. Leveraging this equivalence result combined with the classical spectral learning algorithm for weighted automata, we introduce the first provable learning algorithm for linear 2-RNN defined over sequences of continuous input vectors. This algorithm relies on estimating low-rank sub-blocks of the Hankel tensor, from which the parameters of a linear 2-RNN can be provably recovered. The performances of the proposed learning algorithm are assessed in a simulation study on both synthetic and real-world data. This chapter is based on my publication [Li et al., 2022b].

4.1 Introduction

Many tasks in natural language processing [Devlin et al., 2018], computational biology [Tang et al., 2019], reinforcement learning [Kapturowski et al., 2019], and time series analysis [Connor et al., 1994] rely on learning with sequential data, i.e. estimating functions defined over sequences of observations from training data. Weighted finite automata (WFA) and recurrent neural networks (RNN) are two powerful and flexible classes of models which can efficiently represent such functions. On the one hand, WFA are tractable, they encompass a wide range of machine learning models (they can for example compute any probability distribution defined by a hidden Markov model (HMM) [Denis and Esposito, 2008] and can model the transition and observation behavior of partially observ-

able Markov decision processes [Thon and Jaeger, 2015]) and they offer appealing theoretical guarantees. In particular, the so-called *spectral methods* for learning HMMs [Hsu et al., 2009], WFA [Bailly et al., 2009, Balle et al., 2014a] and related models [Glaude and Pietquin, 2016, Boots et al., 2011], provide an alternative to Expectation-Maximization based algorithms that is both computationally efficient and consistent. On the other hand, RNN are remarkably expressive models — they can represent any computable function [Siegelmann and Sontag, 1992], given infinite precision arithmetic — and they have successfully tackled many practical problems in speech and audio recognition [Graves et al., 2013, Mikolov et al., 2011, Gers et al., 2000], but their theoretical analysis is difficult. Even though recent work provides interesting results on their expressive power [Khrulkov et al., 2018, Yu et al., 2017] as well as alternative training algorithms coming with learning guarantees [Sedghi and Anandkumar, 2016], the theoretical understanding of RNN is still limited.

At the same time, tensor networks are a generalization of tensor decomposition techniques, where complex operations between tensors are represented in a simple diagrammatic notation, allowing one to intuitively represent intricate ways to decompose a high-order tensor into lower-order tensors acting as *building block*. The term *tensor networks* also encompasses a set of optimization techniques to efficiently tackle optimization problems in very high-dimensional spaces, where the optimization variable is represented as a tensor network and the optimization process is carried out with respect to the *building blocks* of the tensor network. As an illustration, such optimization techniques make it possible to efficiently approximate the leading eigen-vectors of matrices of size $2^N \times 2^N$ where N can be as large as 50 [Holtz et al., 2012]. Tensor networks have emerged in the quantum physics community to model many-body systems [Orús, 2014, Biamonte and Bergholm, 2017] and have also been used in numerical analysis as a mean to solve high-dimensional differential equations [Oseledets, 2011, Lubich et al., 2013] and to design efficient algorithms for big data analytics [Cichocki et al., 2016]. Tensor networks have recently been used in the context of machine learning to compress neural networks [Novikov et al.,

2015, 2014, Ma et al., 2019, Yang et al., 2017], to design new approaches and optimization techniques borrowed from the quantum physics literature for supervised and unsupervised learning tasks [Stoudenmire and Schwab, 2016, Han et al., 2018, Miller et al., 2020], as new theoretical tools to understand the expressiveness of neural networks [Cohen et al., 2016, Khrulkov et al., 2018] and for image completion problems [Yang et al., 2017, Wang et al., 2017] among others.

In this work, we bridge a gap between these three classes of models: weighted automata, tensor networks and recurrent neural networks. We first exhibit an intrinsic relation between the computation of a weighted automata and the tensor train decomposition, a particular form of tensor network (also known as matrix product states in the quantum physics community). While such a connection has been sporadically noticed previously, we demonstrate how this relation implies a low tensor train structure of the so-called Hankel matrix of a function computed by a WFA. The Hankel matrix of a function is at the core of the spectral learning algorithm for WFA. This algorithm relies on the fact that the (matrix) rank of the Hankel matrix is directly related to the size of a WFA computing the function it represents. We show that, beyond being low rank, the Hankel matrix of a function computed by a WFA can be seen as a block matrix where each block is a matricization of a tensor with low tensor train rank. Building upon this result, we design an efficient implementation of the spectral learning algorithm that leverages this tensor train structure. When the Hankel matrices needed for the spectral algroithm are given in the tensor train format, the time complexity of the algorithm we propose is exponentially smaller (w.r.t. the size of the Hankel matrix) than the one of the classical spectral learning algorithm.

We then unravel a fundamental connection between WFA and second-order RNN (2-RNN): *when considering input sequences of discrete symbols, 2-RNN with linear activation functions and WFA are one and the same*, i.e. they are expressively equivalent and there exists a one-to-one mapping between the two classes (moreover, this mapping conserves model sizes). While connections between finite state machines (e.g. deterministic finite

automata) and recurrent neural networks have been noticed and investigated in the past (see e.g. [Giles et al., 1992, Omlin and Giles, 1996]), to the best of our knowledge this is the first time that such a rigorous equivalence between linear 2-RNN and *weighted* automata is explicitly formalized. More precisely, we pinpoint exactly the class of recurrent neural architectures to which weighted automata are equivalent, namely second-order RNN with linear activation functions. This result naturally leads to the observation that linear 2-RNN are a natural generalization of WFA (which take sequences of *discrete* observations as inputs) to sequences of *continuous vectors*, and raises the question of whether the spectral learning algorithm for WFA can be extended to linear 2-RNN. The third contribution of this chapter is to show that the answer is in the positive: building upon the classical spectral learning algorithm for WFA [Hsu et al., 2009, Bailly et al., 2009, Balle et al., 2014a] and its recent extension to vector-valued functions [Rabusseau et al., 2017], *we propose the first provable learning algorithm for second-order RNN with linear activation functions*. Our learning algorithm relies on estimating sub-blocks of the so-called Hankel tensor, from which the parameters of a 2-linear RNN can be recovered using basic linear algebra operations. One of the key technical difficulties in designing this algorithm resides in estimating these sub-blocks from training data where the inputs are sequences of *continuous* vectors. We leverage multilinear properties of linear 2-RNN and the tensor train structure of the Hankel matrix to perform this estimation efficiently using matrix sensing and tensor recovery techniques. In particular, we show that the Hankel matrices needed for learning can be estimated directly in the tensor train format, which allows us to use the efficient spectral learning algorithm in the tensor train format discussed previously.

We validate our theoretical findings in a simulation study on synthetic and real world data where we experimentally compare different recovery methods and investigate the robustness of our algorithm to noise. We also show that refining the estimator returned by our algorithm using stochastic gradient descent can lead to significant improvements.

Summary of contributions. We present *novel connections between WFA and the tensor train decomposition* (Section 4.3.1) allowing us to design an *highly efficient implementation of the spectral learning algorithm in the tensor train format* (Section 4.3.2). We formalize a *strict equivalence between weighted automata and second-order RNN with linear activation functions* (Section 4.4), showing that linear 2-RNN can be seen as a natural extension of (vector-valued) weighted automata for input sequences of *continuous* vectors. We then propose a *consistent learning algorithm for linear 2-RNN* (Section 4.5). The relevance of our contributions can be seen from three perspectives. First, while learning feed-forward neural networks with linear activation functions is a trivial task (it reduces to linear or reduced-rank regression), this is not at all the case for recurrent architectures with linear activation functions; to the best of our knowledge, our algorithm is the *first consistent learning algorithm for the class of functions computed by linear second-order recurrent networks*. Second, from the perspective of learning weighted automata, we propose a natural extension of WFA to continuous inputs and *our learning algorithm addresses the long-standing limitation of the spectral learning method to discrete inputs*. Lastly, by connecting the spectral learning algorithm for WFA to recurrent neural networks on one side, and tensor networks on the other, our work opens the door to leveraging highly efficient optimization techniques for large scale tensor problems used in the quantum physics community for designing new learning learning algorithms for both linear and non-linear sequential models, as well as offering new tools for the theoretical analysis of these models.

Related work. Combining the spectral learning algorithm for WFA with matrix completion techniques (a problem which is closely related to matrix sensing) has been theoretically investigated in [Balle and Mohri, 2012]. An extension of probabilistic transducers to continuous inputs (along with a spectral learning algorithm) has been proposed in [Recasens and Quattoni, 2013]. The model considered in this work is closely related to the continuous extension of WFA we consider here but the learning algorithm proposed in [Recasens and Quattoni, 2013] is designed for (and limited to) stochastic transducers,

whereas we consider arbitrary functions computed by linear 2-RNN. The connections between tensors and RNN have been previously leveraged to study the expressive power of RNN in [Khrulkov et al., 2018] and to achieve model compression in [Yu et al., 2017, Yang et al., 2017, Tjandra et al., 2017]. Exploring relationships between RNN and automata has recently received a renewed interest [Peng et al., 2018, Chen et al., 2018, Li et al., 2018, Merrill et al., 2020]. In particular, such connections have been explored for interpretability purposes [Weiss et al., 2018, Ayache et al., 2018] and the ability of RNN to learn classes of formal languages has been investigated in [Avcu et al., 2017]. Connections between the tensor train decomposition and WFA have been previously noticed in [Critch, 2013, Critch and Morton, 2014, Rabusseau, 2016]. However, to the best of our knowledge, this is the first time that the tensor-train structure of the Hankel matrix of a function computed by a WFA is noticed and leveraged to design an efficient spectral learning algorithm for WFA. Other approaches have been proposed to scale the spectral learning algorithm to large datasets, notably by identifying a small basis of informative prefixes and suffixes to build the Hankel matrices [Quattoni et al., 2017]. The predictive state RNN model introduced in [Downey et al., 2017] is closely related to 2-RNN and the authors propose to use the spectral learning algorithm for predictive state representations to initialize a gradient based algorithm; their approach however comes without theoretical guarantees. Lastly, a provable algorithm for RNN relying on the tensor method of moments has been proposed in [Sedghi and Anandkumar, 2016] but it is limited to first-order RNN with quadratic activation functions (which do not encompass linear 2-RNN).

4.2 Preliminaries

In this section, we first present basic notions of tensor algebra and tensor networks before introducing second-order recurrent neural network, weighted finite automata and the spectral learning algorithm. We start by introducing some notations. For any integer k we use $[k]$ to denote the set of integers from 1 to k. We use $\lceil l \rceil$ to denote the smallest

integer greater or equal to l. For any set S, we denote by $S^* = \bigcup_{k\in\mathbb{N}} S^k$ the set of all finite-length sequences of elements of S (in particular, Σ^* will denote the set of strings on a finite alphabet Σ). We use lower case bold letters for vectors (e.g. $\boldsymbol{v} \in \mathbb{R}^{d_1}$), upper case bold letters for matrices (e.g. $\mathbf{M} \in \mathbb{R}^{d_1\times d_2}$) and bold calligraphic letters for higher order tensors (e.g. $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d_1\times d_2\times d_3}$). We use $\boldsymbol{e}_i$ to denote the ith canonical basis vector of $\mathbb{R}^d$ (where the dimension d will always appear clearly from context). The $d \times d$ identity matrix will be written as $\mathbf{I}_d$. The ith row (resp. column) of a matrix $\mathbf{M}$ will be denoted by $\mathbf{M}_{i,:}$ (resp. $\mathbf{M}_{:,i}$). This notation is extended to slices of a tensor in the straightforward way. If $\boldsymbol{v} \in \mathbb{R}^{d_1}$ and $\boldsymbol{v}' \in \mathbb{R}^{d_2}$, we use $\boldsymbol{v} \otimes \boldsymbol{v}' \in \mathbb{R}^{d_1\cdot d_2}$ to denote the Kronecker product between vectors, and its straightforward extension to matrices and tensors. Given a matrix $\mathbf{M} \in \mathbb{R}^{d_1\times d_2}$, we use $\mathrm{vec}(\mathbf{M}) \in \mathbb{R}^{d_1\cdot d_2}$ to denote the column vector obtained by concatenating the columns of $\mathbf{M}$. The inverse of $\mathbf{M}$ is denoted by $\mathbf{M}^{-1}$, its Moore-Penrose pseudo-inverse by $\mathbf{M}^{\dagger}$, and the transpose of its inverse by $\mathbf{M}^{-\top}$; the Frobenius norm is denoted by $\|\mathbf{M}\|_F$ and the nuclear norm by $\|\mathbf{M}\|_*$.

4.2.1 Tensors and Tensor Networks

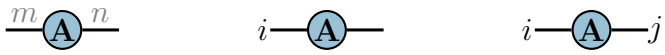
We first recall basic definitions of tensor algebra; more details can be found in [Kolda and Bader, 2009]. A *tensor* $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d_1\times\cdots\times d_p}$ can simply be seen as a multidimensional array $(\boldsymbol{\mathcal{T}}_{i_1,\cdots,i_p} : i_n \in [d_n], n \in [p])$. The *mode-$n$* fibers of $\boldsymbol{\mathcal{T}}$ are the vectors obtained by fixing all indices except the nth one, e.g. $\boldsymbol{\mathcal{T}}_{:,i_2,\cdots,i_p} \in \mathbb{R}^{d_1}$. The *$n$th mode matricization* of $\boldsymbol{\mathcal{T}}$ is the matrix having the mode-n fibers of $\boldsymbol{\mathcal{T}}$ for columns and is denoted by $\boldsymbol{\mathcal{T}}_{(n)} \in \mathbb{R}^{d_n\times d_1\cdots d_{n-1}d_{n+1}\cdots d_p}$. The vectorization of a tensor is defined by $\mathrm{vec}(\boldsymbol{\mathcal{T}}) = \mathrm{vec}(\boldsymbol{\mathcal{T}}_{(1)})$. In the following $\boldsymbol{\mathcal{T}}$ always denotes a tensor of size $d_1 \times \cdots \times d_p$.

The *mode-n matrix product* of the tensor $\boldsymbol{\mathcal{T}}$ and a matrix $\mathbf{X} \in \mathbb{R}^{m\times d_n}$ is a tensor denoted by $\boldsymbol{\mathcal{T}} \times_n \mathbf{X}$. It is of size $d_1 \times \cdots \times d_{n-1} \times m \times d_{n+1} \times \cdots \times d_p$ and is defined by the relation $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{T}} \times_n \mathbf{X} \Leftrightarrow \boldsymbol{\mathcal{Y}}_{(n)} = \mathbf{X}\boldsymbol{\mathcal{T}}_{(n)}$. The *mode-$n$ vector product* of the tensor $\boldsymbol{\mathcal{T}}$ and a vector $\boldsymbol{v} \in \mathbb{R}^{d_n}$ is a tensor defined by $\boldsymbol{\mathcal{T}} \bullet_n \boldsymbol{v} = \boldsymbol{\mathcal{T}} \times_n \boldsymbol{v}^\top \in \mathbb{R}^{d_1\times\cdots\times d_{n-1}\times d_{n+1}\times\cdots\times d_p}$. It is easy to

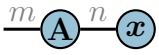
Figure 4.1: Tensor network representation of a vector $\boldsymbol{v} \in \mathbb{R}^d$, a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ and a tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$. The gray labels over the edges indicate the dimensions of the corresponding modes of the tensors (such labels will only be sporadically displayed when necessary to avoid confusion).

check that the n-mode product satisfies $(\boldsymbol{\mathcal{T}} \times_n \mathbf{A}) \times_n \mathbf{B} = \boldsymbol{\mathcal{T}} \times_n \mathbf{BA}$ where we assume compatible dimensions of the tensor $\boldsymbol{\mathcal{T}}$ and the matrices $\mathbf{A}$ and $\mathbf{B}$.

Tensor network diagrams allow one to represent complex operations on tensors in a graphical and intuitive way. A tensor network is simply a graph where nodes represent tensors, and edges represent contractions between tensor modes, i.e. a summation over an index shared by two tensors. In a tensor network, the arity of a vertex (i.e. the number of *legs* of a node) corresponds to the order of the tensor: a node with one leg represents a vector, a node with two legs represents a matrix, and a node with three legs represents a 3rd order tensor (see Figure 4.1). We will sometimes add indices to legs of a tensor network to refer to its components or sub-tensors. For example, the following tensor networks represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the ith row of $\mathbf{A}$ and the component $\mathbf{A}_{i,j}$ respectively:



Connecting two legs in a tensor network represents a contraction over the corresponding indices. Consider the following simple tensor network with two nodes:



The first node represents a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the second one a vector $\boldsymbol{x} \in \mathbb{R}^n$. Since this tensor network has one dangling leg (i.e. an edge which is not connected to any other node), it represents a vector. The edge between the second leg of $\mathbf{A}$ and the leg of $\boldsymbol{x}$ corresponds to a summation over the second mode of $\mathbf{A}$ and the first mode of $\boldsymbol{x}$,. Hence,

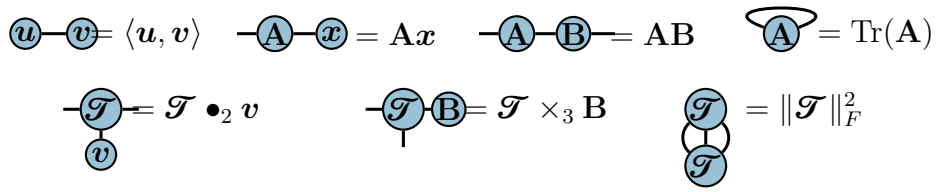


Figure 4.2: Tensor network representation of common operation on vectors, matrices and tensors.

the resulting tensor network represents the classical matrix-product, which can be seen by calculating the ith component of this tensor network:

$$i \text{—} \mathbf{A} \text{—} \boldsymbol{x} = \sum_j \mathbf{A}_{ij} \boldsymbol{x}_j = (\mathbf{A}\boldsymbol{x})_i$$

Other examples of tensor network representations of common operations on vectors, matrices and tensors can be found in Figure 4.2.

Given strictly positive integers $n_1, \cdots, n_k$ satisfying $\sum_i n_i = p$, we use the notation $(\mathcal{T})_{\langle\langle n_1, n_2, \cdots, n_k \rangle\rangle}$ to denote the kth order tensor obtained by reshaping $\mathcal{T} \in \mathbb{R}^{d_1 \times \cdots \times d_p}$ into a tensor* of size

$$(\prod_{i_1=1}^{n_1} d_{i_1}) \times (\prod_{i_2=1}^{n_2} d_{n_1+i_2}) \times \cdots \times (\prod_{i_k=1}^{n_k} d_{n_1+\cdots+n_{k-1}+i_k}).$$

For example, for a tensor $\mathcal{A}$ of size $2 \times 3 \times 4 \times 5 \times 6$, the 3rd order tensor $(\mathcal{A})_{\langle\langle 2,1,2 \rangle\rangle}$ is obtained by grouping the first two modes and the last two modes respectively, to obtain a tensor of size $6 \times 4 \times 30$. This reshaping operation is related to vectorization and matricization by the following relations: $(\mathcal{T})_{\langle\langle p \rangle\rangle} = \mathrm{vec}(\mathcal{T})$ and $(\mathcal{T})_{\langle\langle 1, p-1 \rangle\rangle} = \mathcal{T}_{(1)}$.

A rank R *tensor train (TT) decomposition* [Oseledets, 2011] of a tensor $\mathcal{T} \in \mathbb{R}^{d_1 \times \cdots \times d_p}$ consists in factorizing $\mathcal{T}$ into the product of p core tensors $\mathcal{G}_1 \in \mathbb{R}^{d_1 \times R}, \mathcal{G}_2 \in \mathbb{R}^{R \times d_2 \times R}, \cdots, \mathcal{G}_{p-1} \in \mathbb{R}^{R \times d_{p-1} \times R}, \mathcal{G}_p \in \mathbb{R}^{R \times d_p}$, and is defined† by

$$\mathcal{T}_{i_1, \cdots, i_p} = (\mathcal{G}_1)_{i_1,:} (\mathcal{G}_2)_{:,i_2,:} \cdots (\mathcal{G}_{p-1})_{:,i_{p-1},:} (\mathcal{G}_p)_{:,i_p} \tag{4.1}$$

*Note that the specific ordering used to perform matricization, vectorization and such a reshaping is not relevant as long as it is consistent across all operations.

†The classical definition of the TT-decomposition allows the rank R to be different for each mode, but this definition is sufficient for the purpose of this chapter.

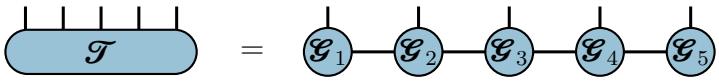


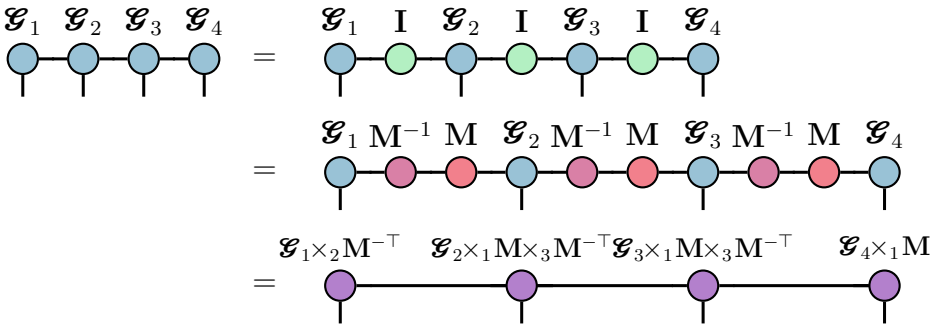
Figure 4.3: Tensor network representation of a tensor train decomposition.

for all indices $i_1 \in [d_1], \cdots, i_p \in [d_p]$ (here $(\mathcal{G}_1)_{i_1,:}$ is a row vector, $(\mathcal{G}_2)_{:,i_2,:}$ is an $R \times R$ matrix, etc.). We will use the notation $\mathcal{T} = [\![\mathcal{G}_1, \cdots, \mathcal{G}_p]\!]$ to denote such a decomposition. A tensor network representation of this decomposition is shown in Figure 4.3. The name of this decomposition comes from the fact that the tensor $\mathcal{T}$ is decomposed into a train of lower-order tensors. This decomposition is also known in the quantum physics community as *Matrix Product States* [Orús, 2014, Schollwöck, 2011], where this denomination comes from the fact that each entry of $\mathcal{T}$ is given by a product of matrices, see Eq. (4.1).

While the problem of finding the best approximation of TT-rank R of a given tensor is NP-hard [Hillar and Lim, 2013], a quasi-optimal SVD based compression algorithm (TT-SVD) has been proposed in [Oseledets, 2011]. It is worth mentioning that the TT decomposition is invariant under change of basis: for any invertible matrix $\mathbf{M}$ and any core tensors $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_p$, we have

$$[\![\mathcal{G}_1, \cdots, \mathcal{G}_p]\!] = [\![\mathcal{G}_1 \times_2 \mathbf{M}^{-\top}, \mathcal{G}_2 \times_1 \mathbf{M} \times_3 \mathbf{M}^{-\top}, \cdots, \mathcal{G}_{p-1} \times_1 \mathbf{M} \times_3 \mathbf{M}^{-\top}, \mathcal{G}_p \times_1 \mathbf{M}]\!].$$

This relation appears clearly using tensor network diagrams, e.g. with $p = 4$ we have[‡]:



[‡]Note that the colors of the nodes do not bear any meaning and are simply used as visual clues to help parse the diagrams.

4.2.2 Vector-valued Weighted Automata and Spectral Learning

Vector-valued weighted finite automata (vv-WFA) have been introduced in [Rabusseau et al., 2017] as a natural generalization of weighted automata from scalar-valued functions to vector-valued ones.

Definition 22. *A p-dimensional vv-WFA with n states is a tuple $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\Omega})$ where $\boldsymbol{\alpha} \in \mathbb{R}^n$ is the initial weights vector, $\boldsymbol{\Omega} \in \mathbb{R}^{p\times n}$ is the matrix of final weights, and $\mathbf{A}_\sigma \in \mathbb{R}^{n\times n}$ is the transition matrix for each symbol σ in a finite alphabet Σ. A vv-WFA A computes a function $f_A : \Sigma^* \to \mathbb{R}^p$ defined by*

$$f_A(x) = \boldsymbol{\Omega}(\mathbf{A}_{x_1}\mathbf{A}_{x_2}\cdots\mathbf{A}_{x_k})^\top\boldsymbol{\alpha}$$

for each word $x = x_1x_2\cdots x_k \in \Sigma^*$.

We call a vv-WFA *minimal* if its number of states is minimal, that is, any vv-WFA computing the same function as at least as many states as the minimal vv-WFA. Given a function $f : \Sigma^* \to \mathbb{R}^p$, we denote by $\mathrm{rank}(f)$ the number of states of a minimal vv-WFA computing f (which is set to ∞ if f cannot be computed by a vv-WFA).

The *spectral learning algorithm* is a consistent learning algorithm for weighted finite automata. It has been introduced concurrently in [Hsu et al., 2009] and [Bailly et al., 2009] (see [Balle et al., 2014a] for a comprehensive presentation of the algorithm). This algorithm relies on a fundamental object: *the Hankel matrix*. Given a function $f : \Sigma^* \to \mathbb{R}$, its Hankel matrix $\mathbf{H} \in \mathbb{R}^{\Sigma^*\times\Sigma^*}$ is the bi-infinite matrix defined by

$$\mathbf{H}_{u,v} = f(uv) \quad \text{for all } u, v \in \Sigma^*$$

where uv denotes the concatenation of the prefix u and the suffix v. The striking relation between the Hankel matrix and the rank of a function f has been well known in the formal language community [Fliess, 1974, Carlyle and Paz, 1971] and is at the heart of the spectral learning algorithm. This relation states that the rank of the Hankel matrix of a function f exactly coincides with the rank of f, i.e. the number of states of the smallest

WFA computing f. In particular, the rank of the Hankel matrix of f is finite if and only if f can be computed by a weighted automaton. An example of a function which cannot be computed by a WFA is the indicator function of the language $a^n b^n$ (on the alphabet $\Sigma = \{a, b\}$):

$$f(x) = \begin{cases} 1 & \text{if } x = a^n b^n \text{ for some integer } n \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

The spectral learning algorithm was naturally extended to vector-valued WFA in [Rabusseau et al., 2017], where the Hankel matrix is replaced by the *Hankel tensor* $\mathscr{H} \in \mathbb{R}^{\Sigma^* \times \Sigma^* \times p}$ of a vector-valued function $f : \Sigma^* \to \mathbb{R}^p$, which is defined by

$$\mathscr{H}_{u,v,:} = f(uv) \quad \text{for all } u, v \in \Sigma^*.$$

The relation between the rank of the Hankel matrix and the function f naturally carries over to the vector-valued case and is given in the following theorem.

Theorem 6 (Rabusseau et al. [2017]). *Let $f : \Sigma^* \to \mathbb{R}^d$ and let $\mathscr{H}$ be its Hankel tensor. Then* $\operatorname{rank}(f) = \operatorname{rank}(\mathscr{H}_{(1)})$.

The vv-WFA learning algorithm leverages the fact that the proof of this theorem is constructive: one can recover a vv-WFA computing f from any low rank factorization of $\mathscr{H}_{(1)}$. In practice, a finite sub-block $\mathscr{H}_{P,S} \in \mathbb{R}^{P \times S \times p}$ of the Hankel tensor is used to recover the vv-WFA, where $P, S \subset \Sigma^*$ are finite sets of prefixes and suffixes forming a *complete basis* for f, i.e. such that $\operatorname{rank}((\mathscr{H}_{P,S})_{(1)}) = \operatorname{rank}(\mathscr{H}_{(1)})$. Indeed, one can show that Theorem 6 still holds when replacing the Hankel tensor by such a sub-block $\mathscr{H}_{P,S}$. The spectral learning algorithm then consists of the following steps:

1. Choose a target rank n and a set of prefixes and suffixes $P, S \subset \Sigma^*$.
2. Estimate the following sub-block of the Hankel tensor from data:
   - $\mathscr{H}_{P,S} \in \mathbb{R}^{P \times S \times p}$ defined by $(\mathscr{H}_{P,S})_{u,v,:} = f(uv)$ for all $u \in P, v \in S$.

- $\mathbf{H}_P \in \mathbb{R}^{P \times p}$ defined by $(\mathbf{H}_P)_{u,:} = f(u)$ for all $u \in P$.
- $\mathbf{H}_S \in \mathbb{R}^{S \times p}$ defined by $(\mathbf{H}_S)_{v,:} = f(v)$ for all $v \in S$.
- $\boldsymbol{\mathcal{H}}^{\sigma}_{P,S} \in \mathbb{R}^{P \times S \times p}$ for each $\sigma \in \Sigma$ defined by $(\boldsymbol{\mathcal{H}}^{\sigma}_{P,S})_{u,v,:} = f(u\sigma v)$ for all $u \in P, v \in S$.

3. Obtain a (approximate) low rank factorization of the Hankel tensor (using e.g. truncated SVD)

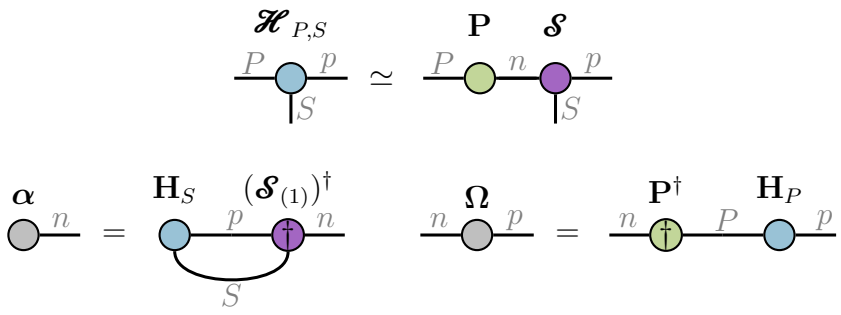
$$(\boldsymbol{\mathcal{H}}_{P,S})_{(1)} \simeq \mathbf{P}\boldsymbol{\mathcal{S}}_{(1)}$$

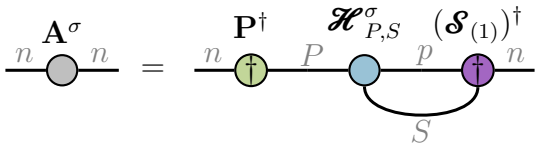
where $\mathbf{P} \in \mathbb{R}^{P \times n}$ and $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{n \times S \times p}$.

4. Compute the parameters of the learned vv-WFA using the relations

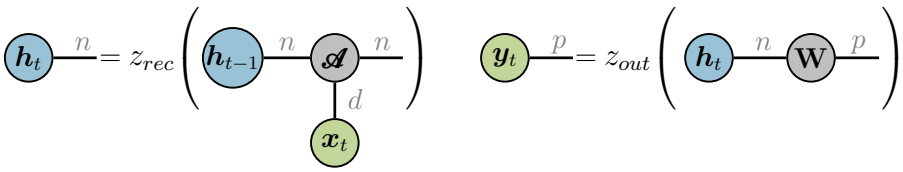
$$\begin{aligned}
\boldsymbol{\alpha}^\top &= \mathrm{vec}(\mathbf{H}_S)^\top (\boldsymbol{\mathcal{S}}_{(1)})^\dagger \\
\boldsymbol{\Omega} &= \mathbf{P}^{-1}\mathbf{H}_P \\
\mathbf{A}_\sigma &= \mathbf{P}^\dagger \boldsymbol{\mathcal{H}}^{\sigma}_{(1)} (\boldsymbol{\mathcal{S}}_{(1)})^\dagger \quad \text{for each } \sigma \in \Sigma.
\end{aligned}$$

This learning algorithm is *consistent*: in the limit of infinite training data (i.e. the Hankel sub-blocks are exactly estimated from data), this algorithm is guaranteed to return a WFA that computes the target function f if P and S form a complete basis. That is, the algorithm is consistent if the rank of the sub-block $(\boldsymbol{\mathcal{H}}_{P,S})_{(1)}$ is equal to the rank of the full Hankel tensor, i.e. $\mathrm{rank}((\boldsymbol{\mathcal{H}}_{P,S})_{(1)}) = \mathrm{rank}(\boldsymbol{\mathcal{H}}_{(1)})$. More details can be found in [Balle et al., 2014a] for WFA and in [Rabusseau et al., 2017] for vv-WFA. Using tensor network diagrams, steps 3) and 4) of the spectral learning algorithm can be represented as follows:





It is worth noting that the recurrent and output functions of a 2-RNN can be represented by the following simple tensor networks:



The difference between WFA and RNN can be summarized by the fact that the recurrent and output functions of a WFA are linear, whereas they are non-linear maps for RNN. In essence, one could say that RNN are non-linear extensions of WFA. In Section 4.4, we will formalize this intuition by proving the exact equivalence between the classes of functions that can be computed by WFA and second-order RNN with linear activation functions.

4.3 Weighted Automata and Tensor Networks

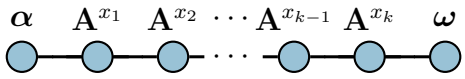
In this section, we present connections between weighted automata and tensor networks. In particular, we will show that the computation of a WFA on a sequence is intrinsically connected to the matrix product states model used in quantum physics and the tensor train decomposition. This connection will allow us to unravel a fundamental structure in the Hankel matrix of a function computed by a WFA: in addition to being low rank, we will show that the Hankel matrix can be decomposed into sub-blocks which are all matricizations of tensors with low tensor train rank. We will then leverage this structure to design an efficient spectral learning algorithm for WFA relying on efficient computations of pseudo-inverse of matrices given in the tensor train format.

4.3.1 Tensor Train Structure of the Hankel Matrix

For the sake of simplicity, we will consider scalar valued WFA in this section but all the results we present can be straightforwardly extended to vv-WFA. Let $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega})$ be a WFA with n states. Recall that A computes a function $f_A : \Sigma^* \to \mathbb{R}$ defined by

$$f_A(x_1x_2\cdots x_k) = \boldsymbol{\alpha}^\top \mathbf{A}_{x_1}\mathbf{A}_{x_2}\cdots\mathbf{A}_{x_k}\boldsymbol{\omega}$$

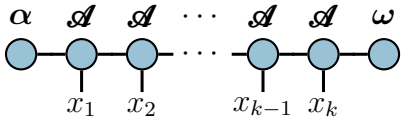
for any $k \geq 0$ and $x_1, x_2, \cdots, x_k \in \Sigma$. The computation of a WFA on a sequence can be represented by the following tensor network:

$f(x_1x_2\cdots x_k) =$ 

By stacking the transition matrices $\{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}$ into a third order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n\times\Sigma\times n}$ defined by

$$\boldsymbol{\mathcal{A}}_{:,\sigma,:} = \mathbf{A}_\sigma \quad \text{for all } \sigma \in \Sigma,$$

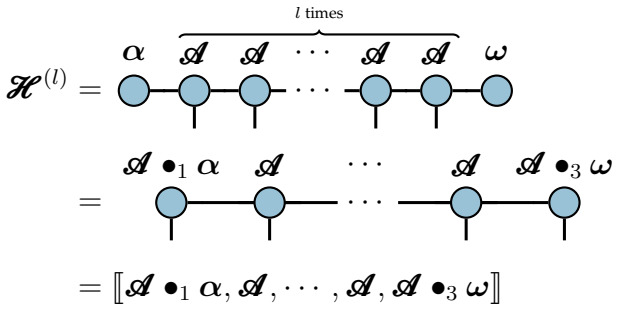
this computation can be rewritten into

$f(x_1x_2\cdots x_k) =$ 

This graphically shows the tight connection between WFA and the tensor train decomposition. More formally, for any integer l, let us define the lth order Hankel tensor $\boldsymbol{\mathcal{H}}^{(l)} \in \mathbb{R}^{\Sigma\times\Sigma\times\cdots\Sigma}$ by

$$\boldsymbol{\mathcal{H}}^{(l)}_{\sigma_1,\sigma_2,\cdots,\sigma_l} = f(\sigma_1\sigma_2\cdots\sigma_l) \text{ for all } \sigma_1,\cdots\sigma_l \in \Sigma. \tag{4.3}$$

Then, one can easily check that each such Hankel tensor admits the following rank n tensor train decomposition:



It follows that the Hankel matrix of a recognizable function can be decomposed into sub-blocks which are all matricization of Hankel tensors with low tensor train rank. To the best of our knowledge, this is a novel result that has not been noticed in the past. We conclude this section by formalizing this result in the following theorem.

Theorem 7. *Let $f : \Sigma^* \to \mathbb{R}$ be a function computed by a WFA with n states and let $\mathbf{H} \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$ be its Hankel matrix defined by $\mathbf{H}_{u,v} = f(uv)$ for all $u, v \in \Sigma^*$. Furthermore, for any integer l, let $\mathscr{H}^{(l)} \in \mathbb{R}^{\Sigma \times \Sigma \times \cdots \times \Sigma}$ be the lth order tensor defined by $\mathscr{H}^{(l)}_{\sigma_1, \sigma_2, \cdots, \sigma_l} = f(\sigma_1 \sigma_2 \cdots \sigma_l)$.*

Then, the Hankel matrix $\mathbf{H}$ can be decomposed into sub-blocks, each sub-block being the matricization of a tensor of tensor train rank at most n. More precisely, each of these sub-blocks is equal to $(\mathscr{H}^{(l)})_{\langle\langle k, l-k \rangle\rangle}$ for some values of l and k, and each Hankel tensor $\mathscr{H}^{(l)}$ has tensor train rank at most n.

Proof. For each $m, k \in \mathbb{N}$, let $\mathbf{H}^{(m,k)} \in \mathbb{R}^{\Sigma^m \times \Sigma^k}$ denote the sub-block of the Hankel matrix with prefixes Σ^m and suffixes Σ^k. It is easy to check that the Hankel matrix $\mathbf{H} \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$ can be partitioned into the sub-blocks $\mathbf{H}^{(m,k)}$ for $m, k \in \mathbb{N}$:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}^{(0,0)} & \mathbf{H}^{(0,1)} & \mathbf{H}^{(0,2)} & \mathbf{H}^{(0,3)} & \cdots \\ \mathbf{H}^{(1,0)} & \mathbf{H}^{(1,1)} & \mathbf{H}^{(1,2)} & \mathbf{H}^{(1,3)} & \cdots \\ \mathbf{H}^{(2,0)} & \mathbf{H}^{(2,1)} & \mathbf{H}^{(2,2)} & \mathbf{H}^{(2,3)} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

Now, by definition of the tensors $\mathscr{H}^{(l)}$, we have $\mathbf{H}^{(m,k)} = (\mathscr{H}^{(m+k)})_{\langle\langle m,k\rangle\rangle}$. Moreover, let $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega})$ be a WFA with n states computing f and let $\mathscr{A} \in \mathbb{R}^{n\times\Sigma\times n}$ be the 3rd order tensor defined by $\mathscr{A}_{:,\sigma,:} = \mathbf{A}_\sigma$ for each $\sigma \in \Sigma$. For any $m, k \in \mathbb{N}$ and any $\sigma_1, \cdots, \sigma_{m+k} \in \Sigma$, we have

$$\begin{aligned}
\mathbf{H}^{(m,k)}_{\sigma_1,\cdots,\sigma_{m+k}} &= ((\mathscr{H}^{(m+k)})_{\langle\langle m,k\rangle\rangle})_{\sigma_1\cdots\sigma_m,\sigma_{m+1}\cdots\sigma_{m+k}} \\
&= f(\sigma_1, \sigma_2, \cdots, \sigma_{m+k}) \\
&= \boldsymbol{\alpha}^\top \mathbf{A}_{\sigma_1}\mathbf{A}_{\sigma_2}\cdots\mathbf{A}_{\sigma_{m+k}}\boldsymbol{\omega} \\
&= \boldsymbol{\alpha}^\top \mathscr{A}_{:,\sigma_1,:}\mathscr{A}_{:,\sigma_2,:}\cdots\mathscr{A}_{:,\sigma_{m+k},:}\boldsymbol{\omega} \\
&= [\![\mathscr{A} \bullet_1 \boldsymbol{\alpha}, \mathscr{A}, \cdots, \mathscr{A}, \mathscr{A} \bullet_3 \boldsymbol{\omega}]\!]_{\sigma_1,\cdots,\sigma_{m+k}}.
\end{aligned}$$

It follows that

$$\mathbf{H}^{(m,k)} = (\mathscr{H}^{(m+k)})_{\langle\langle m,k\rangle\rangle} = ([\![\mathscr{A} \bullet_1 \boldsymbol{\alpha}, \overbrace{\mathscr{A}, \cdots, \mathscr{A}}^{m+k-2 \text{ times}}, \mathscr{A} \bullet_3 \boldsymbol{\omega}]\!])_{\langle\langle m,k\rangle\rangle}$$

and thus that each sub-block $\mathbf{H}^{(m,k)}$ is a matricization of a tensor of tensor train rank at most n. □

4.3.2 Spectral Learning in the Tensor Train Format

We now present how the tensor train structure of the Hankel matrix can be leveraged to significantly improve the computational complexity of steps 3 and 4 of the spectral learning algorithm described in Section 4.2.2. These two steps consist in first computing a low rank approximation of the Hankel sub-block

$$\mathbf{H}_{P,S} \simeq \mathbf{PS}$$

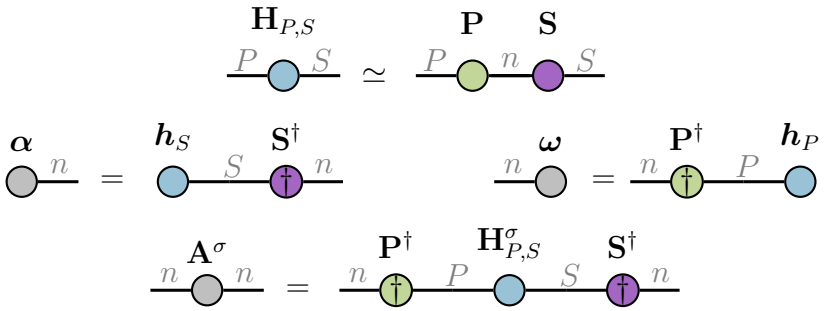
before estimating the parameters of the WFA using simple pseudo-inverse and matrix product computations

$$\boldsymbol{\alpha}^\top = \boldsymbol{h}_S^\top \mathbf{S}^\dagger,\ \boldsymbol{\omega} = \mathbf{P}^{-1}\boldsymbol{h}_P \text{ and } \mathbf{A}_\sigma = \mathbf{P}^\dagger \mathbf{H}_{P,S}^\sigma \mathbf{S}^\dagger \text{ for each } \sigma \in \Sigma$$

where the Hankel sub-blocks are defined by

$$(\boldsymbol{h}_P)_u = f(u),\ \ (\boldsymbol{h}_S)_v = f(v),\ \ (\mathbf{H}_{P,S})_{u,v} = f(uv) \text{ and } (\mathbf{H}_{P,S}^\sigma)_{u,v} = f(u\sigma v)$$

for all $\sigma \in \Sigma, u \in P, v \in S$. Note that we again focus on scalar-valued WFA for the sake of clarity (i.e. $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\omega})$) but the results we present can be straightforwardly extended to vector-valued WFA. Using tensor networks, these two steps are described as follows:



We now focus on the case where the basis of prefixes and suffixes are both equal to the set of all sequences of length l for some integer l, i.e. $P = S = \Sigma^l$. A first important observation is that, in this case, the Hankel sub-block $\mathbf{H}_{P,S}$ is a matricization of the $2l$-th order Hankel tensor $\boldsymbol{\mathscr{H}}^{(2l)} \in \mathbb{R}^{\Sigma\times\cdots\times\Sigma}$ defined in Eq. (4.3):

$$\mathbf{H}_{P,S} = (\boldsymbol{\mathscr{H}}^{(2l)})_{\langle\langle l,l\rangle\rangle}.$$

Indeed, for $u = u_1u_2\cdots u_l \in P = \Sigma^l$ and $v = v_1v_2\cdots v_l \in S = \Sigma^l$ we have

$$(\mathbf{H}_{P,S})_{u,v} = f(uv) = f(u_1u_2\cdots u_lv_1v_2\cdots v_l) = \boldsymbol{\mathscr{H}}^{(2l)}_{u_1,u_2,\cdots,u_l,v_1,v_2,\cdots,v_l}.$$

Using the same argument, one can easily show that $\boldsymbol{h}_P = \boldsymbol{h}_S = \mathrm{vec}(\boldsymbol{\mathcal{H}}^{(l)})$. Lastly, a similar observation can be done for the Hankel sub-blocks $\mathbf{H}^{\sigma}_{P,S}$ for each $\sigma \in \Sigma$: all of these sub-blocks are slices of the Hankel tensor $\boldsymbol{\mathcal{H}}^{(2l+1)}$. Indeed, for any $u = u_1 u_2 \cdots u_l \in P = \Sigma^l$ and $v = v_1 v_2 \cdots v_l \in S = \Sigma^l$ we have

$$(\mathbf{H}^{\sigma}_{P,S})_{u,v} = f(u\sigma v) = f(u_1 \cdots u_l \sigma v_1 \cdots v_l) = \boldsymbol{\mathcal{H}}^{(2l+1)}_{u_1,\cdots,u_l,\sigma,v_1,\cdots,v_l}$$

from which it follows that

$$\mathbf{H}^{\sigma}_{P,S} = \left((\boldsymbol{\mathcal{H}}^{(2l+1)})_{\langle\langle l,1,l \rangle\rangle} \right)_{:,\sigma,:} \text{ for all } \sigma \in \Sigma.$$

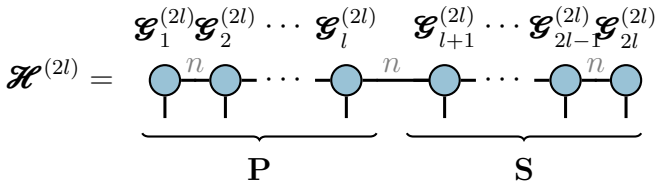
Thus, in the case where $P = S = \Sigma^l$, all the sub-blocks of the Hankel matrix one needs to estimate for the spectral learning algorithm are matricization of Hankel tensors of tensor train rank at most n (where n is the number of states of the target WFA). Let us assume for now that we have access to the true Hankel tensors $\boldsymbol{\mathcal{H}}^{(l)}$, $\boldsymbol{\mathcal{H}}^{(2l)}$ and $\boldsymbol{\mathcal{H}}^{(2l+1)}$ given in the tensor train format (how to estimate these Hankel tensors in the tensor train format from data will be discussed in Section 4.5.3):

$$\begin{aligned}
\boldsymbol{\mathcal{H}}^{(l)} &= [\![\boldsymbol{\mathcal{G}}^{(l)}_1, \cdots, \boldsymbol{\mathcal{G}}^{(l)}_l]\!] \\
\boldsymbol{\mathcal{H}}^{(2l)} &= [\![\boldsymbol{\mathcal{G}}^{(2l)}_1, \cdots, \boldsymbol{\mathcal{G}}^{(2l)}_{2l}]\!] \\
\boldsymbol{\mathcal{H}}^{(2l+1)} &= [\![\boldsymbol{\mathcal{G}}^{(2l+1)}_1, \cdots, \boldsymbol{\mathcal{G}}^{(2l+1)}_{2l+1}]\!]
\end{aligned}$$

where all tensor train decompositions are of rank n. We now show how the tensor train structure of the Hankel tensors can be leveraged to significantly improve the computational complexity of the spectral learning algorithm. Recall first that in the standard case, this complexity is in $\mathcal{O}\left(n|P||S| + n^2|P||\Sigma|\right)$ (where the first term corresponds to the truncated SVD of the Hankel matrix, and the second one to computing the transition matrices $\mathbf{A}_\sigma$), which is equal to $\mathcal{O}\left(n|\Sigma|^{2l} + n^2|\Sigma|^{l+1}\right)$ when $P = S = \Sigma^l$. In contrast, we will show

that if the Hankel tensors are given in the tensor train format, the complexity of the spectral learning algorithm can be reduced to $\mathcal{O}\left(n^3 l|\Sigma|\right)$.

First observe that the tensor train decomposition of the Hankel tensor $\mathscr{H}^{(2l)}$ already gives us the rank n factorization of the Hankel matrix $\mathbf{H}_{P,S} = (\mathscr{H}^{(2l)})_{\langle\langle l,l\rangle\rangle}$, which can easily be seen from the following tensor network:



More formally, this shows that $\mathbf{H}_{P,S} = \mathbf{PS}$ with $\mathbf{P} = (\llbracket \mathscr{G}_1^{(2l)}, \cdots, \mathscr{G}_l^{(2l)}, \mathbf{I} \rrbracket)_{\langle\langle l,1\rangle\rangle}$ and $\mathbf{S} = (\llbracket \mathbf{I}, \mathscr{G}_{l+1}^{(2l)}, \cdots, \mathscr{G}_{2l}^{(2l)} \rrbracket)_{\langle\langle 1,l\rangle\rangle}$. The remaining step of the spectral learning algorithm consists in computing the pseudo-inverse of $\mathbf{P}$ and $\mathbf{S}$ and performing various matrix products involving the Hankel sub-blocks $\boldsymbol{h}_P$, $\boldsymbol{h}_S$ and $\mathbf{H}_{P,S}^{\sigma}$ for each $\sigma \in \Sigma$. Observe that all the elements involved in these computations are tensors of tensor train rank at most n (or matricizations of such tensors). It turns out that all these operations can be performed efficiently in the tensor tensor train format: the pseudo-inverses of $\mathbf{P}$ and $\mathbf{S}$ in the tensor train format can be computed in time $\mathcal{O}\left(n^3 l|\Sigma|\right)$ and all the matrix products between $\mathbf{P}^\dagger$ and $\mathbf{S}^\dagger$ and the Hankel tensors can also be done in time $\mathcal{O}\left(n^3 l|\Sigma|\right)$. Describing these tensor train computations in details go beyond the scope of this chapter but these algorithms are well known in the tensor train and matrix product states communities. We refer the reader to [Oseledets, 2011] for efficient computations of matrix products in the tensor train format, and to [Gelß, 2017] and [Klus et al., 2018] for the computation of pseudo-inverse in the tensor train format.

We showed that in the case where $P = S = \Sigma^l$, the time complexity of the last two steps of the spectral learning algorithm can be reduced from an exponential dependency on l to a linear one. This is achieved by leveraging the tensor train structure of the Hankel sub-blocks. However, recall that the spectral learning algorithm is consistent (i.e. guaranteed

to return the target WFA from an infinite amount of training data) only if P and S form a complete basis, that is P and S are such that $\text{rank}((\boldsymbol{\mathscr{H}}_{P,S})_{(1)}) = \text{rank}(\mathbf{H})$. In the case where $P = S = \Sigma^l$, this condition is equivalent to $\text{rank}((\boldsymbol{\mathscr{H}}^{(2l)})_{\langle\langle l,l\rangle\rangle}) = \text{rank}(\mathbf{H})$. But it is not the case that for any function computed by a WFA, there exists an integer l such that $P = S = \Sigma^l$ form a complete basis. Indeed, consider for example the function f on the alphabet $\{a, b\}$ defined by $f(x) = 1$ if $x = aa$ and 0 otherwise. One can easily show that there exists a minimal WFA with 3 states computing f. However, it is easy to check that $\text{rank}((\boldsymbol{\mathscr{H}}^{(2l)})_{\langle\langle l,l\rangle\rangle})$ is equal to 1 for $l = 1$ and to 0 for any other value of l. This implies that not all functions can be consistently recovered from training data using the efficient spectral learning algorithm we propose.

Luckily, this caveat can be addressed using a simple workaround. For any function $f : \Sigma^* \to \mathbb{R}$, one can define a new alphabet $\tilde{\Sigma} = \Sigma \cup \{\lambda\}$ where λ is a new symbol not in Σ which will be treated as the empty string. One can then extend f to $\tilde{f} : \tilde{\Sigma}^* \to \mathbb{R}$ naturally by ignoring the new symbol λ, e.g. $\tilde{f}(\lambda ab\lambda c) = f(abc)$. Let $\mathbf{H} \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$, $\tilde{\mathbf{H}} \in \mathbb{R}^{\tilde{\Sigma}^* \times \tilde{\Sigma}^*}$, $\boldsymbol{\mathscr{H}}^{(2l)} \in \mathbb{R}^{\Sigma \times \cdots \times \Sigma}$ and $\tilde{\boldsymbol{\mathscr{H}}}^{(2l)} \in \mathbb{R}^{\tilde{\Sigma} \times \cdots \times \tilde{\Sigma}}$ be the Hankel matrix and tensors of f and $\tilde{f}$. Then, one can show that if f can be computed by a WFA with n states, there always exists an integer l such that $\text{rank}((\tilde{\boldsymbol{\mathscr{H}}}^{(2l)})_{\langle\langle l,l\rangle\rangle}) = \text{rank}(\mathbf{H}) = n$. Indeed, in contrast with the Hankel tensor $\boldsymbol{\mathscr{H}}^{(2l)}$ which only contains the values of f on sequences of length exactly $2l$, the Hankel tensors $\tilde{\boldsymbol{\mathscr{H}}}^{(2l)}$ contains the values of f on all sequences of length smaller than or equal to $2l$. One potential workaround would consist in estimating the Hankel sub-blocks of $\tilde{f}$ from data generated by f and perform steps 3 and 4 of the spectral learning to recover the parameters of a WFA computing $\tilde{f}$. The transition matrix associated with the new symbol λ can be discarded to obtain the parameters of a WFA estimating f (note that since the spectral learning algorithm is consistent, the transition matrix associated with the new symbol λ estimated from data is guaranteed to converge to the identity matrix as the training data increases).

In practice, to estimate a Hankel tensor of length L, one could append every sequence in the dataset that is of length L or smaller than L with λ until it reaches length L and

then perform the standard Hankel recovery routine. It is worth mentioning that we did not have to use this workaround for any of the experiments presented in Section 4.6. More importantly, one can show that if the parameters of a 2-RNN are drawn randomly then the workaround discussed above is not necessary (i.e., one can consistently recover a random 2-RNN from data using the learning algorithm we propose), which is shown in the following proposition.

Proposition 1. *Let $\mathscr{A} = \langle \boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}, \boldsymbol{\omega} \rangle$ be a 2-RNN with n states whose parameters are randomly drawn from a continuous distribution (w.r.t. the Lebesgue measure) and let $\mathbf{H} \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$ be its Hankel matrix. Then, with probability one, $\operatorname{rank}((\boldsymbol{\mathcal{H}}^{(2l)})_{\langle\langle l,l \rangle\rangle}) = \operatorname{rank}(\mathbf{H})$ for any l such that $|\Sigma|^l \geq n$ (where $\boldsymbol{\mathcal{H}}^{(2l)}$ is as defined in Eq. (4.3)).*

Proof. Let $\mathbf{F}_l \in \mathbb{R}^{\Sigma^l \times n}$ and $\mathbf{B}_l \in \mathbb{R}^{n \times \Sigma^l}$ be the forward and backward matrices of the random 2-RNN, that is the rows of $\mathbf{F}_l$ are $\boldsymbol{\alpha}^\top \boldsymbol{\mathcal{A}}_{:,u_1,:} \cdots \boldsymbol{\mathcal{A}}_{:,u_l,:}$ for $u_1 \cdots u_l \in \Sigma^l$ and the columns of $\mathbf{B}_l$ are $\boldsymbol{\mathcal{A}}_{:,v_1,:} \cdots \boldsymbol{\mathcal{A}}_{:,v_l,:} \boldsymbol{\omega}$ for $v_1 \cdots v_l \in \Sigma^l$. Let l be any integer such that $|\Sigma|^l \geq n$. We first show that both $\mathbf{F}_l$ and $\mathbf{B}_l$ are full rank with probability one. Observe that $\det(\mathbf{F}_l^\top \mathbf{F}_l)$ is a polynomial of the 2-RNN parameters $\boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}$. Since a polynomial is either zero or non-zero almost everywhere [Caron and Traynor, 2005], and since one can easily find a 2-RNN such that $\det(\mathbf{F}_l^\top \mathbf{F}_l) \neq 0$ (using the fact that $|\Sigma|^l \geq n$), it follows that $\det(\mathbf{F}_l^\top \mathbf{F}_l)$ is non-zero almost everywhere. Consequently, since the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\mathcal{A}}$ are drawn from a continuous distribution, $\det(\mathbf{F}_l^\top \mathbf{F}_l) \neq 0$ with probability one, i.e. $\mathbf{F}_l$ is of rank n with probability one. With a similar argument, one can show that $\mathbf{B}_l$ is of rank n with probability one.

To conclude, since $(\boldsymbol{\mathcal{H}}^{(2l)})_{\langle\langle l,l \rangle\rangle} = \mathbf{F}_l \mathbf{B}_l$, both $\mathbf{F}_l$ and $\mathbf{B}_l$ have rank n, and $|\Sigma|^l \geq n$, it follows that $(\boldsymbol{\mathcal{H}}^{(2l)})_{\langle\langle l,l \rangle\rangle}$ has rank n with probability one.

□

4.4 Weighted Automata and Second-Order Recurrent Neural Networks

In this section, we present an equivalence result between weighted automata and second-order RNN with linear activation functions (linear 2-RNN). This result rigorously formalizes the idea that WFA are linear RNN.

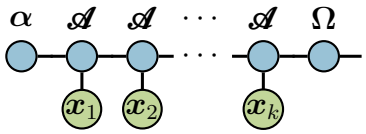
Recall that a 2-RNN $R = (\boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ maps any sequence of inputs $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k \in \mathbb{R}^d$ to a sequence of outputs $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_k \in \mathbb{R}^p$ defined for any $t = 1, \cdots, k$ by

$$\boldsymbol{y}_t = z_2(\boldsymbol{\Omega} \boldsymbol{h}_t) \text{ with } \boldsymbol{h}_t = z_1(\boldsymbol{\mathcal{A}} \bullet_1 \boldsymbol{h}_{t-1} \bullet_2 \boldsymbol{x}_t) \tag{4.4}$$

where $z_1 : \mathbb{R}^n \to \mathbb{R}^n$ and $z_2 : \mathbb{R}^p \to \mathbb{R}^p$ are activation functions. We think of a 2-RNN as computing a function $f_R : (\mathbb{R}^d)^* \to \mathbb{R}^p$ mapping each input sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k$ to the corresponding final output $\boldsymbol{y}_k$. While z_1 and z_2 are usually non-linear component-wise functions, we consider here the case where both z_1 and z_2 are the identity, and we refer to the resulting model as a *linear 2-RNN*.

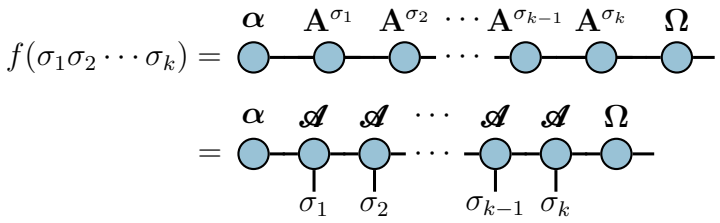
Observe that for a linear 2-RNN R, the function f_R is multilinear in the sense that, for any integer l, its restriction to the domain $(\mathbb{R}^d)^l$ is multilinear. Another useful observation is that linear 2-RNN are invariant under change of basis: for any invertible matrix $\mathbf{P}$, the linear 2-RNN $\tilde{M} = (\mathbf{P}^{-\top} \boldsymbol{h}_0, \boldsymbol{\mathcal{A}} \times_1 \mathbf{P} \times_3 \mathbf{P}^{-\top}, \mathbf{P}\boldsymbol{\Omega})$ is such that $f_{\tilde{M}} = f_M$.

One can easily show that the computation of the linear 2-RNN $R = (\boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ boils down to the following tensor network (see proof of Theorem 8):

$f_R(\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_k) =$ 

This computation is very similar, not to say equivalent, to the computation of a WFA $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma \in \Sigma}, \boldsymbol{\Omega})$. Indeed, as we showed in the previous section, by stacking the transition matrices $\{\mathbf{A}_\sigma\}_{\sigma \in \Sigma}$ into a third order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n \times \Sigma \times n}$ the computation of the

WFA A can be written as



Thus, if we restrict the input vectors of a linear 2-RNN to be one-hot encoding (i.e. vectors from the canonical basis), the two models are strictly equivalent.

These observations unravel a fundamental connection between vv-WFA and linear 2-RNN: vv-WFA and linear 2-RNN are expressively equivalent for representing functions defined over sequences of discrete symbols. Moreover, both models have the same capacity in the sense that there is a direct correspondence between the hidden units of a linear 2-RNN and the states of a vv-WFA computing the same function. More formally, we have the following theorem.

Theorem 8. *Any function that can be computed by a vv-WFA with n states can be computed by a linear 2-RNN with n hidden units. Conversely, any function that can be computed by a linear 2-RNN with n hidden units on sequences of one-hot vectors (i.e. canonical basis vectors) can be computed by a WFA with n states.*

More precisely, the WFA $A = (\boldsymbol{\alpha}, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}, \boldsymbol{\Omega})$ with n states and the linear 2-RNN $M = (\boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ with n hidden units, where $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n\times\Sigma\times n}$ is defined by $\boldsymbol{\mathcal{A}}_{:,\sigma,:} = \mathbf{A}_\sigma$ for all $\sigma \in \Sigma$, are such that $f_A(\sigma_1\sigma_2\cdots\sigma_k) = f_M(\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_k)$ for all sequences of input symbols $\sigma_1, \cdots, \sigma_k \in \Sigma$, where for each $i \in [k]$ the input vector $\boldsymbol{x}_i \in \mathbb{R}^\Sigma$ is the one-hot encoding of the symbol σ_i.

Proof. We first show by induction on k that, for any sequence $\sigma_1\cdots\sigma_k \in \Sigma^*$, the hidden state $\boldsymbol{h}_k$ computed by M (see Eq. (4.4)) on the corresponding one-hot encoded sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k \in \mathbb{R}^d$ satisfies $\boldsymbol{h}_k = (\mathbf{A}_{\sigma_1}\cdots\mathbf{A}_{\sigma_k})^\top\boldsymbol{\alpha}$. The case $k = 0$ is immediate. Suppose the result true for sequences of length up to k. One can check easily check that $\boldsymbol{\mathcal{A}} \bullet_2 \boldsymbol{x}_i = \mathbf{A}_{\sigma_i}$

for any index i. Using the induction hypothesis it then follows that

$$\begin{aligned}\boldsymbol{h}_{k+1} &= \boldsymbol{\mathscr{A}} \bullet_1 \boldsymbol{h}_k \bullet_2 \boldsymbol{x}_{k+1} = \mathbf{A}_{\sigma_{k+1}} \bullet_1 \boldsymbol{h}_k = (\mathbf{A}_{\sigma_{k+1}})^\top \boldsymbol{h}_k \\ &= (\mathbf{A}_{\sigma_{k+1}})^\top (\mathbf{A}_{\sigma_1} \cdots \mathbf{A}_{\sigma_k})^\top \boldsymbol{\alpha} = (\mathbf{A}_{\sigma_1} \cdots \mathbf{A}_{\sigma_{k+1}})^\top \boldsymbol{\alpha}.\end{aligned}$$

To conclude, we thus have

$$f_M(\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_k) = \boldsymbol{\Omega} \boldsymbol{h}_k = \boldsymbol{\Omega} (\mathbf{A}_{\sigma_1} \cdots \mathbf{A}_{\sigma_k})^\top \boldsymbol{\alpha} = f_A(\sigma_1 \sigma_2 \cdots \sigma_k).$$

□

This result first implies that linear 2-RNN defined over sequences of discrete symbols (using one-hot encoding) *can be provably learned using the spectral learning algorithm for WFA/vv-WFA*; indeed, these algorithms have been proved to compute consistent estimators. Let us stress again that, contrary to the case of feed-forward architectures, learning recurrent networks with linear activation functions is not a trivial task. Furthermore, Theorem 8 reveals that linear 2-RNN are a natural generalization of classical weighted automata to functions defined over sequences of continuous vectors (instead of discrete symbols). Therefore, we will also refer to the linear 2-RNN as continuous weighted finite automata (CWFAs) in the remaining of the thesis. The above observations spontaneously raise the question of whether the spectral learning algorithms for WFA and vv-WFA can be extended to the general setting of linear 2-RNN, namely, CWFA; we show that the answer is in the positive in the next section.

4.5 Spectral Learning of Continuous Weighted Automata

In this section, we extend the learning algorithm for vv-WFA to linear 2-RNN, thus at the same time addressing the limitation of the spectral learning algorithm to discrete inputs and providing the first consistent learning algorithm for linear second-order RNN.

4.5.1 Recovering 2-RNN from Hankel Tensors

We first present an identifiability result showing how one can recover a linear 2-RNN computing a function $f : (\mathbb{R}^d)^* \to \mathbb{R}^p$ from observable tensors extracted from some Hankel tensor associated with f. Intuitively, we obtain this result by reducing the problem to the one of learning a vv-WFA. This is done by considering the restriction of f to canonical basis vectors; loosely speaking, since the domain of this restricted function is isomorphic to $[d]^*$, this allows us to fall back onto the setting of sequences of discrete symbols.

It is not straightforward how the notion of Hankel matrix can be extended to a function $f : (\mathbb{R}^d)^* \to \mathbb{R}^p$ taking sequences of *continuous* vectors as input. One natural way to proceed is to consider how f acts on sequences of vectors from the canonical basis. Given a function $f : (\mathbb{R}^d)^* \to \mathbb{R}^p$, we define its Hankel tensor $\mathscr{H}_f \in \mathbb{R}^{[d]^* \times [d]^* \times p}$ by

$$(\mathscr{H}_f)_{i_1 \cdots i_s, j_1 \cdots j_t, :} = f(\boldsymbol{e}_{i_1}, \cdots, \boldsymbol{e}_{i_s}, \boldsymbol{e}_{j_1}, \cdots, \boldsymbol{e}_{j_t}),$$

for all $i_1, \cdots, i_s, j_1, \cdots, j_t \in [d]$, which is infinite in two of its modes. It is easy to see that $\mathscr{H}_f$ is also the Hankel tensor associated with the function $\tilde{f} : [d]^* \to \mathbb{R}^p$ mapping any sequence $i_1 i_2 \cdots i_k \in [d]^*$ to $f(\boldsymbol{e}_{i_1}, \cdots, \boldsymbol{e}_{i_k})$. Moreover, in the special case where f can be computed by a linear 2-RNN, one can use the multilinearity of f to show that

$$f(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k) = \sum_{i_1, \cdots, i_k = 1}^{d} (\boldsymbol{x}_1)_{i_1} \cdots (\boldsymbol{x}_l)_{i_k} \tilde{f}(i_1 \cdots i_k).$$

This gives us some intuition on how one could learn f by learning a vv-WFA computing $\tilde{f}$ using the spectral learning algorithm. That is, assuming access to the sub-blocks of the Hankel tensor $\mathscr{H}$ for a complete basis of prefixes and suffixes $P, S \subseteq [d]^*$, the spectral learning algorithm can be used to recover a vv-WFA computing $\tilde{f}$ and consequently a linear 2-RNN computing f using Theorem 8.

We now state the main result of this section, showing that a (minimal) linear 2-RNN computing a function $f : (\mathbb{R}^d)^* \to \mathbb{R}$ can be exactly recovered from sub-blocks of the

Hankel tensor $\mathscr{H}_f$. For the sake of clarity, we present the learning algorithm for the particular case where there exists an L such that the prefix and suffix sets consisting of all sequences of length L, that is $P = S = [d]^L$, forms a complete basis for $\tilde{f}$ (i.e. the sub-block $\mathscr{H}_{P,S} \in \mathbb{R}^{[d]^L \times [d]^L \times p}$ of the Hankel tensor $\mathscr{H}_f$ is such that $\mathrm{rank}((\mathscr{H}_{P,S})_{(1)}) = \mathrm{rank}((\mathscr{H}_f)_{(1)})$). As discussed in Section 4.3.2, such an integer L does not always exist even when the underlying function f can be computed by a linear 2-RNN. However, the workaround described at the end of Section 4.3.2 can be used here as well to extend this theorem to the case of any function f that can be computed by a linear 2-RNN.

The following theorem can be seen as a reformulation of the classical spectral learning theorem using the low rank Hankel tensors $\mathscr{H}^{(l)}$ introduced in Section 4.3.1. In the case of a continuous function $f : (\mathbb{R}^d)^* \to \mathbb{R}$, for any integer l, the finite tensor $\mathscr{H}_f^{(l)} \in \mathbb{R}^{d \times \cdots \times d \times p}$ of order $l+1$ is defined by

$$(\mathscr{H}_f^{(l)})_{i_1,\cdots,i_l,:} = f(\boldsymbol{e}_{i_1}, \cdots, \boldsymbol{e}_{i_l}) \quad \text{for all } i_1, \cdots, i_l \in [d].$$

Observe that for any integer l, the tensor $\mathscr{H}_f^{(l)}$ can be obtained by reshaping a finite sub-block of the Hankel tensor $\mathscr{H}_f$.

Theorem 9. *Let $f : (\mathbb{R}^d)^* \to \mathbb{R}^p$ be a function computed by a minimal linear 2-RNN with n hidden units and let L be an integer such that*

$$\mathrm{rank}((\mathscr{H}_f^{(2L)})_{\langle\langle L, L+1 \rangle\rangle}) = n.$$

Then, for any $\mathbf{P} \in \mathbb{R}^{d^L \times n}$ and $\mathbf{S} \in \mathbb{R}^{n \times d^L p}$ such that

$$(\mathscr{H}_f^{(2L)})_{\langle\langle L, L+1 \rangle\rangle} = \mathbf{P}\mathbf{S},$$

the linear 2-RNN $R = (\boldsymbol{\alpha}, \mathscr{A}, \boldsymbol{\Omega})$ defined by

$$\boldsymbol{\alpha} = (\mathbf{S}^\dagger)^\top (\mathscr{H}_f^{(L)})_{\langle\langle L+1 \rangle\rangle}, \quad \boldsymbol{\Omega}^\top = \mathbf{P}^\dagger (\mathscr{H}_f^{(L)})_{\langle\langle L, 1 \rangle\rangle}$$

$$\boldsymbol{\mathcal{A}} = ((\boldsymbol{\mathcal{H}}_f^{(2L+1)})_{\langle\langle L,1,L+1\rangle\rangle}) \times_1 \mathbf{P}^\dagger \times_3 (\mathbf{S}^\dagger)^\top$$

is a minimal linear 2-RNN computing f.

Proof. Let $\mathbf{P} \in \mathbb{R}^{d^L \times n}$ and $\mathbf{S} \in \mathbb{R}^{n \times d^L p}$ be such that $(\boldsymbol{\mathcal{H}}_f^{(2L)})_{\langle\langle L,L+1\rangle\rangle} = \mathbf{PS}$ and let $R^\star = (\boldsymbol{\alpha}, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ be a minimal linear 2-RNN computing f. Define the tensors

$$\boldsymbol{\mathcal{P}}^\star = [\![\boldsymbol{\mathcal{A}}^\star \bullet_1 \boldsymbol{\alpha}^\star, \underbrace{\boldsymbol{\mathcal{A}}^\star, \cdots, \boldsymbol{\mathcal{A}}^\star}_{L-1 \text{ times}}, \mathbf{I}_n]\!] \in \mathbb{R}^{d \times \cdots \times d \times n}$$

and

$$\boldsymbol{\mathcal{S}}^\star = [\![\mathbf{I}_n, \underbrace{\boldsymbol{\mathcal{A}}^\star, \cdots, \boldsymbol{\mathcal{A}}^\star}_{L \text{ times}}, \boldsymbol{\Omega}^\star]\!] \in \mathbb{R}^{n \times d \times \cdots \times d \times p}$$

of order $L+1$ and $L+2$ respectively, and let $\mathbf{P}^\star = (\boldsymbol{\mathcal{P}}^\star)_{\langle\langle l,1\rangle\rangle} \in \mathbb{R}^{d^L \times n}$ and $\mathbf{S}^\star = (\boldsymbol{\mathcal{S}}^\star)_{\langle\langle 1,L+1\rangle\rangle} \in \mathbb{R}^{n \times d^L p}$. Using the identity $\boldsymbol{\mathcal{H}}_f^{(l)} = [\![\boldsymbol{\mathcal{A}} \bullet_1 \boldsymbol{\alpha}, \underbrace{\boldsymbol{\mathcal{A}}, \cdots, \boldsymbol{\mathcal{A}}}_{l-1 \text{ times}}, \boldsymbol{\Omega}^\top]\!]$ for any l, one can easily check the following identities (see also Section 4.3.1):

$$(\boldsymbol{\mathcal{H}}_f^{(2L)})_{\langle\langle L,L+1\rangle\rangle} = \mathbf{P}^\star \mathbf{S}^\star, \quad (\boldsymbol{\mathcal{H}}_f^{(2L+1)})_{\langle\langle L,1,L+1\rangle\rangle} = \boldsymbol{\mathcal{A}}^\star \times_1 \mathbf{P}^\star \times_3 (\mathbf{S}^\star)^\top,$$
$$(\boldsymbol{\mathcal{H}}_f^{(L)})_{\langle\langle L,1\rangle\rangle} = \mathbf{P}^\star (\boldsymbol{\Omega}^\star)^\top, \qquad (\boldsymbol{\mathcal{H}}_f^{(L)})_{\langle\langle L+1\rangle\rangle} = (\mathbf{S}^\star)^\top \boldsymbol{\alpha}.$$

Let $\mathbf{M} = \mathbf{P}^\dagger \mathbf{P}^\star$. We will show that $\boldsymbol{\alpha} = \mathbf{M}^{-\top} \boldsymbol{\alpha}^\star$, $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{A}}^\star \times_1 \mathbf{M} \times_3 \mathbf{M}^{-\top}$ and $\boldsymbol{\Omega} = \mathbf{M}\boldsymbol{\Omega}^\star$, which will entail the results since linear 2-RNN are invariant under change of basis. First observe that $\mathbf{M}^{-1} = \mathbf{S}^\star \mathbf{S}^\dagger$. Indeed, we have $\mathbf{P}^\dagger \mathbf{P}^\star \mathbf{S}^\star \mathbf{S}^\dagger = \mathbf{P}^\dagger (\boldsymbol{\mathcal{H}}_f^{(2L)})_{\langle\langle L,L+1\rangle\rangle} \mathbf{S}^\dagger = \mathbf{P}^\dagger \mathbf{P} \mathbf{S} \mathbf{S}^\dagger = \mathbf{I}$ where we used the fact that $\mathbf{P}$ (resp. $\mathbf{S}$) is of full column rank (resp. row rank) for the last equality.

The following derivations then follow from basic tensor algebra:

$$\boldsymbol{\alpha} = (\mathbf{S}^\dagger)^\top (\boldsymbol{\mathcal{H}}_f^{(L)})_{\langle\langle L+1 \rangle\rangle} = (\mathbf{S}^\dagger)^\top (\mathbf{S}^\star)^\top \boldsymbol{\alpha} = (\mathbf{S}^\star \mathbf{S}^\dagger)^\top = \mathbf{M}^{-\top} \boldsymbol{\alpha}^\star,$$

$$\begin{aligned}
\boldsymbol{\mathcal{A}} &= ((\boldsymbol{\mathcal{H}}_f^{(2L+1)})_{\langle\langle L,1,L+1 \rangle\rangle}) \times_1 \mathbf{P}^\dagger \times_3 (\mathbf{S}^\dagger)^\top \\
&= (\boldsymbol{\mathcal{A}}^\star \times_1 \mathbf{P}^\star \times_3 (\mathbf{S}^\star)^\top) \times_1 \mathbf{P}^\dagger \times_3 (\mathbf{S}^\dagger)^\top \\
&= \boldsymbol{\mathcal{A}}^\star \times_1 \mathbf{P}^\dagger \mathbf{P}^\star \times_3 (\mathbf{S}^\star \mathbf{S}^\dagger)^\top = \boldsymbol{\mathcal{A}}^\star \times_1 \mathbf{M} \times_3 \mathbf{M}^{-\top},
\end{aligned}$$

$$\boldsymbol{\Omega}^\top = \mathbf{P}^\dagger (\boldsymbol{\mathcal{H}}_f^{(L)})_{\langle\langle L,1 \rangle\rangle} = \mathbf{P}^\dagger \mathbf{P}^\star (\boldsymbol{\Omega}^\star)^\top = \mathbf{M} \boldsymbol{\Omega}^\star,$$

which concludes the proof. □

Observe that such an integer L exists under the assumption that $\mathscr{P} = \mathcal{S} = [d]^L$ forms a complete basis for $\tilde{f}$. It is also worth mentioning that a necessary condition for $\operatorname{rank}((\boldsymbol{\mathcal{H}}_f^{(2L)})_{\langle\langle L,L+1 \rangle\rangle}) = n$ is that $d^L \geq n$, i.e. L must be of the order $\log_d(n)$.

4.5.2 Hankel Tensors Recovery from Linear Measurements

We showed in the previous section that, given the Hankel tensors $\boldsymbol{\mathcal{H}}_f^{(L)}$, $\boldsymbol{\mathcal{H}}_f^{(2L)}$ and $\boldsymbol{\mathcal{H}}_f^{(2L+1)}$, one can recover a linear 2-RNN computing f if it exists. This first implies that the class of functions that can be computed by linear 2-RNN is learnable in Angluin's exact learning model [Angluin, 1988] where one has access to an oracle that can answer membership queries (e.g. *what is the value computed by the target f on $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k)$?*) and equivalence queries (e.g. *is the current hypothesis h equal to the target f?*). While this fundamental result is of significant theoretical interest, assuming access to such an oracle is unrealistic. In this section, we show that a stronger learnability result can be obtained in a more realistic setting, where we only assume access to randomly generated input/output examples $((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)}) \in (\mathbb{R}^d)^* \times \mathbb{R}^p$ where $\boldsymbol{y}^{(i)} = f(\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)})$.

The key observation is that such an example $((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)})$ can be seen as a *linear measurement* of the Hankel tensor $\boldsymbol{\mathcal{H}}^{(l)}$. Indeed, let f be a function computed by a linear 2-RNN. Using the multilinearity of f we have

$$\begin{aligned}
f(\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_l) &= f\left(\sum_{i_1} (\boldsymbol{x}_1)_{i_1} \boldsymbol{e}_{i_1}, \sum_{i_2} (\boldsymbol{x}_2)_{i_2} \boldsymbol{e}_{i_2}, \cdots, \sum_{i_l} (\boldsymbol{x}_l)_{i_l} \boldsymbol{e}_{i_l}\right) \\
&= \sum_{i_1, \cdots, i_l} (\boldsymbol{x}_1)_{i_1} \cdots (\boldsymbol{x}_l)_{i_l} f(\boldsymbol{e}_{i_1}, \cdots, \boldsymbol{e}_{i_l}) \\
&= \sum_{i_1, \cdots, i_l} (\boldsymbol{x}_1)_{i_1} \cdots (\boldsymbol{x}_l)_{i_l} \boldsymbol{\mathcal{H}}^{(l)}_{i_1, \cdots, i_l} \\
&= \boldsymbol{\mathcal{H}}^{(l)}_f \bullet_1 \boldsymbol{x}_1 \bullet_2 \cdots \bullet_l \boldsymbol{x}_l \\
&= (\boldsymbol{\mathcal{H}}^{(l)})^\top_{\langle\langle l,1 \rangle\rangle} (\boldsymbol{x}_1 \otimes \cdots \otimes \boldsymbol{x}_l)
\end{aligned}$$

where $(\boldsymbol{e}_1, \cdots, \boldsymbol{e}_l)$ denotes the canonical basis of $\mathbb{R}^l$. It follows that each input/output example $((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)})$ constitutes a linear measurement of $\boldsymbol{\mathcal{H}}^{(l)}$:

$$\boldsymbol{y}^{(i)} = (\boldsymbol{\mathcal{H}}^{(l)})^\top_{\langle\langle l,1 \rangle\rangle} (\boldsymbol{x}_1^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)}) = (\boldsymbol{\mathcal{H}}^{(l)})^\top_{\langle\langle l,1 \rangle\rangle} \boldsymbol{x}^{(i)}$$

where $\boldsymbol{x}^{(i)} := \boldsymbol{x}_1^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)} \in \mathbb{R}^{d^l}$. Hence, by regrouping N output examples $\boldsymbol{y}^{(i)}$ into the matrix $\mathbf{Y} \in \mathbb{R}^{N \times p}$ and the corresponding input vectors $\boldsymbol{x}^{(i)}$ into the matrix $\mathbf{X} \in \mathbb{R}^{N \times d^l}$, one can recover $\boldsymbol{\mathcal{H}}^{(l)}$ by solving the linear system $\mathbf{Y} = \mathbf{X}(\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1 \rangle\rangle}$, which has a unique solution whenever $\mathbf{X}$ is of full column rank. This simple estimation technique for the Hankel tensors allows us to design the first consistent learning algorithm for linear 2-RNN, which is summarized in Algorithm 5 (with the "Least-Squares" recovery method). More efficient recovery methods for the Hankel tensors will be discussed in the next section. The following theorem shows that this learning algorithm is consistent. Its proof relies on the fact that $\mathbf{X}$ will be of full column rank whenever $N \geq d^l$ and the components of each $\boldsymbol{x}_j^{(i)}$ for $j \in [l], i \in [N]$ are drawn independently from a continuous distribution over $\mathbb{R}^d$ (w.r.t. the Lebesgue measure).

Theorem 10. *Let $(\boldsymbol{h}_0, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ be a minimal linear 2-RNN with n hidden units computing a function $f : (\mathbb{R}^d)^* \to \mathbb{R}^p$, and let L be an integersuch that $\operatorname{rank}((\boldsymbol{\mathcal{H}}_f^{(2L)})_{\langle\langle L, L+1\rangle\rangle}) = n$.*

Suppose we have access to 3 datasets

$$D_l = \{((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)})\}_{i=1}^{N_l} \subset (\mathbb{R}^d)^l \times \mathbb{R}^p \text{ for } l \in \{L, 2L, 2L+1\}$$

where the entries of each $\boldsymbol{x}_j^{(i)}$ are drawn independently from the standard normal distribution and where each $\boldsymbol{y}^{(i)} = f(\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)})$.

Then, if $N_l \geq d^l$ for $l = L,\ 2L,\ 2L+1$, the linear 2-RNN M returned by Algorithm 5 with the least-squares method satisfies $f_M = f$ with probability one.

Proof. We just need to show for each $l \in \{L, 2L, 2L+1\}$ that, under the hypothesis of the Theorem, the Hankel tensors $\hat{\boldsymbol{\mathcal{H}}}^{(l)}$ computed in line 4 of Algorithm 5 are equal to the true Hankel tensors $\boldsymbol{\mathcal{H}}^{(l)}$ with probability one. Recall that these tensors are computed by solving the least-squares problem

$$\hat{\boldsymbol{\mathcal{H}}}^{(l)} = \underset{T \in \mathbb{R}^{d \times \cdots \times d \times p}}{\arg\min} \|\mathbf{X}(\boldsymbol{\mathcal{T}})_{\langle\langle l,1\rangle\rangle} - \mathbf{Y}\|_F^2$$

where $\mathbf{X} \in \mathbb{R}^{N_l \times d_l}$ is the matrix with rows $\boldsymbol{x}_1^{(i)} \otimes \boldsymbol{x}_2^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)}$ for each $i \in [N_l]$. Since $\mathbf{X}(\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1\rangle\rangle} = \mathbf{Y}$ and the solution of the least-squares problem is unique as soon as $\mathbf{X}$ is of full column rank, we just need to show that this is the case with probability one when the entries of the vectors $\boldsymbol{x}_j^{(i)}$ are drawn at random from a standard normal distribution. The result will then directly follow by applying Theorem 9.

We will show that the set

$$\mathcal{S} = \{(\boldsymbol{x}_1^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}) \mid\ i \in [N_l],\ dim(span(\{\boldsymbol{x}_1^{(i)} \otimes \boldsymbol{x}_2^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)}\})) < d^l\}$$

has Lebesgue measure 0 in $((\mathbb{R}^d)^l)^{N_l} \simeq \mathbb{R}^{dlN_l}$ as soon as $N_l \geq d^l$, which will imply that it has probability 0 under any continuous probability, hence the result. For any $S = \{(\boldsymbol{x}_1^{(i)}, \cdots, \boldsymbol{x}_l^{(i)})\}_{i=1}^{N_l}$, we denote by $\mathbf{X}_S \in \mathbb{R}^{N_l \times d^l}$ the matrix with rows $\boldsymbol{x}_1^{(i)} \otimes \boldsymbol{x}_2^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)}$.

One can easily check that $S \in \mathcal{S}$ if and only if $\mathbf{X}_S$ is of rank strictly less than d^l, which is equivalent to the determinant of $\mathbf{X}_S^\top \mathbf{X}_S$ being equal to 0. Since this determinant is a polynomial in the entries of the vectors $\boldsymbol{x}_j^{(i)}$, $\mathcal{S}$ is an algebraic subvariety of $\mathbb{R}^{dlN_l}$. It is then easy to check that the polynomial $det(\mathbf{X}_S^\top \mathbf{X}_S)$ is not uniformly 0 when $N_l \geq d^l$. Indeed, it suffices to choose the vectors $\boldsymbol{x}_j^{(i)}$ such that the family $(\boldsymbol{x}_1^{(i)} \otimes \boldsymbol{x}_2^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)})_{n=1}^{N_l}$ spans the whole space $\mathbb{R}^{d^l}$ (which is possible since we can choose arbitrarily any of the $N_l \geq d^l$ elements of this family), hence the result. In conclusion, $\mathcal{S}$ is a proper algebraic subvariety of $\mathbb{R}^{dlN_l}$ and hence has Lebesgue measure zero [Federer, 2014, Section 2.6.5]. □

A few remarks on this theorem are in order. The first observation is that the 3 datasets D_L, D_{2L} and D_{2L+1} do not need to be drawn independently from one another (e.g. the sequences in D_L can be prefixes of the sequences in D_{2L} but it is not necessary). In particular, the result still holds when the datasets D_L, D_{2L} and D_{2L+1} are constructed from a unique dataset

$$S = \{((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_T^{(i)}), (\boldsymbol{y}_1^{(i)}, \boldsymbol{y}_2^{(i)}, \cdots, \boldsymbol{y}_T^{(i)}))\}_{i=1}^N$$

of input/output sequences with $T \geq 2L+1$, where $\boldsymbol{y}_t^{(i)} = f(\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_t^{(i)})$ for any $t \in [T]$. Observe that having access to such input/output training sequences is not an unrealistic assumption: for example when training RNN for language modeling the output $\boldsymbol{y}_t$ is the conditional probability vector of the next symbol. Lastly, when the outputs $\boldsymbol{y}^{(i)}$ are noisy, one can solve the least-squares problem $\|\mathbf{Y} - \mathbf{X}(\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1 \rangle\rangle}\|_F^2$ to approximate the Hankel tensors; we will empirically evaluate this approach in Section 4.6 and we defer its theoretical analysis in the noisy setting to future work.

Algorithm 5 `2RNN-SL`: Spectral Learning of linear 2-RNN

Input: Three training datasets D_L, D_{2L}, D_{2L+1} with input sequences of length L, $2L$ and $2L+1$ respectively, a `recovery_method`, rank R, noise tolerance parameter ε (for Nuclear Norm) and learning rate γ (for IHT/TIHT/SGD).

for $l \in \{L, 2L, 2L+1\}$ **do**
  Use $D_l = \{((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)})\}_{i=1}^{N_l} \subset (\mathbb{R}^d)^l \times \mathbb{R}^p$ to build $\mathbf{X} \in \mathbb{R}^{N_l \times d^l}$ with rows $\boldsymbol{x}_1^{(i)} \otimes \boldsymbol{x}_2^{(i)} \otimes \cdots \otimes \boldsymbol{x}_l^{(i)}$ for $i \in [N_l]$ and $\mathbf{Y} \in \mathbb{R}^{N_l \times p}$ with rows $\boldsymbol{y}^{(i)}$ for $i \in [N_l]$.
  if `recovery_method` = "Least-Squares" **then**
    $\boldsymbol{\mathcal{H}}^{(l)} = \underset{\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d \times \cdots \times d \times p}}{\arg\min} \|\mathbf{X}(\boldsymbol{\mathcal{T}})_{\langle\langle l,1 \rangle\rangle} - \mathbf{Y}\|_F^2$.
  else if `recovery_method` = "Nuclear Norm" **then**
    $\boldsymbol{\mathcal{H}}^{(l)} = \underset{\boldsymbol{\mathcal{T}} \in \mathbb{R}^{d \times \cdots \times d \times p}}{\arg\min} \|(\boldsymbol{\mathcal{T}})_{\langle\langle \lceil l/2 \rceil, l - \lceil l/2 \rceil + 1 \rangle\rangle}\|_*$ subject to $\|\mathbf{X}(\boldsymbol{\mathcal{T}})_{\langle\langle l,1 \rangle\rangle} - \mathbf{Y}\| \leq \varepsilon$.
  else if `recovery_method` = "IHT" **or** `recovery_method` = "TIHT" **then**
    Initialize $\boldsymbol{\mathcal{H}}^{(l)} \in \mathbb{R}^{d \times \cdots \times d \times p}$.
    repeat
      $(\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1 \rangle\rangle} \leftarrow (\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1 \rangle\rangle} + \gamma \mathbf{X}^\top (\mathbf{Y} - \mathbf{X}(\boldsymbol{\mathcal{H}}^{(l)})_{\langle\langle l,1 \rangle\rangle})$
      $\boldsymbol{\mathcal{H}}^{(l)} \leftarrow$ `project`$(\boldsymbol{\mathcal{H}}^{(l)}, R)$ (using either SVD for IHT or TT-SVD for TIHT)
    until convergence
  else if `recovery_method` = "SGD" **or** `recovery_method` = "ALS" **then**
    Initialize all cores of the rank R TT-decomposition $\boldsymbol{\mathcal{H}}^{(l)} = [\![\boldsymbol{\mathcal{G}}_1^{(l)}, \cdots, \boldsymbol{\mathcal{G}}_{l+1}^{(l)}]\!]$.
    // Note that $\boldsymbol{\mathcal{H}}^{(l)}$ is never explicitly constructed.
    repeat
      for $i = 1, \cdots, l+1$ **do**
        $$\boldsymbol{\mathcal{G}}_i^{(l)} \leftarrow \begin{cases} \boldsymbol{\mathcal{G}}_i^{(l)} - \gamma \nabla_{\boldsymbol{\mathcal{G}}_i^{(l)}} \|\mathbf{X}([\![\boldsymbol{\mathcal{G}}_1^{(l)}, \cdots, \boldsymbol{\mathcal{G}}_{l+1}^{(l)}]\!])_{\langle\langle l,1 \rangle\rangle} - \mathbf{Y}\|_F^2 & \text{for SGD} \\ \underset{\boldsymbol{\mathcal{G}}_i^{(l)}}{\arg\min} \|\mathbf{X}([\![\boldsymbol{\mathcal{G}}_1^{(l)}, \cdots, \boldsymbol{\mathcal{G}}_{l+1}^{(l)}]\!])_{\langle\langle l,1 \rangle\rangle} - \mathbf{Y}\|_F^2 & \text{for ALS} \end{cases}$$
      end for
    until convergence
  end if
end for

22 Let $(\mathscr{H}^{(2L)})_{\langle\langle L,L+1\rangle\rangle} = \mathbf{PS}$ be a rank R factorization, then return the linear 2-RNN $(\boldsymbol{h}_0, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ where

$$\begin{aligned}\boldsymbol{h}_0 &= (\mathbf{S}^\dagger)^\top (\mathscr{H}_f^{(L)})_{\langle\langle L+1\rangle\rangle}, \quad \boldsymbol{\Omega}^\top = \mathbf{P}^\dagger (\mathscr{H}_f^{(L)})_{\langle\langle L,1\rangle\rangle} \\ \boldsymbol{\mathcal{A}} &= ((\mathscr{H}_f^{(2L+1)})_{\langle\langle L,1,L+1\rangle\rangle}) \times_1 \mathbf{P}^\dagger \times_3 (\mathbf{S}^\dagger)^\top\end{aligned}$$

4.5.3 Leveraging the low rank structure of the Hankel tensors

While the least-squares method is sufficient to obtain the theoretical guarantees of Theorem 10, it does not leverage the low rank structure of the Hankel tensors $\mathscr{H}^{(L)}$, $\mathscr{H}^{(2L)}$ and $\mathscr{H}^{(2L+1)}$. We now propose several alternative recovery methods to leverage this structure, in order to improve both sample complexity and time complexity. The sample efficiency and running time of these methods will be assessed in a simulation study in Section 4.6 (deriving improved sample complexity guarantees using these methods is left for future work).

We first propose two alternatives to solving the least-squares problem $\mathbf{Y} = \mathbf{X}(\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle}$ that leverage the low matrix rank structure of the Hankel tensor. Indeed, knowing that $(\mathscr{H}^{(l)})_{\langle\langle \lceil l/2\rceil, l-\lceil l/2\rceil+1\rangle\rangle}$ can be approximately low rank (if the target function is computed by a WFA with a small number of states), one can achieve better sample complexity by taking into account the fact that the effective number of parameters needed to describe this matrix can be significantly lower than its number of entries. The first approach is to reformulate the least-squares problem as a nuclear norm minimization problem (see line 6 of Algorithm 5). The nuclear norm is the tightest convex relaxation of the matrix rank and the resulting optimization problem can be solved using standard convex optimization toolbox [Candes and Plan, 2011, Recht et al., 2010]. A second approach is a non-convex optimization algorithm: iterative hard thresholding (IHT) [Jain et al., 2010] (see lines 7-12 of Algorithm 5). This optimization method is iterative and boils down to a projected gradient descent algorithm: at each iteration, the Hankel tensor is updated by taking a

step in the direction opposite to the gradient of the least-squares objective, before being projected onto the manifold of low rank matrices using truncated SVD. More precisely, first the following gradient update is performed:

$$\begin{aligned}(\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle} &\leftarrow (\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle} - \gamma \nabla_{(\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle}} \|\mathbf{X}(\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle} - \mathbf{Y}\|_F^2 \\ &= (\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle} + \gamma \mathbf{X}^\top(\mathbf{Y} - \mathbf{X}(\mathscr{H}^{(l)})_{\langle\langle l,1\rangle\rangle})\end{aligned}$$

where γ is the learning rate. Then, a truncated SVD of the matricization $(\mathscr{H}^{(l)})_{\langle\langle \lceil l/2\rceil, l-\lceil l/2\rceil+1\rangle\rangle}$ is performed to obtain a low rank approximation of the Hankel tensor.

Both the nuclear norm minimization and the iterative hard thresholding algorithm only leverages the fact that the matrix rank of $(\mathscr{H}^{(l)})_{\langle\langle \lceil l/2\rceil, l-\lceil l/2\rceil+1\rangle\rangle}$ is small. However, as we have shown in Section 4.3.1, the Hankel tensor $\mathscr{H}^{(l)}$ exhibits a stronger structure: it is of low tensor train rank (which implies that *any* of its matricization is a low rank matrix). We now present three methods leveraging this structure for the recovery of the Hankel tensors from linear measurements. The first optimization algorithm is tensor iterative hard thresholding (TIHT) [Rauhut et al., 2017] which is the tensor generalization of IHT. Similarly to IHT, TIHT is a projected gradient descent algorithm where the projection step consists in projecting the Hankel tensor onto the manifold of tensors with low tensor train rank (instead of projecting onto the set of low rank matrices): after the gradient update described above, a low rank tensor train approximation of the Hankel tensor $\mathscr{H}^{(l)}$ is computed using the TT-SVD algorithm [Oseledets, 2011].

Even though TIHT leverages the tensor train structure of the Hankel tensors to obtain better sample complexity, its computational complexity remains high since the Hankel tensor $\mathscr{H}^{(l)}$ needs to alternatively be converted between its dense form (for the gradient descent step) and its tensor train decomposition (for the projection steps). Observe here that the size of these two objects significantly differs: the full Hankel tensor $\mathscr{H}^{(l)}$ has size $d^l p$ whereas the number of parameters of its tensor train decomposition is only in $\mathcal{O}\left(ldR^2 + pR\right)$, where R is the rank of the tensor train decomposition. Similarly to the effi-

cient learning algorithm in the tensor train format presented in Section 4.3.2, the recovery of the Hankel tensors can be carried out in the tensor train format without never having to explicitly construct the tensor $\mathscr{H}^{(l)}$. We conclude by presenting two optimization methods to recover the Hankel tensors from data directly in the tensor train format. For both methods, the Hankel tensor $\mathscr{H}^{(l)}$ is never explicitly constructed but parameterized by the core tensors $\mathscr{G}_1, \cdots, \mathscr{G}_{l+1}$ of its TT decomposition:

$$\mathscr{H}^{(l)} = [\![\mathscr{G}_1, \cdots, \mathscr{G}_{l+1}]\!].$$

Both methods are iterative and will optimize the least-squares objective with respect to each of the core tensors in turn until convergence. The first one is the alternating least-squares algorithm (ALS), which is one of the workhorse of tensor decomposition algorithms [Kolda and Bader, 2009]. In ALS, at each iteration a least-squares problem is solved in turn for each one of the cores of the TT decomposition:

$$\mathscr{G}_i \leftarrow \arg\min_{\mathscr{G}_i} \|\mathbf{X}([\![\mathscr{G}_1, \cdots, \mathscr{G}_{l+1}]\!])_{\langle\langle l,1\rangle\rangle} - \mathbf{Y}\|_F^2 \text{ for } i = 1, \cdots, l+1.$$

The second one consists in simply using gradient descent to perform a gradient step with respect to each one of the core tensors at each iteration:

$$\mathscr{G}_i \leftarrow \mathscr{G}_i - \gamma \nabla_{\mathscr{G}_i} \|\mathbf{X}([\![\mathscr{G}_1, \cdots, \mathscr{G}_{l+1}]\!])_{\langle\langle l,1\rangle\rangle} - \mathbf{Y}\|_F^2 \text{ for } i = 1, \cdots, l+1$$

where γ is the learning rate. Both methods are described in lines 15-19 of Algorithm 5. Combining these two optimization methods with the spectral learning algorithm in the tensor train format described in Section 4.3.2 results in an efficient learning algorithm to estimate a linear 2-RNN from training data, where the Hankel tensors are never explicitly constructed but always manipulated in the tensor train format.

To conclude this section, we briefly mention that the ALS and gradient descent algorithms can straightforwardly be adapted to perform optimization with respect to mini-

batches instead of the whole training dataset. This allows us to further scale the algorithm to large training sets.

4.6 Experiments

In this section, we perform experiments on two toy examples to compare how the choice of the recovery method (`LeastSquares`, `NuclearNorm`, `IHT`, `TIHT`, `ALS` and `Gradient Descent`) affects the sample efficiency of Algorithm 5, and the corresponding computation time. We also report the performance obtained by refining the solutions returned by our algorithm (with both TIHT and ALS recovery methods) using stochastic gradient descent (`TIHT+SGD`, `ALS+SGD`). In addition, we perform experiments on a real world dataset of wind speed data from TUDelft, which is used in [Lin et al., 2016]. For the real world data, we include the original results for competitive approaches from [Lin et al., 2016].

4.6.1 Synthetic data

We perform experiments on two toy problems: recovering a random 2-RNN from data and a simple addition task. For the random 2-RNN problem, we randomly generate a linear 2-RNN with 5 units computing a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ by drawing the entries of all parameters $(\boldsymbol{h}_0, \boldsymbol{\mathcal{A}}, \boldsymbol{\Omega})$ independently from a normal distribution $\mathcal{N}(0, 0.2)$. The training data consists of 3 independently drawn sets $D_l = \{((\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}), \boldsymbol{y}^{(i)})\}_{i=1}^{N_l} \subset (\mathbb{R}^d)^l \times \mathbb{R}^p$ for $l \in \{L, 2L, 2L+1\}$ with $L = 2$, where each $\boldsymbol{x}_j^{(i)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and where the outputs can be noisy, i.e. $\boldsymbol{y}^{(i)} = f(\boldsymbol{x}_1^{(i)}, \boldsymbol{x}_2^{(i)}, \cdots, \boldsymbol{x}_l^{(i)}) + \boldsymbol{\xi}^{(i)}$ where $\boldsymbol{\xi}^{(i)} \sim \mathcal{N}(0, \sigma^2)$ for some noise variance parameter σ^2. For the addition problem, the goal is to learn a simple arithmetic function computing the sum of the running differences between the two components of a sequence of 2-dimensional vectors, i.e. $f(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k) = \sum_{i=1}^{k} \boldsymbol{v}^\top \boldsymbol{x}_i$ where $\boldsymbol{v}^\top = (-1 \;\; 1)$. The 3 training datasets are generated using the same process as above and a constant

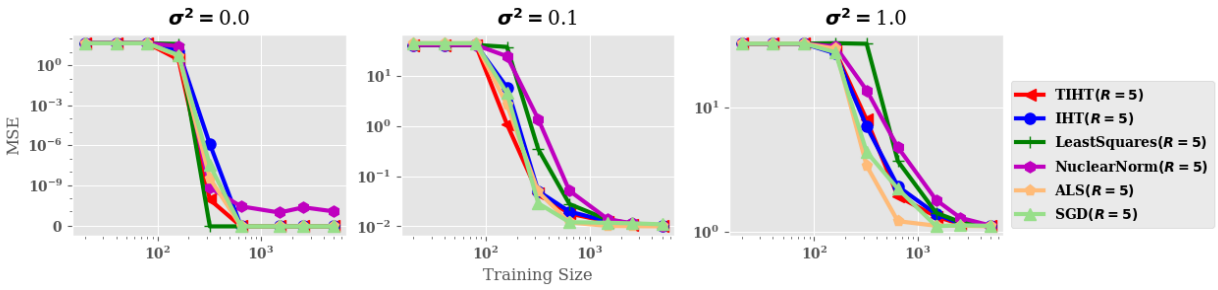


Figure 4.4: Average MSE as a function of the training set size for learning a random linear 2-RNN with different values of output noise.

entry equal to one is added to all the input vectors to encode a bias term (one can check that the resulting function can be computed by a linear 2-RNN with 2 hidden units).

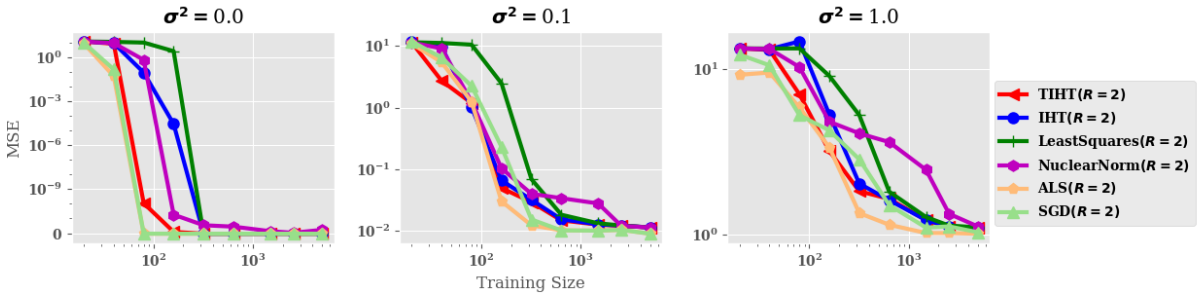


Figure 4.5: Average MSE as a function of the training set size for learning a simple arithmetic function with different values of output noise.

We run the experiments for different sizes of training data ranging from $N = 20$ to $N = 5,000$ (we set $N_L = N_{2L} = N_{2L+1} = N$) and we compare the different methods in terms of mean squared error (MSE) on a test set of $1,000$ sequences of length 6 generated in the same way as the training data (note that the training data only contains sequences of length up to 5). The IHT/TIHT methods sometimes returned aberrant models (due to numerical instabilities), we used the following scheme to circumvent this issue: when the training MSE of the hypothesis was greater than the one of the zero function, the zero function was returned instead (we applied this scheme to all other methods in the exper-

iments). For the gradient descent approach, we use the autograd method from Pytorch with the Adam [Kingma and Ba, 2014] optimizer with learning rate 0.001.

Results

The results are reported in Figure 4.4 and 4.5 where we see that all recovery methods lead to consistent estimates of the target function given enough training data. This is the case even in the presence of noise (in which case more samples are needed to achieve the same accuracy, as expected). We can also see that `TIHT` and `ALS` tend to be overall more sample efficient than the other methods (especially with noisy data), showing that taking the low rank structure of the Hankel tensors into account is profitable. Moreover, `TIHT` tends to perform better than its matrix counterpart, confirming our intuition that leveraging the tensor train structure is beneficial.

We also found that using gradient descent to refine the learned 2-RNN model often leads to a performance boost. In Figure 4.6 and Figure 4.7 we show the advantage MSE obtained by fine-tuning the learned 2-RNN using gradient descent. We use Pytorch to implement the fine-tuning process with the Adam optimizer with a learning rate of 0.0001. Fine-tuning helps the model to converge to the optimal solution with less data, resulting in a more sample-efficient approach. Lastly, we briefly mention that on these two tasks, previous experiments showed that both non-linear and linear recurrent neural network architectures trained with the back-propagation algorithm performed significantly worse than the spectral learning based learning algorithm we propose (see Rabusseau et al. [2019]), where we experiment with LSTM of both linear and nonlinear (tanh) activation function. The LSTM with linear activation function always underperforms the nonlinear counterpart, where all our methods show significantly better sample efficiency than nonlinear LSTM. The structure of the LSTM is of one layer with 20 hidden units (tanh activation for nonlinearity) and a fully-connected output layer. The training is done using Adam optimizer with a learning rate of 0.001.

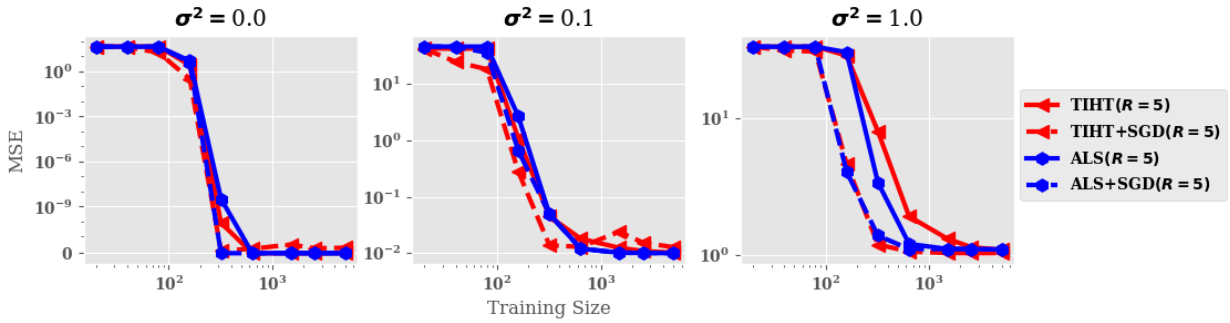


Figure 4.6: Performance comparison between vanilla methods and fine-tuned methods on Random 2-RNN problem.

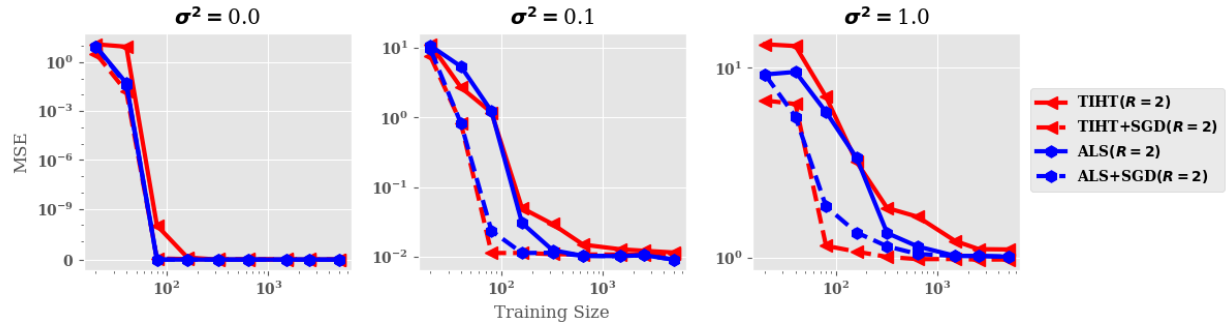


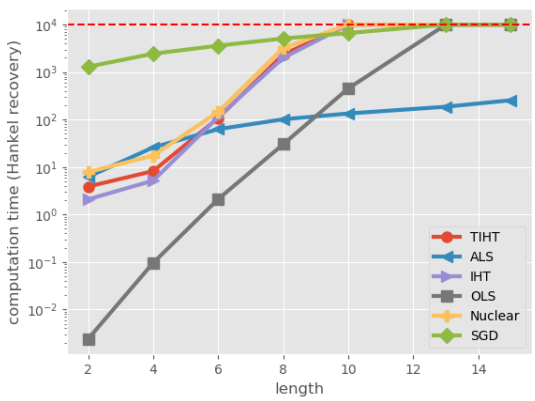
Figure 4.7: Performance comparison between vanilla methods and fine-tuned methods on Addition problem.

Running time analysis

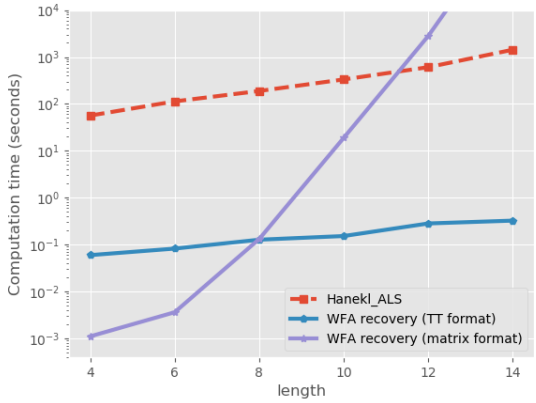
By directly recovering the Hankel tensor in its tensor train form, `ALS` and `SGD` significantly reduces the computation time needed to recover the Hankel tensor. In Figure 4.8a, we report the computation time of the different Hankel recovery methods for Hankel tensors with various length (L). The experiment is performed with 1,000 examples for the addition problem and all iterative methods (excluding `OLS`) are stopped when reaching the same fixed training accuracy. In the figure, there is a clear reduction in computation time for both `ALS` and `SGD` compared to other methods, which is expected. More specifically, these methods have much smaller computation time growth rate with respect to the length L compared to the matrix-based methods. This is especially beneficial when

dealing with data that exhibits long term dependencies of the input variables. In comparison to the Hankel tensor recovery time, the spectral learning step takes significantly less time, typically within a second. However, one important note is that if the length L gets larger, directly performing spectral learning on the matrix form of the Hankel tensor may not be possible due to the curse of dimensionality. Therefore, under this circumstance, one should directly perform the spectral learning algorithm in its tensor train form as described in Section 4.3.2.

To demonstrate the benefits of performing the spectral learning algorithm in the TT format (as described in Section 4.3.2), we perform an additional experiment showing that leveraging the TT format allows one to save significant amount of computation time and memory resources in the spectral learning phase, especially when the corresponding Hankel tensor is large (i.e. large length and input dimension). In Figure 4.8b we compare the running time of the spectral learning phase (after recovering the Hankel tensors) in the matrix and TT formats, where the latter leverages the TT structure in the spectral learning routine. We randomly generate 100,000 input-output examples using a Random 2-RNN with 3 states, input dimension 5 and output dimension 1. We use ALS to recover the Hankel tensors in the TT format and compare the running time of the spectral learning in the TT format with the time needed to perform the classical spectral learning algorithm after reshaping the Hankel tensors in matrices (note that the time needed to convert the TT Hankel tensors into the corresponding Hankel matrices is not counted towards the matrix spectral learning time). In Figure 4.8b, we report the time needed to recover the Hankel tensors from data (`Hankel_ALS`) and the time to recover the WFA in both the matrix and TT formats. One can observe that although classic matrix-based spectral learning is significantly faster than the TT-based one when the length is relatively small, the running time of the matrix method grows exponentially with the length while the one of the TT method is linear. For example, when the length equals to 12, TT spectral learning is more than 1,000 times faster than the classic spectral learning. This computation time gap significantly shows the benefit of leveraging TT format in the spectral learning phase. One



(a) Computation time comparison between different Hankel recovery methods on addition problem with 1,000 data. Computation time is capped at 10,000 seconds for all methods (the red dashed line).



(b) Computation time comparison between TT-spectral learning and classic matrix based spectral learning on Random 2-RNN problem with 100,000 examples. The ground truth 2-RNN has 3 states, with input dimension 5 and output dimension 1.

Figure 4.8: Running time comparison

remark is that other types of Hankel tensor recovery methods that we mentioned (i.e. TIHT, IHT, LeastSquares and NuclearNorm) fail to scale in this setup, due to excessive memory required by these algorithms in preparing the training data.

In addition, directly recovering the Hankel tensors and performing spectral learning in the TT format also helps drastically reduce the memory resources. As an illustration, we compare the size of the Hankel matrix in the TT format and the matrix format in Table 4.1. As one can see the size of the matrix version of the Hankel grows exponentially w.r.t the length while the TT Hankel size grows linearly. This also echoes with the computation time for these two methods.

Length	4	6	8	10	12	14
TT Hankel Size (GB)	2.68e-06	4.69e-06	6.70e-06	8.71e-06	1.07e-05	1.27e-05
Matrix Hankel Size (GB)	1.40e-05	3.49e-04	8.73e-03	2.20e-01	5.40e-01	136

Table 4.1: Memory size of the Hankel tensor $\mathscr{H}^{(\ell)}$ for the random 2-RNN problem (see Figure 4.8b) in both TT and matrix formats.

4.6.2 Real world data

In addition to the synthetic data experiments presented above, we conduct experiments on the wind speed data from TUDelft [§]. For this experiment, to compare with existing results, we specifically use the data from Rijnhaven station as described in Lin et al. [2016], which proposed a regression automata model and performed various experiments on the wind speed dataset. The data contains wind speed and related information at the Rijnhaven station from 2013-04-22 at 14:55:00 to 2018-10-20 at 11:40:00 and was collected every five minutes. To compare with the results in [Lin et al., 2016], we strictly followed the data preprocessing procedure described in the chapter. We use the data from 2013-04-23 to 2015-10-12 as training data and the rest as our testing data. The chapter uses SAX as a preprocessing method to discretize the data. However, as there is no need to discretize data for our algorithm, we did not perform this procedure. For our method, we set the length $L = 3$ and we use a window size of 6 to predict the future values at test time. We calculate hourly averages of the wind speed, and predict one/three/six hour(s) ahead, as in [Lin et al., 2016]. In this experiment, our model only predicts the next hour from the past 6 observations. To make k-hour-ahead prediction, we use the forecast of the model itself as input and bootstrap from it. For our methods we use a linear 2-RNN with 10 states. Averages over 5 runs of this experiment for one-hour-ahead, three-hour-ahead, six-hour-ahead prediction error can be found in Table 4.2, 4.3 and 4.4. The results for RA, RNN and persistence are taken directly from [Lin et al., 2016], where the RNN structure is selected to be LSTM with 3 layers and 15 hidden neurons with ReLu activation function.

The results of this experiment are presented in Table 4.2-4.4 where we can see that while TIHT+SGD performs slightly worse than ARIMA and RA for one-hour-ahead prediction, it outperforms all other methods for three-hours and six-hours ahead predictions (and the superiority w.r.t. other methods increases as the prediction horizon gets longer). One important note is that although ALS and ALS+SGD is slightly under-performing

[§]http://weather.tudelft.nl/csv/

compared to TIHT and TIHT+SGD, the computation time has been significantly reduced for ALS by a factor of 5 (TIHT takes 3,542 seconds while ALS takes 804 seconds).

Table 4.2: One-hour-ahead Speed Prediction Performance Comparisons

Method	TIHT	TIHT +SGD	ALS	ALS +SGD	Regression Automata	ARIMA	RNN	Persistence
RMSE	0.573	0.519	0.586	0.522	0.500	**0.496**	0.606	0.508
MAPE	21.35	18.79	22.12	19.01	**18.58**	18.74	24.48	18.61
MAE	0.412	0.376	0.423	0.388	0.363	**0.361**	0.471	0.367

Table 4.3: Three-hour-ahead Speed Prediction Performance Comparisons

Method	TIHT	TIHT +SGD	ALS	ALS +SGD	Regression Automata	ARIMA	RNN	Persistence
RMSE	0.868	**0.854**	0.875	0.864	0.872	0.882	1.002	0.893
MAPE	33.98	**31.70**	34.67	32.13	32.52	33.165	37.24	33.29
MAE	0.632	**0.624**	0.648	0.628	0.632	0.642	0.764	0.649

Table 4.4: Six-hour-ahead Speed Prediction Performance Comparisons

Method	TIHT	TIHT +SGD	ALS	ALS +SGD	Regression Automata	ARIMA	RNN	Persistence
RMSE	1.234	**1.145**	1.283	1.128	1.205	1.227	1.261	1.234
MAPE	49.08	**44.88**	47.65	45.03	46.809	48.02	47.03	48.11
MAE	0.940	**0.865**	0.932	0.869	0.898	0.919	0.944	0.923

4.7 Conclusion and Future Directions

We proposed the first provable learning algorithm for second-order RNN with linear activation functions: we showed that linear 2-RNN are a natural extension of vv-WFA to the setting of input sequences of *continuous vectors* (rather than discrete symbol) and we extended the vv-WFA spectral learning algorithm to this setting. We also presented novel connections between WFA and tensor networks, showing that the computation of a WFA is intrinsically linked with the tensor train decomposition. We leveraged this connection

to adapt the standard spectral learning algorithm to the tensor train format, allowing one to scale up the spectral algorithm to exponentially large sub-blocks of the Hankel matrix.

We believe that the results presented in this chapter open a number of exciting and promising research directions on both the theoretical and practical perspectives. We first plan to use the spectral learning estimate as a starting point for gradient based methods to train non-linear 2-RNN. More precisely, linear 2-RNN can be thought of as 2-RNN using LeakyRelu activation functions with negative slope 1, therefore one could use a linear 2-RNN as initialization before gradually reducing the negative slope parameter during training. The extension of the spectral method to linear 2-RNN also opens the door to scaling up the classical spectral algorithm to problems with large discrete alphabets (which is a known caveat of the spectral algorithm for WFA) since it allows one to use low dimensional embeddings of large vocabularies (using e.g. word2vec or latent semantic analysis). From the theoretical perspective, we plan on deriving learning guarantees for linear 2-RNN in the noisy setting (e.g. using the PAC learnability framework). Even though it is intuitive that such guarantees should hold (given the continuity of all operations used in our algorithm), we believe that such an analysis may entail results of independent interest. In particular, analogously to the matrix case studied in [Cai et al., 2015], obtaining optimal convergence rates for the recovery of the low TT-rank Hankel tensors from rank one measurements is an interesting direction; such a result could for example allow one to improve the generalization bounds provided in [Balle and Mohri, 2012] for spectral learning of general WFA. Lastly, establishing other equivalence results between classical classes of formal languages and functions computed by recurrent architectures is a worthwhile endeavor; such equivalence results give a novel light on classical models from theoretical computer science and linguistics while at the same time sparkling original perspectives on modern machine learning architectures. A first direction could be to establish connections between weighted tree automata and tree-structured neural models such as recursive tensor neural networks [Socher et al., 2013b,a].

Chapter 5

Nonlinear Weighted Finite Automata

Spectral learning and WFAs present several attractive features in their learning properties. However, as previously noted, their expressivity is limited as they are linear models. This limited expressivity often results in a large or infinite state representation size, which can negatively impact computation and learning efficiency. In the next two chapters, we will delve into the methods of incorporating nonlinearities into WFAs and their continuous counterparts.

In this chapter, we present the Nonlinear Weighted Finite Automata (NL-WFAs) model. The inspiration behind NL-WFAs stems from the realization that the low-rank factorization of the Hankel matrix in the spectral learning algorithm is equivalent to an encoder-decoder neural network with linear activation functions. To incorporate nonlinearities, we use a nonlinear encoder-decoder to decompose the Hankel matrix and then proceed with the remaining steps of the spectral learning process. The transition functions are learned through the use of gradient descent. The expressive power of NL-WFA and the proposed learning algorithm are assessed on both synthetic and real world data, showing that NL-WFA can lead to smaller model sizes and infer complex grammatical structures from data. This chapter is based on my publication [Li et al., 2018].

5.1 Introduction

Although WFA have been successfully applied in various areas of machine learning, they are inherently linear models: their computation boils down to the composition of linear maps. Recent positive results in machine learning have shown that models based on composing nonlinear functions are both very expressive and able to capture complex structures in data. For example, by leveraging the expressive power of deep convolutional neural networks in the context of reinforcement learning, agents can be trained to outperform humans in Atari games [Mnih et al., 2013] or to defeat world-class go players [Silver et al., 2016]. Deep convolutional networks have also recently led to considerable breakthroughs in computer vision [Krizhevsky et al., 2012], where they showed their ability to disentangle the complex structure of the data by learning a representation which unfold the original complex feature space (where the data lies on a low-dimensional manifold) into a representation space where the structure has been linearized. It is thus natural to wonder to which extent introducing non-linearity in WFA could be beneficial. We will show that both these advantages of nonlinear models, namely their expressiveness and their ability to learn rich representations, can be brought to the classical WFA computational model.

In this chapter, we propose a nonlinear WFA model (NL-WFA) based on neural networks, along with a learning algorithm. In contrast with WFA, the computation of an NL-WFA relies on *successive compositions of nonlinear mappings*. This model can be seen as an extension of dynamical recognizers [Moore, 1997] — which are in some sense a nonlinear extension of deterministic finite automata — to the quantitative setting. In contrast with the training of recurrent neural networks (RNN), our learning algorithm does not rely on back-propagation through time. It is inspired by the spectral learning algorithm for WFA, which can be seen as a two-step process: first, find a low-rank factorization of the so-called *Hankel matrix* leading to a natural embedding of the set of words into a low-dimensional vector space, and then performing regression in this representation space

to recover the transition matrices. Similarly, our learning algorithm first finds a nonlinear factorization of the Hankel matrix using an auto-encoder network, thus learning a rich nonlinear representation of the set of strings, and then performs nonlinear regression using a feed-forward network to recover the transition operators in the representation space.

Related works. NL-WFA and RNN are closely related: their computation relies on the composition of nonlinear mappings directed by a sequence of observations. In this paper, we explore a somehow orthogonal direction to the recent RNN literature by trying to connect such models back with classical computational models from formal language theory. Such connections have been explored in the past in the non-quantitative setting with dynamical recognizers [Moore, 1997], whose inference has been studied in e.g. [Pollack, 1991]. The ability of RNN to learn classes of formal languages has also been investigated, see e.g. [Avcu et al., 2017] and references therein. It is well known that predictive state representations (PSRs) [Littman and Sutton, 2002] are strongly related to WFA [Thon and Jaeger, 2015]. A nonlinear extension of PSR has been proposed for deterministic controlled dynamical systems in [Rudary and Singh, 2004]. More recently, building upon reproducing kernel Hilbert space embedding of PSR [Boots et al., 2013], non-linearity is introduced into PSR using recurrent neural networks [Downey et al., 2017, Venkatraman et al., 2017]. In addition to the RNNs based models, [Hefny et al., 2015] proposed a two-stage regression framework as well as a nonlinear extension. Then, by leveraging this two-stage framework with inference machine [Sun et al., 2016] proposed another different approach to tackle nonlinear dynamics.

One of the main differences with these approaches is that our learning algorithm does not rely on back-propagation through time and we instead investigate how the spectral learning method for WFA can be beneficially extended to the nonlinear setting. However, although bearing some similarities, the major difference between our work and theirs is that we want to build a nonlinear model for sequential data while they focus on filtering and prediction tasks (the model is not the actual goal). Moreover, they focus on

continuous dynamical systems, which are also the major focus for [Downey et al., 2017, Venkatraman et al., 2017], but we are mostly interested in discrete tasks. Although it is true that the factorization step in our learning algorithm is related to the S1A and S1B stages in [Hefny et al., 2015], the regression step of our method (which is at the core of the NL-WFA model) cannot be seen as an instantiation of that framework. In addition, it is not clear how performing non-linear regression in S1A and S1B in [Hefny et al., 2015] would result in a nonlinear computational model.

5.2 Nonlinear Weighted Finite Automata

The WFA model assumes that the transition operators $\mathbf{A}_\sigma$ are linear. It is natural to wonder whether this linear assumption sometimes induces a too strong model bias (e.g. if one tries to learn a function that is not recognizable by a WFA). Moreover, even for recognizable functions, introducing non-linearity could potentially reduce the number of states needed to represent the function. Consider the following example: given a WFA $A = \langle \boldsymbol{\alpha}, \boldsymbol{\alpha}_\infty, \{\mathbf{A}_\sigma\} \rangle$, the function $(f_A)^2 : u \mapsto f_A(u)^2$ is recognizable and can be computed by the WFA $A' = \langle \boldsymbol{\alpha}', \boldsymbol{\alpha}'_\infty, \{\mathbf{A}'_\sigma\} \rangle$ with $\boldsymbol{\alpha}' = \boldsymbol{\alpha} \otimes \boldsymbol{\alpha}$, $\boldsymbol{\alpha}'_\infty = \boldsymbol{\alpha}_\infty \otimes \boldsymbol{\alpha}_\infty$ and $\mathbf{A}'_\sigma = \mathbf{A}_\sigma \otimes \mathbf{A}_\sigma$, where $\otimes$ denotes Kronecker product. One can check that if $\mathrm{rank}(f_A) = k$, then $\mathrm{rank}(f_{A'})$ can be as large as k^2, but intuitively the *true dimension* of the model is k using non-linearity*. These two observations motivate us to introduce *nonlinear WFA* (NL-WFA).

5.2.1 Definition of NL-WFA

We will use the notation $\tilde{g}$ to stress that a function g may be nonlinear. We define a NL-WFA $\tilde{A}$ of with k states as a tuple $\langle \boldsymbol{\alpha}, \tilde{G}_\lambda, \{\tilde{G}_\sigma\}_{\sigma \in \Sigma} \rangle$, where $\boldsymbol{\alpha} \in \mathbb{R}^k$ is a vector of initial weights, $\tilde{G}_\sigma : \mathbb{R}^k \to \mathbb{R}^k$ is a transition function for each $\sigma \in \Sigma$ and $\tilde{G}_\lambda : \mathbb{R}^k \to \mathbb{R}$ is a

*By applying the spectral method on the component-wise square root of the Hankel matrix of A', one would recover the WFA A of rank k.

termination function. A NL-WFA $\tilde{A}$ computes a function $f_{\tilde{A}} : \Sigma^* \to \mathbb{R}$ defined by

$$f_{\tilde{A}}(x) = \tilde{G}_\lambda(\tilde{G}_{x_t}(\cdots \tilde{G}_{x_2}(\tilde{G}_{x_1}(\boldsymbol{\alpha}_0))\cdots))$$

for any word $x = x_1 x_2 \cdots x_t \in \Sigma^*$. Similarly to the linear case, we will sometimes use the shorthand notation $\tilde{G}_x = \tilde{G}_{x_t} \circ \tilde{G}_{x_{t-1}} \circ \cdots \circ \tilde{G}_{x_1}$. This nonlinear model can be seen as a generalization of dynamical recognizers [Moore, 1997] to the quantitative setting. It is easy to see that one recovers the classical WFA model by restricting the functions $\tilde{G}_\sigma$ and $\tilde{G}_\lambda$ to be linear. Of course, some restrictions on these nonlinear functions have to be imposed in order to control the expressiveness of the model. In this paper, we consider nonlinear functions computed by neural networks.

5.2.2 A Representation learning perspective on the spectral algorithm

Our learning algorithm is inspired by the spectral learning method for WFA. In order to give some insights and further motivate our approach, we will first show how the spectral method can be interpreted as a representation learning scheme.

The spectral method can be summarized as a two-stage process consisting of a *factorization step* and a *regression step*: first, find a low rank factorization of the Hankel matrix and then perform regression to estimate the transition operators $\{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}$.

First, focusing on the factorization step, let us observe that one can naturally embed the set of prefixes into the vector space $\mathbb{R}^{\mathcal{S}}$ by mapping each prefix u to the corresponding row of the Hankel matrix $\mathbf{H}_{u,:}$. However, it is easy to check that this representation is highly redundant when the Hankel matrix is of low rank. In the factorization step of the spectral learning algorithm, the rank k factorization $\mathbf{H} = \mathbf{PS}$ can be seen as finding a low dimensional representation $\mathbf{P}_{u,:} \in \mathbb{R}^k$ for each prefix u, from which the original *Hankel representation* $\mathbf{H}_{u,:}$ can be recovered using the linear map $\mathbf{S}$ (indeed $\mathbf{H}_{u,:} = \mathbf{P}_{u,:}\mathbf{S}$). We can formalize this encoder-decoder perspective by defining two maps $\Psi_p : \mathscr{P} \mapsto \mathbb{R}^k$ and $\Psi_s : \mathbb{R}^k \mapsto \mathbb{R}^{\mathcal{S}}$ by $\Psi_p(u)^\top = \mathbf{P}_{u,:}$ and $\Psi_s(\boldsymbol{x})^\top = \boldsymbol{x}^\top\mathbf{S}$. One can easily check that

$\Psi_s(\Psi_p(u))^\top = \mathbf{H}_{u,:}$, which implies that $\Psi_p(u)$ encodes all the information sufficient to predict the value $f(uv)$ for any suffix $v \in \mathcal{S}$ (indeed $f(uv) = \Psi_p(u)^\top \mathbf{S}_{:,v}$).

The regression step of the spectral algorithms consists in recovering the matrices $\mathbf{A}_\sigma$ satisfying $\mathbf{H}_\sigma = \mathbf{P}\mathbf{A}_\sigma\mathbf{S}$. From our encoder-decoder perspective, this can be seen as recovering the compositional mappings $\mathbf{A}_\sigma$ satisfying $\Psi_p(u\sigma)^\top = \Psi_p(u)^\top \mathbf{A}_\sigma$ for each $\sigma \in \Sigma$.

It follows from the previous discussion that non-linearity could be beneficially brought to WFA and into the spectral learning algorithm in two ways: first by using nonlinear methods to perform the factorization of the Hankel matrix, thus discovering a potentially nonlinear embedding of the Hankel representation, and second by allowing the compositional feature maps associated with each symbol to be nonlinear.

5.3 Learning NL-WFA

Introducing non-linearity can be achieved in several ways. In this paper, we will use neural networks due to their ability to discover relevant nonlinear low-dimensional representation spaces and their expressive power as function approximators.

5.3.1 Nonlinear factorization

Introducing non-linearity in the factorization step boils down to finding two mappings Ψ_p and Ψ_s such that $\Psi_s(\Psi_p(u)) = \mathbf{H}_{u,:}$ for any prefix $u \in \mathscr{P}$. Briefly going back to the linear case, one can check that if $\mathbf{H} = \mathbf{P}\mathbf{S}$, then we have $\mathbf{H}_{u,:} = \mathbf{H}_{u,:}\mathbf{S}^+\mathbf{S}$ for each prefix u, implying that the encoder-decoder maps satisfy $\Psi_p(u)^\top = \mathbf{H}_{u,:}\mathbf{S}^+$ and $\Psi_s(\boldsymbol{x})^\top = \boldsymbol{x}^\top\mathbf{S}$. Thus the factorization step can essentially be interpreted as finding an auto-encoder able to project down the Hankel representation $\mathbf{H}_{u,:}$ to a low dimensional space while preserving the relevant information captured by $\mathbf{H}_{u,:}$.

How to extend the factorization step to the nonlinear setting should now appear clearly: by training an auto-encoder to learn a low-dimensional representation of the

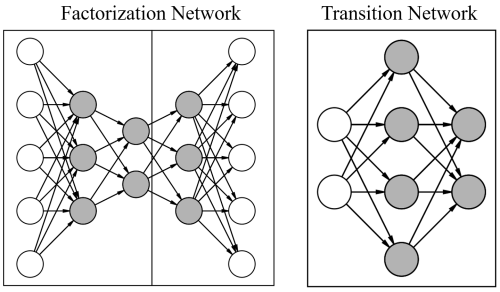


Figure 5.1: Factorization network and transition network: grey units are nonlinear while white ones are linear.

Hankel representations $\mathbf{H}_{u,:}$, one will potentially unravel a rich representation of the set of prefixes from which an NL-WFA can be recovered.

Let $\tilde{\phi} : \mathbb{R}^{\mathcal{S}} \mapsto \mathbb{R}^k$ and $\tilde{\phi}' : \mathbb{R}^k \to \mathbb{R}^{\mathcal{S}}$ be the encoder and decoder maps respectively. We will train the auto-encoder shown in Figure 5.1 (left) to achieve

$$\tilde{\phi}'(\tilde{\phi}(\mathbf{H}_{u,:})) \simeq \mathbf{H}_{u,:}.$$

More precisely, if $\mathbf{H} \in \mathbb{R}^{m \times n}$, the model is trained to map the original Hankel representation $\mathbf{H}_{u,:} \in \mathbb{R}^n$ of each prefix u to a latent representation vector in $\mathbb{R}^k$, where $k \ll n$, and then map this vector back to the original representation $\mathbf{H}_{u,:}$. This is achieved by minimizing the reconstruction error (i.e. the ℓ_2 distance between the original representation and its reconstruction). Instead of linearly factorizing the Hankel matrix, we use an auto-encoder framework consisting of two networks, whose hidden layer activation functions are nonlinear[†].

More precisely, if we denote the nonlinear activation function by θ, and we let $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ be the weights matrices from the left to the right of the neural net shown in Figure 5.1 (left), the function $\hat{f} : \mathbb{R}^n \to \mathbb{R}^n$ computed by the auto-encoder can be written as

$$\hat{f} = \tilde{\phi}' \circ \tilde{\phi} : (\mathbf{H})^\top_{u,:} \mapsto \theta(\theta(\theta(\mathbf{H}^\top_{u,:}\mathbf{A})^\top\mathbf{B})^\top\mathbf{C})^\top\mathbf{D}$$

[†]We use the (component-wise) $\tanh$ function in our experiments.

where the encoder-decoder functions $\tilde{\phi} : \mathbb{R}^n \to \mathbb{R}^k$ and $\tilde{\phi}' : \mathbb{R}^k \to \mathbb{R}^n$ are defined by $\tilde{\phi}(\boldsymbol{x})^\top = \theta(\theta(\boldsymbol{x}^\top \mathbf{A})^\top \mathbf{B})$ and $\tilde{\phi}'(\boldsymbol{h})^\top = \theta(\boldsymbol{h}^\top \mathbf{C})^\top \mathbf{D}$ for vectors $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{h} \in \mathbb{R}^k$ and n is the number of suffixes.

It is easy to check that if the activation function θ is the identity, one will exactly recover a rank k factorization of the Hankel matrix, thus falling back onto the classical factorization step of the spectral learning algorithm.

5.3.2 Nonlinear regression

Given the encoder-decoder maps $\tilde{\phi}$ and $\tilde{\phi}'$, we then move on to recovering the transition functions. Recall that we wish to find the compositional feature maps $\tilde{G}_\sigma : \mathbb{R}^k \to \mathbb{R}^k$ for each σ satisfying $\Psi_p(u\sigma) = \tilde{G}_\sigma(\Psi_p(u))$ for all $u \in \mathscr{P}$. Using the encoder map $\tilde{\phi}$ obtained in the factorization step, the mapping Ψ_p can be written as $\Psi_p(u) = \tilde{\phi}(\mathbf{H}_{u,:})$.

In order to learn these transition maps, we will thus train one neural network for each symbol σ to minimize the following squared error loss function

$$\sum_{u \in \mathscr{P}} \|\tilde{G}_\sigma(\tilde{\phi}(\mathbf{H}_{u,:})) - \tilde{\phi}(\mathbf{H}_{u\sigma,:})\|^2.$$

The structure of the simple feed-forward network used to learn the transition maps is shown in Figure 5.1 (right). Let $\mathbf{E}, \mathbf{F}$ be the two weights matrices, the function $\hat{g} : \mathbb{R}^k \to \mathbb{R}^k$ computed by this network can be written as

$$\hat{g} : \boldsymbol{h}^\top \mapsto \theta(\theta(\boldsymbol{h}^\top \mathbf{E})^\top \mathbf{F})$$

We want to point out that both hidden units and output units of this network are nonlinear. Since this network will be trained to map between latent representations computed by the factorization network, the output units of the transition network and the units corresponding to the latent representation in the factorization network should be of the same nature to facilitate the optimization process.

5.3.3 Overall learning algorithm

Let $(\mathscr{P}, \mathscr{S}) \subset \Sigma^* \times \Sigma^*$ be a basis of suffixes and prefixes such that $\lambda \in \mathscr{P} \cap \mathscr{S}$. Let $(\mathscr{P}', \mathscr{S})$ be its p-closure (i.e. $\mathscr{P}' = \mathscr{P} \cup \mathscr{P}\Sigma$) and let $m = |\mathscr{P}'|,\ n = |\mathscr{S}|$. For reasons that will be clarified in the next section, we assume that $\mathscr{P}$ is prefix-closed (i.e. for any $x \in \mathscr{P}$, all prefixes of x also belong to $\mathscr{P}$). The first step consists in building the estimate $\mathbf{H} \in \mathbb{R}^{m \times n}$ of the Hankel matrix from the training data (by using e.g. the empirical frequencies in the train set), where the rows of $\mathbf{H}$ are indexed by prefixes in $\mathscr{P}' = \mathscr{P} \cup \mathscr{P}\Sigma$ and its columns by suffixes in $\mathscr{S}$. The learning algorithm for NL-WFA then consists of two steps:

1. Train the factorization network to obtain a nonlinear decomposition of the Hankel matrix $\mathbf{H}$ through the mappings $\tilde{\phi} : \mathbb{R}^n \to \mathbb{R}^k$ and $\tilde{\phi}' : \mathbb{R}^k \to \mathbb{R}^n$ satisfying

$$\tilde{\phi}'(\tilde{\phi}(\mathbf{H}_{u,:})) \simeq \mathbf{H}_{u,:} \text{ for all } u \in \mathscr{P} \cup \mathscr{P}\Sigma. \tag{5.1}$$

2. Train the transition networks for each symbol $\sigma \in \Sigma$ to learn the transition maps $\tilde{G}_\sigma : \mathbb{R}^k \to \mathbb{R}^k$ satisfying

$$\tilde{G}_\sigma(\tilde{\phi}(\mathbf{H}_{u,:})) \simeq \tilde{\phi}(\mathbf{H}_{u\sigma,:}) \text{ for all } u \in \mathscr{P}. \tag{5.2}$$

The resulting NL-WFA is then given by $\tilde{A} = \langle \boldsymbol{\alpha}_0, \tilde{G}_\lambda, \{\tilde{G}_\sigma\}_{\sigma \in \Sigma} \rangle$ where $\boldsymbol{\alpha} = \tilde{\phi}(\mathbf{H}_{\lambda,:})$ and $\tilde{G}_\lambda$ is defined by

$$\tilde{G}_\lambda(\boldsymbol{x}) = \boldsymbol{\lambda}^\top \tilde{\phi}'(\boldsymbol{x}) \text{ for all } \boldsymbol{x} \in \mathbb{R}^k$$

where $\boldsymbol{\lambda}$ is the one-hot encoding of the empty suffix λ.

5.3.4 Theoretical analysis

While the definitions of the initial vector $\boldsymbol{\alpha}_0$ and termination function G_λ given above may seem *ad-hoc*, we will now show that the learning algorithm we derived corresponds to minimizing an error loss function between $f_{\tilde{A}}(u)$ and the estimated value $\mathbf{H}_{u,\lambda}$ over all

prefixes in $\mathscr{P}$. Intuitively, this means that our learning algorithm aims at minimizing the empirical squared error loss over the training set $\mathscr{P} \subset \Sigma^*$. More formally, we show in the following theorem that if both the factorization network and the transition networks are trained to optimality (i.e. they both achieve 0 training error), then the resulting NL-WFA exactly recovers the values given in the first column of the estimate of the Hankel matrix.

Theorem 11. *If the prefix set $\mathscr{P}$ is prefix-closed and if equality holds in Eq. (5.1) and Eq. (5.2), then the NL-WFA $\tilde{A} = \langle \boldsymbol{\alpha}_0, \tilde{G}_\lambda, \{\tilde{G}_\sigma\}_{\sigma\in\Sigma}\rangle$, where $\boldsymbol{\alpha} = \tilde{\phi}(\mathbf{H}_{\lambda,:})$ and $\tilde{G}_\lambda : \boldsymbol{x} \mapsto \boldsymbol{\lambda}^\top \tilde{\phi}'(\boldsymbol{x})$, is such that $f_{\tilde{A}}(u) = \mathbf{H}_{u,\lambda}$ for all $u \in \mathscr{P}$.*

Proof. We first show by induction on the length of a word $u = u_1 u_2 \cdots u_t \in \mathscr{P}$ that

$$\tilde{G}_u(\boldsymbol{\alpha}_0) = \tilde{G}_{u_t}(\tilde{G}_{u_{t-1}}(\cdots \tilde{G}_1(\boldsymbol{\alpha}_0)\cdots)) = \tilde{\phi}(\mathbf{H}_{u,:}).$$

If $u = \sigma \in \Sigma$, using the fact that $\lambda \in \mathscr{P}$ we have $\tilde{G}_\sigma(\boldsymbol{\alpha}_0) = \tilde{G}_\sigma(\tilde{\phi}(\mathbf{H}_{\lambda,:})) = \tilde{\phi}(\mathbf{H}_{\sigma,:})$ by Eq. (5.2). Now if $u = u_1 u_2 \cdots u_t \in \mathscr{P}$, we can apply the induction hypothesis on $u_1 u_2 \cdots u_{t-1}$ (since $\mathscr{P}$ is prefix-closed) to obtain $\tilde{G}_u(\boldsymbol{\alpha}_0) = \tilde{G}_{u_t}(\tilde{G}_{u_1\cdots u_{t-1}}(\boldsymbol{\alpha}_0)) = \tilde{G}_{u_t}(\tilde{\phi}(\mathbf{H}_{u_1\cdots u_{t-1},:})) = \tilde{\phi}(\mathbf{H}_{u,:})$ by Eq. (5.2).

To conclude, for any $u \in \mathscr{P}$ we have $f_{\tilde{A}}(u) = \tilde{G}_\lambda(\tilde{G}_u(\boldsymbol{\alpha}_0)) = \tilde{G}_\lambda(\tilde{\phi}(\mathbf{H}_{u,:})) = \boldsymbol{\lambda}^\top \tilde{\phi}'(\tilde{\phi}(\mathbf{H}_{u,:})) = \mathbf{H}_{u,:}\boldsymbol{\lambda} = \mathbf{H}_{u,\lambda}$ by Eq. (5.1).

□

Intuitively, it follows that the learning algorithm described in Section 5.3.3 aims at minimizing the following loss function

$$\begin{aligned} J(\tilde{\phi}, \tilde{\phi}', \{\tilde{G}_\sigma\}_{\sigma\in\Sigma}) &= \sum_{u\in\mathscr{P}} (\boldsymbol{\lambda}^\top \tilde{\phi}'(\tilde{G}_u(\tilde{\phi}(\mathbf{H}_{\lambda,:})) - \mathbf{H}_{u,\lambda})^2 \\ &= \sum_{u\in\mathscr{P}} (f_{\tilde{A}}(u) - \hat{f}(u))^2 \end{aligned}$$

where $\hat{f}(u)$ is the estimated value of the target function on the word u, and where the NL-WFA $\tilde{A}$ is a function of the encoder-decoder maps $\tilde{\phi}, \tilde{\phi}'$ and of the transition maps $\tilde{G}_\sigma$ as described in Section 5.3.3.

Even though Theorem 11 seems to suggest that our learning algorithm is prone to over-fitting, but this is not the case. Indeed, akin to the linear spectral learning algorithm, the restriction on the number of states of the NL-WFA (which corresponds to the size of the latent representation layer in the factorization network) induces regularization and enforces the learning process to discriminate between signal and noise (i.e. in practice, the networks will not achieve 0 error due to the bottleneck structure of the factorization network).

5.3.5 Applying non-linearity independently in the factorization and transition networks

We have shown that non-linearity can be introduced into the two steps of our learning algorithm. We can thus consider three variants of this algorithm where we either apply non-linearity in the factorization step only, in the regression step only, or in both steps. It is easy to check that these three different settings correspond to three different NL-WFA models depending on whether the termination function only is nonlinear, the transition functions only are nonlinear, or both the termination and transition functions are nonlinear. Indeed, recall that that a NL-WFA $\tilde{A}$ is defined as a tuple $\tilde{A} = \langle \boldsymbol{\alpha}, \tilde{G}_\lambda, \{\tilde{G}_\sigma\}_{\sigma \in \Sigma} \rangle$. If no non-linearity are introduced in the factorization network, the termination function will have the form

$$\tilde{G}_\lambda : \boldsymbol{x} \mapsto \boldsymbol{\lambda}^\top \tilde{\phi}'(\boldsymbol{x}) = \boldsymbol{\lambda}^\top \mathbf{D}^\top \mathbf{C}^\top \boldsymbol{x}$$

(using the notations from the previous sections), which is linear. Similarly, if no non-linearity are used in the transition networks, the resulting maps $\tilde{G}_\sigma$ will be linear.

One may argue that only applying non-linearity in the termination function $\tilde{G}_\lambda$ would not lead to an expressive enough model. However, it is worth noting that in this case, after

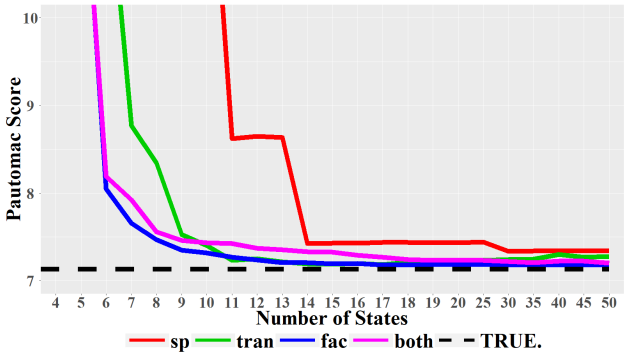


Figure 5.2: Pautomac score for the Dyck language experiment for different model sizes (trained on a sample size of 20,000).

the nonlinear factorization step, even though the transition functions are linear, they are operating on a nonlinear feature space. This is similar in spirit to the kernel trick, where a linear model is learned in a feature space resulting from a nonlinear transformation of the initial input space. Moreover, if we go back to the example of the squared function $(f_A)^2$ for some WFA A with k states (see beginning of Section 5.2), even though $(f_A)^2$ may have rank up to k^2, one can easily build a NL-WFA with k states computing $(f_A)^2$ where only the termination function is nonlinear.

5.4 Experiments

We compare the classical spectral learning algorithm with the three configurations of our neural networks based NL-WFA learning algorithms: applying non-linearity only in the factorization step (denoted by *fac.non*), only in the regression step (denoted by *tran.non*), and in both phases (denoted by *both.non*). We will perform experiments on a grammatical inference task (i.e. learn a distribution over Σ^* from samples drawn from this distribution) with both synthetic and real data

5.4.1 Metrics

We use two metrics to evaluate the trained models on a test set: Pautomac score and word error rate.

- The Pautomac score was first proposed for the Pautomac challenge [Verwer et al., 2014] and is defined by

$$Pauto(M) = -2^{\sum_{x \in T} P_*(x) \log(P_M(x))}$$

  where $P_M(x)$ is the normalized probability assigned to x by the learned model and $P_*(x)$ is the normalized true probability (both P_M and P_* are normalized to sum to 1 over the test set T). Since the models returned by both our method and the spectral learning algorithm are not ensured to output positive values, while the logarithm of a negative value is not defined, we take the absolute values of all the negative outputs.

- The word error rate (WER) measures the percentage of incorrectly predicted symbols when, given each prefix of strings in the test set, the most likely next symbol is predicted. To calculate WER, we are using the prefix version of the Hankel matrix. In the next subsection, we will discuss how to obtain WER for our models.

5.4.2 Calculating Word Error Rate

Word error rate reports the element-wise letter prediction error percentage. Given a predictor $\mathscr{M} : \Sigma^* \mapsto \sigma$, and a test word $x = x_1, x_2, \cdots, x_n$ of size n, the calculation of the word error rate can be performed in the following procedure:

1. Use the empty word as the first input, and get the prediction $\hat{x}_1 = \mathscr{M}(\lambda)$.
2. Use the word x_1 as the input to predict x_2: $\hat{x}_2 = \mathscr{M}(x_1)$
3. Use the word x_1, x_2 as the input to predict x_3: $\hat{x}_2 = \mathscr{M}(x_1, x_2)$

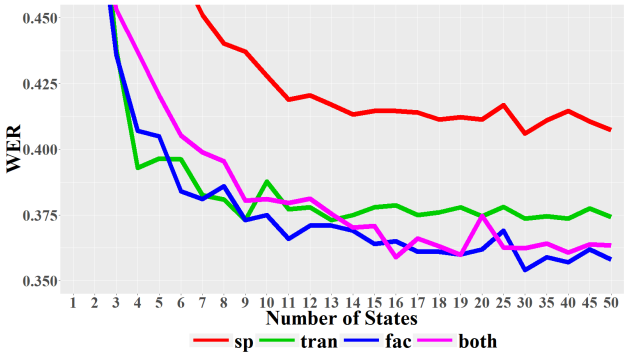


Figure 5.3: Word error rate for the Dyck language experiment for different model sizes (trained on a sample size of 20,000).

4. Keep adding the letter to the input word and get new predictions.

5. The word error rate can be computed by:

$$\frac{\sum_{i=1}^{n} \mathbb{1}(\hat{x}_i \neq x_i)}{n}$$

Then the problem is how to derive a predictor using the automaton model. For a linear automaton, we will do so by maintaining a vector $\boldsymbol{v}$ as the model receiving letters. For a word $x = x_1, x_2, \cdots, x_n$, if the model has already received the prefix $u = x_1, \cdots, x_t$, then we will have:

$$\boldsymbol{v}_u = \frac{\boldsymbol{\alpha}^\top \mathbf{A}_u}{\boldsymbol{\alpha}^\top \mathbf{A}_u (\mathbf{I} - \sum_{\sigma \in \Sigma} \mathbf{A}_\sigma)^{-1} \boldsymbol{\omega}}$$

To predict the next letter x_{t+1}, we will obtain the score $\mathbb{S}_\sigma$ corresponding to each of the letter $\sigma \in \Sigma$ by:

$$\mathbb{S}_\sigma = \boldsymbol{v}_u^\top \mathbf{A}_\sigma (\mathbf{I} - \sum_{\sigma \in \Sigma} \mathbf{A}_\sigma)^{-1} \boldsymbol{\omega}$$

We will compare all the scores obtained, and pick the letter with the highest score to be the prediction for the next symbol.

It is easy to check, for a word x, the probability of u being a prefix is $\mathbb{P}(u\Sigma^*) = \boldsymbol{\alpha}^\top \mathbf{A}_u (\mathbf{I} - \sum_{\sigma \in \Sigma} \mathbf{A}_\sigma)^{-1} \boldsymbol{\omega}$. Then the probability of u specifically followed by the letter σ

can be computed by $\boldsymbol{\alpha}^\top \mathbf{A}_u \mathbf{A}_\sigma (\mathbf{I} - \sum_{\sigma \in \Sigma} \mathbf{A}_\sigma)^{-1} \boldsymbol{\omega}$. In fact $\mathbb{S}_\sigma = \frac{\mathbb{P}(u\sigma\Sigma^*)}{\mathbb{P}(u\Sigma^*)}$, which is the conditional probability that u will be followed by σ given u has appeared.

However, this treatment is only true if $\mathbf{A}_\sigma$ is a matrix, while for tran.non and both.non, we have nonlinear transition functions. To solve this problem, we introduce the prefix version of the Hankel matrix, defined as $\mathbf{H}^p_{u,v} = \mathbb{P}(uv\Sigma^*)$ for the function $f_p(x) = \mathbb{P}(x\Sigma^*)$. For linear case, assume $\mathbf{H}^p = \mathbf{P}^p \mathbf{S}^p$ is a rank factorization, define the automaton $A_p = \langle \boldsymbol{\alpha}^p, \mathbf{A}^p_\sigma, \boldsymbol{\omega}^p \rangle$, where $\boldsymbol{\alpha} = \mathbf{P}^p_{\lambda,:}$, $\mathbf{A}_\sigma = (\mathbf{p}^p)^+ \mathbf{H}_\sigma (\mathbf{S}^p)^+$ and $\boldsymbol{\omega}^p = \mathbf{S}^p_{:,\lambda}$. Then due to the duality between WFA and Hankel matrix, A_p is the minimal WFA for f_p. We can then compute the score for predicting the next letter:

$$\mathbb{S}_\sigma = \frac{\boldsymbol{\alpha}^{p\top} \mathbf{A}^p_u \mathbf{A}^p_\sigma \boldsymbol{\omega}^p}{\boldsymbol{\alpha}^{p\top} \mathbf{A}^p_u \boldsymbol{\omega}^p}$$

Therefore to calculate the score function for an NL-WFA, naturally, we can work with the prefix version of the Hankel matrix using our method. Namely, we first obtain a pair of encoder-decoder functions Φ_p and Φ'_p using $\mathbf{H}^p$ so that $\Phi'_p(\Phi_p(\mathbf{H}^p)) = \mathbf{H}^p$. Next we solve the regression problem using $\mathbf{H}^p_\sigma$, i.e. $\tilde{G}_\sigma(\Phi_p(\mathbf{H}^p)) = \Phi'_p(\mathbf{H}^p_\sigma)$. In the end, we obtain a NL-WFA $\tilde{A} = \langle \boldsymbol{\alpha}, \tilde{G}_\lambda, \{\tilde{G}_\sigma\}_{\sigma\in\Sigma} \rangle$, where $\boldsymbol{\alpha} = \Phi_p(\lambda)$ and $\tilde{G}_\lambda(x) = \Phi'_p(x)_{:,\lambda}$. Given this NL-WFA, we modify the score function as:

$$\mathbb{S}_\sigma = \frac{\tilde{G}_\lambda(\tilde{G}_\sigma(\tilde{G}_u(\boldsymbol{\alpha})))}{\tilde{G}_\lambda(\tilde{G}_u(\boldsymbol{\alpha}))}$$

Notice we use the exact same algorithm as before. The only difference is that we replace the classical Hankel matrix with the prefix version.

In the experiments, in order to calculate WER, we use the prefix version of the Hankel matrix for both the spectral learning method as well as our methods.

5.4.3 Synthetic data: probabilistic Dyck language

For the synthetic data experiment, we generate data from a probabilistic Dyck language. Let $\Sigma = \{[,]\}$, we consider the language generated by the following probabilistic context

free grammar

$$
\begin{aligned}
S &\to SS && \text{with probability } 0.2 \\
S &\to [S] && \text{with probability } 0.4 \\
S &\to [\,] && \text{with probability } 0.4
\end{aligned}
$$

i.e. starting from the symbol S, we draw one of the rules according to their probability and apply it to transform S into the corresponding right hand side; this process is repeated until no S symbol is left. One can check that this distribution will generate balanced strings of brackets. It is well known that this distribution cannot be computed by a WFA (since its support is a context free grammar). However, as a WFA can compute any distribution with finite support, it can model the restriction of this distribution to words of length less than some threshold N. By using this distribution for our synthetic experiments, we want to showcase the fact that NL-WFA can lead to models with better predictive accuracy when the number of states is limited and that they can better capture the complex structure of this distribution.

In our experiments, we use empirical frequencies in a testing data set to estimate the Hankel matrix $\mathbf{H}_{\mathscr{B}} \in \mathbb{R}^{1000\times 1000}$, where the p-closed basis $\mathscr{B}$ is obtained by selecting the $1,000$ most frequent prefixes and suffixes in the training data. We first assess the ability of NL-WFA to better capture the structure in the data when the number of states is limited. We compared the models for different model sizes k ranging from 1 to 50, where k is the number of states of the learned WFA and NL-WFA. For the latter, we used a three hidden layers structure for the factorization network where the number of hidden units are set to $2k$, k and $2k$. For the transition networks, we use a neural network with $2k$ hidden units[‡]. We used Adamax [Kingma and Ba, 2014] with learning rates 0.015 and 0.001 respectively to train these two networks.

[‡]These hyperparameters are not finely tuned, thus some optimization might potentially improve the results.

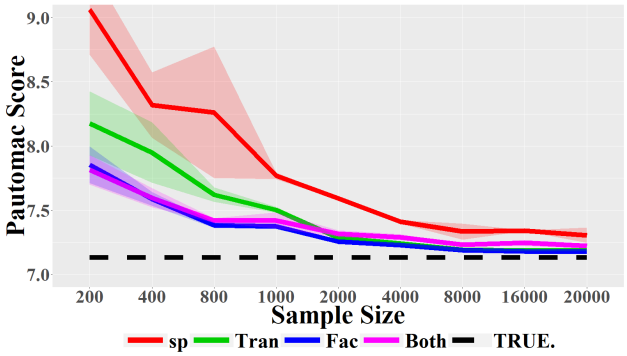


Figure 5.4: Average Pautomac score for learning the Dyck language with different sample sizes.

All models are trained on a training set of size $20,000$ and the Pautomac score and WER on a test set of size 250 are reported in Figure 5.2 and 5.3 respectively. For both metrics, we see that NL-WFA gives better results for small model sizes. While NL-WFA and WFA tend to perform similarly for the Pautomac score for larger model sizes, NL-WFA clearly outperforms WFA in terms of WER in this case. This shows that including non-linearity can increase the prediction power of WFA by discovering the underlying nonlinear structure and can be beneficial when dealing with a small number of states.

We then compared the sample complexity of learning NL-WFA and WFA by training the different models on training set of sizes ranging from 200 to $20,000$. For all models the rank is chosen by cross-validation. In Figure 5.4 and Figure 5.5, we show the performances for the four models on a test set of size 250 by reporting the average and standard deviation over 10 runs of this experiment. We can see that NL-WFA achieve better results on small sample sizes for the Pautomac score and consistently outperforms the linear model for all sample sizes for WER. This shows that NL-WFA can use the training data more efficiently and again that the expressiveness of NL-WFA is beneficial to this learning task.

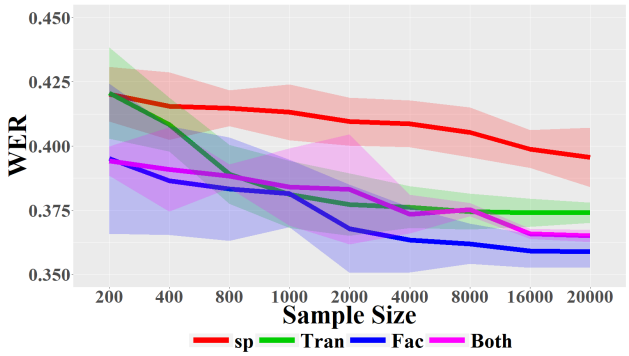


Figure 5.5: Average word error rate for learning the Dyck language with different sample sizes.

5.4.4 Synthetic data: Pautomac Chanllenge

In this experiment, we still use empirical frequency to estimate the sub-block of Hankel matrix $\mathbf{H}_{\mathscr{B}}$. Especially, for computational efficiency, we use the first 1000 most frequent prefixes and suffixes from the training set to construct our basis $\mathscr{B}$. The number of hidden units of both ϕ and ϕ' is set to 2000. For the size of the model, while for the transition function, the number of hidden units is two times the model size, where the model size ranges from 1 to 500 and this is set to be the number of states for spectral learning. We have also tried a different number of sample sizes from 1500 to 20000, and the size of the test set is 1000.

The dataset we used is from Pautomac challenge [Verwer et al., 2014]. We will also use the Pautomac score for the metric. We should keep in mind that all the datasets from Pautomac are generated from linear models, thus using nonlinear models to approximate might suffer from the intrinsic characteristics.

The datasets we use for this experiment are pautomac2 and pautomac3. For pautomac2, the result is shown in Figure 5.6 . From the figure we observed that when the sample size is relatively small, such as 1500 and 5000, without a large enough number of states, the original linear spectral learning will be outperformed by our methods, and when sample size increases, even with extreme small model size, nonlinear models still

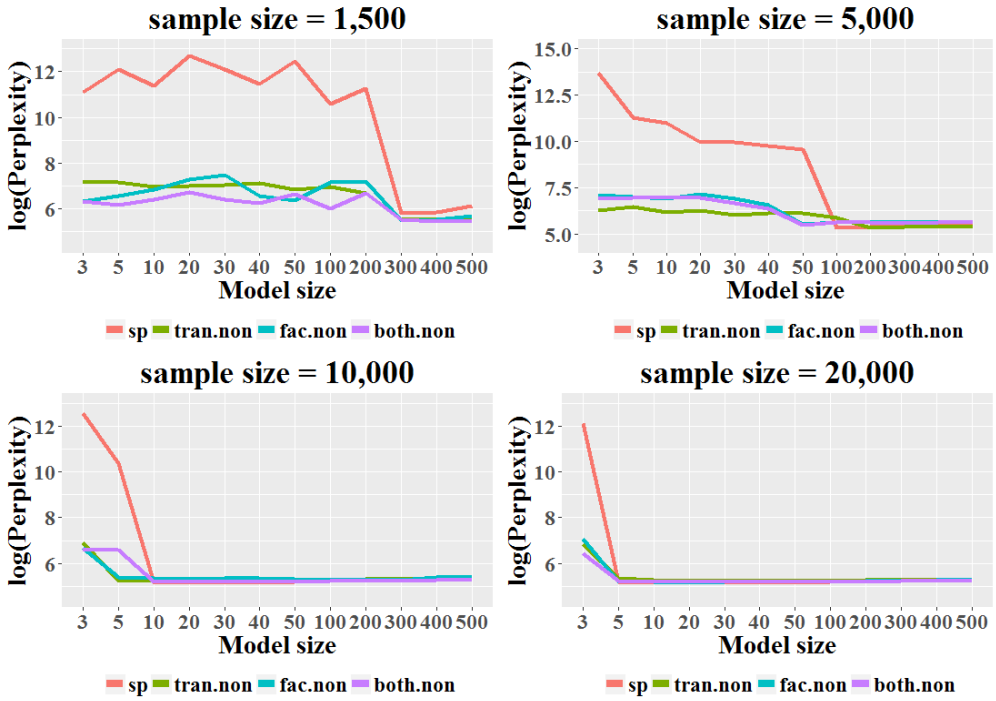


Figure 5.6: log(Perplexity) of Pautomac2 dataset based on a different number of states and sample size. The data is generated from an HMM with 63 states and 18 actions (letters).

show reasonable perplexity, indicating their superiority over the linear model. This thus gives us a confirmation that applying non-linearity can give us major advantages when dealing with a smaller number of states and insufficient data.

We have already covered the explanation of potential model size reduction when one applies non-linearity in the model in the Kronecker product example. Although the transition functions can appear to be linear, they may have nonlinear lower dimensional representation. Therefore, by using a nonlinear model, one can discover this underlying relationship and thus potentially reduce the model size.

On the other hand, for small sample sizes, spectral learning tends to have worse performance than our model, especially with small model size. This could potentially result from the expressiveness of the neural network as it can aggregate useful information from what the linear model deems to be noise. To be more specific, let us look at the graph with 1500 training samples. When we increase the number of states for spectral learning, it

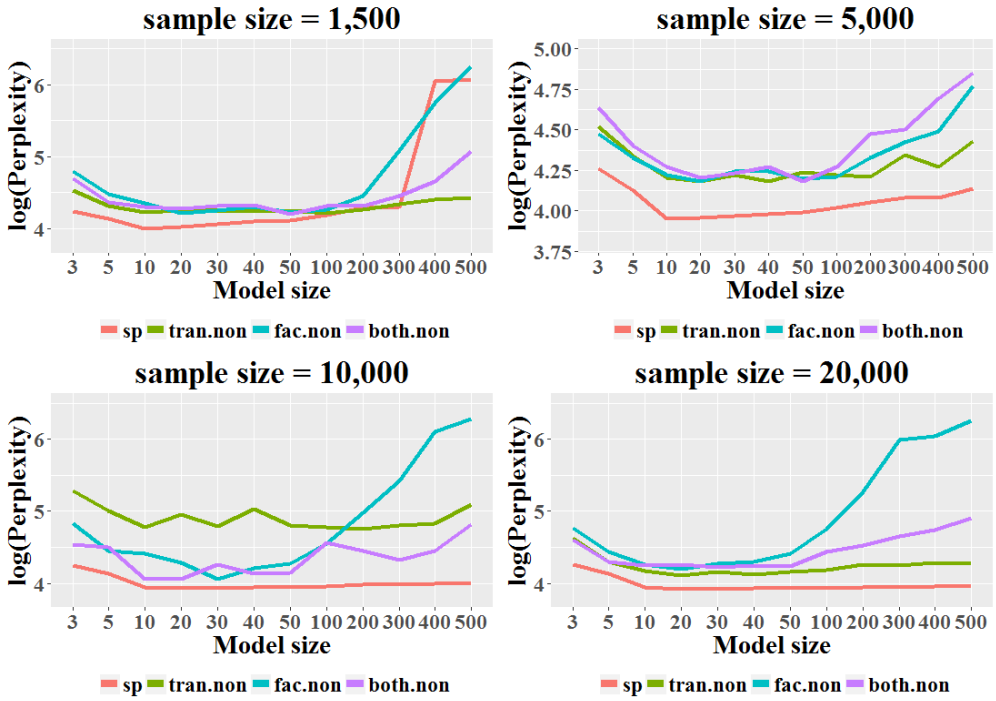


Figure 5.7: log(Perplexity) of Pautomac3 dataset based on different number of states and sample size. The data is generated from a PFA with 25 states and 4 actions (letters).

starts to perform well, and specifically, there is a significant improvement from 200 states to 300, indicating some of the states it believes useless turn out to carry very important information for the reconstruction of the model.

Nevertheless, this result is for a relatively more complex model in terms of the number of states and actions it has. When dealing with a simpler model, such as pautomac3, our experiment shows that non-linearity in this case could be detrimental. From Figure 5.7, we can see that in every sample size spectral learning outperforms our models. This is actually not surprising, as the underlying structure of the model is easier to capture and is essentially linear. However, more interestingly, we have seen the overfitting behavior that we did not see in the previous experiment, due to the simplicity of the underlying model. In general, nonlinear factorization suffers mostly from overfitting as shown in the graph, while nonlinear regression and applying non-linearity in both cases are less prone to overfitting and are relatively steady even compared to spectral learning.

5.4.5 Real data: Penn treebank

The Penn Treebank [Taylor et al., 2003] is a well-known benchmark dataset for natural language processing. It consists of approximately 7 million words of part-of-speech tagged text, 3 million words of skeletally parsed text, over 2 million words of text parsed for predicate argument structure, and 1.6 million words of transcribed spoken text annotated for speech disfluencies. In this experiment, we use a small portion of the Treebank dataset: the character level of English verbs which was used in the SPICE challenge [Balle et al., 2017]. This dataset contains 5,987 sentences over an alphabet of 33 symbols as the training set. It also provides two test sets of size 750. We used one of the test sets as a validation set and then tested our models on the other.

For this experiment, the Hankel matrix $\mathbf{H}_{\mathscr{B}}$ is of size 3000×300 where the prefixes and suffixes have been selected again by taking the most frequent in the training data. We used a five layers factorization network where the layers are of size $4k$, $2k$, k, $2k$ and $4k$ respectively, where k is the number of states of the NL-WFA. The structure of the transition networks is the same as in the previous experiment. For all models, the rank is selected using the validation set.

In the experiments, we compare the performance of NL-WFA with recurrent neural networks (RNNs), HMM (Using the Baum-Welch algorithm) and linear spectral learning. For RNNs, we use a three-layers LSTM networks with 128 units for each layer. We use RMSprop with 0.001 learning rate to optimize on the categorical entropy. The results are reported in Table 5.1 and Table 5.2. We can see that from the language modeling perspective (Pautomac score), our model (both.non) outperforms all the baselines for every sample size. While from prediction perspective, RNNs are the best model given enough data. However, with data being insufficient (sample size smaller than 3000), our method (fac.non) performs the best. This indicates our model's strong ability in modeling the distribution, as well as relatively good performance in prediction task, especially when dealing with small sample sizes. In addition, it is known that classical spectral learning method may give negative values even under the probabilistic setting, which can com-

Table 5.1: Log Pautomac Score For Real Data

Sample Size	SP	EM	RNN	Fac.non	Tran.non	Both.non
1000	9.098	4.252	4.765	8.005	3.480	**2.937**
2000	4.995	3.723	4.6053	4.874	3.374	**2.923**
3000	4.532	3.570	4.398	4.431	3.423	**2.894**
4000	4.235	3.542	4.244	4.166	3.198	**2.880**
ALL	4.234	3.496	4.191	4.144	3.098	**2.748**

Table 5.2: WER For Real Data

Sample Size	SP	EM	RNN	Fac.non	Tran.non	Both.non
1000	0.8432	0.808	0.806	**0.7630**	0.8834	0.8630
2000	0.8342	0.793	0.788	**0.7332**	0.8762	0.8435
3000	0.8195	0.781	0.736	**0.7134**	0.8679	0.8212
4000	0.8141	0.776	**0.692**	0.6935	0.8563	0.8098
ALL	0.8033	0.753	**0.669**	0.6831	0.8441	0.7910

promise the validity of the predicted distribution. However, we noticed that this issue has been significantly alleviated for our methods. This result further strengthens our conclusion on the language modeling aspect.

5.5 Discussion

We believe that trying to combine models from formal languages theory (such as weighted automata) and models that have recently led to several successes in machine learning (e.g. neural networks) is an exciting and promising line of research, both from the theoretical and practical sides. This work is a first step in this direction: we proposed a novel non-linear weighted automata model along with a learning algorithm inspired by the spectral learning method for classical WFA. We showed that non-linearity can be introduced in two ways in WFA, in the termination function or in the transition maps, which directly translates into the two steps of our learning algorithm.

In our experiment, we showed on both synthetic and real data that (i) NL-WFA can lead to models with better predictive accuracy than WFA when the number of states is limited, (ii) NL-WFA are able to capture the complex underlying structure of challenging

languages (such as the Dyck language used in our experiments) and (iii) NL-WFA exhibit better sample complexity when learning on data with a complex grammatical structure.

In the future, we intend to investigate further the properties of NL-WFA from both theoretical and experimental perspectives. For the former, one natural question is whether we could obtain learning guarantees for some specific classes of nonlinear functions. Indeed, one of the main advantages of the spectral learning algorithm is that it provides consistent estimators. While it may be difficult to obtain such guarantees when considering functions computed by neural networks, we believe that studying the case of more tractable nonlinear functions (e.g. polynomials) could be very insightful. We also plan on thoroughly investigating connections between NL-WFA and RNN. From the practical perspective, we want to first tune the hyper-parameters for NL-WFA more extensively on the current datasets to potentially improve the results. In addition, we intend to run further experiments on real data and on different kinds of tasks besides language modeling (e.g. classification, regression). Moreover, due to the strong connection between WFA and PSR, it will be very interesting to use NL-WFA in the context of reinforcement learning.

Chapter 6

Sequential Density Estimation via Nonlinear Continuous Weighted Finite Automata

The goal of this chapter is to investigate the possibilities of introducing nonlinearities into WFAs with continuous input space (CWFAs/linear 2-RNN) and to propose a spectral learning like algorithm to recover such WFAs. In this chapter, through the application of sequential density estimation, we illustrate that one can leverage a nonlinear feature map and a nonlinear output function to improve the expressivity of CWFAs. The choices of these functions correlate to the predefined tasks. In the scope of this chapter, we will focus on density estimation and propose the RNADE-NCWFA model. We present the specific form of the feature mapping as well as the output function for estimating the density function of a Gaussian HMM. Combining gradient descent and the spectral learning algorithm, we propose a spectral learning based method for the learning of RNADE-NCWFA. This chapter is based on my publication [Li et al., 2022a].

6.1 Introduction

One of the major applications of WFA is to approximate probability distribution over sequences of discrete symbols. Although the WFA model has been extended to the continuous domain [Li et al., 2020b, Rabusseau et al., 2019] as the so-called linear 2-RNN model (or continuous WFA model), approximating density functions for sequential data under continuous domain using this model is not straight-forward, as the model does not guarantee to compute a density function by construction. Moreover, due to the linearity of the model, the continuous WFA model (CWFA) is not expressive enough to estimate some of the common density functions over sequences of continuous random variables such as a Gaussian hidden Markov model.

In recent years, neural networks have been widely applied in density estimation and have been proven to be particularly successful. To estimate a density function via neural networks, the neural density estimator need to be flexible enough to represent complex densities but have tractable inference functions and learning algorithms. One particular example of such models is the class of autoregressive models [Uria et al., 2016, 2013], where the joint density is decomposed into a product of conditionals, and each conditional is approximated by a neural network. One other type of method is the so-called flow-based methods (normalizing flows) [Dinh et al., 2014, 2016, Rezende and Mohamed, 2015]. Flow-based methods transform a base density (e.g. a standard Gaussian) into the target density by an invertible transformation with tractable Jacobian. Although these methods have been used to estimate sequential densities, the sequences often come as fixed lengths. It is often unclear how to generalize these methods to account for varying lengths of the sequences in the testing phase, which can be important for some sequential tasks, such as language modeling for NLP tasks. Weighted finite automata, on the other hand, are designed to carry out such tasks under the discrete setting. The question is, how to generalize WFA to approximate density functions over continuous domains?

In this chapter, by extending the classic CWFA model with a (nonlinear) feature mapping and a (nonlinear) termination function, we first propose our nonlinear continuous weighted finite automata (NCWFA) model. Combining this model with the RNADE framework [Uria et al., 2013], we propose RNADE-NCWFA to approximate sequential density functions. The model is flexible as it naturally generalizes to sequences of varying lengths. Moreover, we show that the RNADE-NCWFA model is strictly more expressive than the Gaussian HMM model. In addition, we propose a spectral learning based algorithm for efficiently learning the parameters of an RNADE-NCWFA. For the empirical study, we conduct synthetic experiments using data generated from a Gaussian HMM model. We compare our proposed spectral learning of RNADE-NCWFA with HMM learned with the EM algorithm, RNADE with LSTM, and RNADE-NCWFA learned with stochastic gradient descent. We evaluate the models' performance through their log likelihood over sequences of unseen length, meaning the testing sequences are longer than the training sequences, to observe the model's generalization ability. We show that our model outperforms all the baseline models on this metric, especially for long testing sequences. Moreover, the advantage of our model is more significant when dealing with small training sizes and noisy data.

6.2 Related Works

In this section, we will first present some recent methods of density estimation. Then we will cover more details on the real-valued neural autoregressive density estimator (RNADE) [Uria et al., 2013], which is the inspiration for this work.

6.2.1 Density Estimation

In recent years, neural networks have been widely applied in density estimation and have been proven to be particularly successful. To estimate a density function via neural networks, the neural density estimator need to be flexible enough to represent complex

densities but have tractable inference functions and learning algorithms. One particular example of such models is the class of autoregressive models [Uria et al., 2016, 2013], where the joint density is decomposed into a product of conditionals, and each conditional is approximated by a neural network. One other type of method is the so-called flow-based methods (normalizing flows) [Dinh et al., 2014, 2016, Rezende and Mohamed, 2015]. Flow-based methods transform a base density (e.g., a standard Gaussian) into the target density by an invertible transformation with tractable Jacobian. Although these methods have been used to estimate sequential densities, the sequences often come as fixed lengths. It is often unclear how to generalize these methods to account for the varying length of the sequences in the testing phase, which can be important for some sequential tasks, such as language modeling for NLP tasks. Weighted finite automata, on the other hand, are designed to carry out such tasks under the discrete setting. The question is, how to generalize WFA to approximate density functions over continuous domains?

6.2.2 Real-valued neural autoregressive density estimator (RNADE)

RRNADE is a generalization of the original neural autoregressive density estimator (NADE) [Uria et al., 2016] to continuous variables. The core idea of RNADE is to estimate the joint density using the chain rule and approximate each conditional density via neural networks, i.e.

$$p(x_1, \cdots, x_n) = \prod_{i=1}^{n} p(x_i|x_{<i}) \;\text{ with }\; p(x_i|x_{<i}) = p_M(x_i|\theta_i), \tag{6.1}$$

where $x_{<i}$ denotes all attributes preceding $x_i \in \mathbb{R}$ in a fixed ordering, p_M is a mixture of m Gaussians with parameters $\theta_i = \{\boldsymbol{\beta}_i \in \mathbb{R}^m, \boldsymbol{\mu}_i \in \mathbb{R}^m, \boldsymbol{\sigma}_i \in \mathbb{R}^m\}$. Moreover, we have: $p_M(x_i|\theta_i) = \sum_{j=1}^{m} \boldsymbol{\beta}_i^j \mathcal{N}(x_i|\boldsymbol{\mu}_i^j, \boldsymbol{\sigma}_i^j)$, where $\boldsymbol{\beta}_i^j$ denotes the jth element of $\boldsymbol{\beta}_i$, same for $\boldsymbol{\mu}_i^j$ and $\boldsymbol{\sigma}_i^j$ and $\mathcal{N}(x_i|\boldsymbol{\mu}_i^j, \boldsymbol{\sigma}_i^j)$ denotes the Gaussian density with mean $\boldsymbol{\mu}_i^j$ and standard deviation $\boldsymbol{\sigma}_i^j$ evaluated at x_i. Note that $\boldsymbol{\beta}_i, \boldsymbol{\mu}_i, \boldsymbol{\sigma}_i$ are functions of $x_{<i}$. These functions are often

chosen to be various forms of neural networks. In the classic setting, RNADE with m mixing components and k hidden states has the following update rules:

$$\boldsymbol{h}_i = g_i(\boldsymbol{h}_{i-1}), \quad \boldsymbol{\beta}_i = \text{softmax}(\mathbf{V}_i^{\beta}\boldsymbol{h}_{i-1} + \boldsymbol{b}_i^{\beta}) \tag{6.2}$$

$$\boldsymbol{\mu}_i = \mathbf{V}_i^{\mu}\boldsymbol{h}_{i-1} + \boldsymbol{b}_i^{\mu}, \quad \boldsymbol{\sigma}_i = \exp(\mathbf{V}_i^{\sigma}\boldsymbol{h}_{i-1} + \boldsymbol{b}_i^{\sigma}), \tag{6.3}$$

where $\mathbf{V}_i^{\beta}, \mathbf{V}_i^{\mu}$ and $\mathbf{V}_i^{\sigma}$ are $m \times k$ matrices, $\boldsymbol{b}_i^{\beta}, \boldsymbol{b}_i^{\mu}$ and $\boldsymbol{b}_i^{\sigma}$ are vectors of size m, and g_i is an update function for the hidden state which is time step dependent (see [Uria et al., 2013] for more details on the specific update functions used in the original RNADE formulation). The softmax function [Bridle, 1990] ensures the mixing weights $\boldsymbol{\beta}$ are positive and sum to one and the exponential ensures the variances are positive. RNADE is trained to minimize the negative log likelihood: $\mathscr{L}(x_1 \cdots x_n, \theta_i) = -\sum_{i=1}^{n} \log(p_M(x_i|\theta_i))$ via gradient descent.

6.3 Methodology

To approximate density functions with CWFA, we need to improve the expressivity of the model and constrain it to compute a valid density function. In this section, we first introduce nonlinear continuous weighted finite automata. Then, we present RNADE-NCWFA for sequential density approximation, which combines CWFA with the RNADE framework. In the end, we show that RNADE-NCWFA is strictly more expressive than Gaussian HMM and present our spectral learning based algorithm for learning RNADE-NCWFA.

6.3.1 Nonlinear Continuous Weighted Finite Automata (NCWFAs)

To leverage CWFAs to estimate density functions, we first need to improve the expressivity of the model. We will do so by introducing a nonlinear feature map as well as a

nonlinear termination function. We hence propose the nonlinear continuous weighted finite automata (NCWFA) model as the following:

Definition 23. *A nonlinear continuous weighted finite automaton (NCWFA) is defined by a tuple* $\tilde{A} = \langle \boldsymbol{\alpha}, \xi, \phi, \mathscr{A} \rangle$*, where* $\boldsymbol{\alpha} \in \mathbb{R}^k$ *is the initial weight,* $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ *is the feature map,* $\xi : \mathbb{R}^k \to \mathbb{R}^p$ *is the termination function and* $\mathscr{A} \in \mathbb{R}^{k \times d \times k}$ *is the transition tensor. Given a sequence* $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_l$*, the function that the NCWFA* $\tilde{A}$ *computes is:*

$$\boldsymbol{h_0} = \boldsymbol{\alpha}, \quad \boldsymbol{h_t} = \mathscr{A} \bullet_1 \boldsymbol{h}_{t-1} \bullet_2 \phi(\boldsymbol{x}_t), \quad f_{\tilde{A}}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_l) = \xi(\boldsymbol{h}_l). \tag{6.4}$$

One immediate observation is that we can exactly recover the definition of a CWFA by letting $\phi(\boldsymbol{x}_i) = \boldsymbol{x}_i$ and $\xi(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{\Omega}$.

6.3.2 Density Estimation with NCWFAs

The second problem to tackle is that we need to constrain the NCWFA so that it can tractably compute a density function. In this section, we will leverage the RNADE method to propose the RNADE-NCWFAs model. The proposed model is flexible and can compute sequential densities of arbitrary sequence length. Moreover, we will show that this model is strictly more expressive than the classic Gaussian HMM model.

Recall the core idea of RNADE is to estimate the joint density using the chain rule as in Equation 6.1. Instead of approximating the conditionals via the classic RNADE treatment as in Equations 6.2, we use an NCWFA $\tilde{A} = \langle \boldsymbol{\alpha}, \xi, \phi, \boldsymbol{\mathcal{A}} \rangle$, i.e., $p(\boldsymbol{x}_i | \boldsymbol{x}_{<i}) = f_{\tilde{A}}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_i)$. One key difference with the classic RNADE model is that the state update function is independent of the time step, allowing the model to generalize to sequences of arbitrary lengths. However, an NCWFA does not readily compute a density function, as the function does not necessarily integrate to one and the output is non-negative. To overcome this issue, we adopt the approach used in RNADE by constraining the output

of the NCWFA to be a mixture of Gaussians with diagonal covariance matrices:

$$\phi(\boldsymbol{x}_i) = \tanh(\boldsymbol{x}_i^\top \mathbf{W}), \quad \boldsymbol{h}_i = \boldsymbol{\mathcal{A}} \bullet_1 \boldsymbol{h}_{i-1} \bullet_2 \phi(\boldsymbol{x}_i), \quad \boldsymbol{\beta}_i = \mathrm{softmax}(\mathbf{V}_i^\beta \boldsymbol{h}_{i-1} + \boldsymbol{b}_i^\beta) \tag{6.5}$$

$$\mathbf{M}_i = \boldsymbol{\mathcal{V}}^\mu \bullet_1 \boldsymbol{h}_{i-1} + \mathbf{B}^\mu, \quad \boldsymbol{\Sigma}_i = \exp(\boldsymbol{\mathcal{V}}^\sigma \bullet_1 \boldsymbol{h}_{i-1} + \boldsymbol{\mathcal{B}}^\sigma) \tag{6.6}$$

$$\xi(\boldsymbol{x}_i, \boldsymbol{h}_{i-1}) = \sum_{j=1}^{m} \boldsymbol{\beta}_i^j \mathcal{N}(\boldsymbol{x}_i | \mathbf{M}_i^j, \mathrm{diag}(\boldsymbol{\Sigma}_i^j)), \quad f_{\tilde{A}}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_l) = \xi(\boldsymbol{x}_l, \boldsymbol{h}_{l-1}) \tag{6.7}$$

where $\boldsymbol{h}_0 = \boldsymbol{\alpha}, \boldsymbol{\mathcal{V}}^\mu \in \mathbb{R}^{k \times m \times d}, \boldsymbol{\mathcal{V}}^\sigma \in \mathbb{R}^{k \times m \times d}, \mathbf{B}^\mu \in \mathbb{R}^{m \times d}, \boldsymbol{\mathcal{B}}^\sigma \in \mathbb{R}^{m \times d}$, $\boldsymbol{\mu}_i^j = (\mathbf{M}_i)_{:,j} \in \mathbb{R}^d, \boldsymbol{\Sigma}_i^j = (\boldsymbol{\Sigma}_i)_{:,j} \in \mathbb{R}^d$. diag is defined to be $\mathrm{diag}(\boldsymbol{\Sigma}_i^j) = (\boldsymbol{\Sigma}_i^j \otimes \mathbf{1}) \circ \mathbf{I}$, where $\circ$ denotes the Hadamard product, $\mathbf{1} \in \mathbb{R}^d$ is an all one vector and $\mathbf{I} \in \mathbb{R}^{d \times d}$ is an identity matrix. For simplicity, we let $d' = d$ and approximate each conditional via a mixture of Gaussian with a diagonal covariance matrix. This can be changed to a full covariance matrix, should the corresponding assumption (positive semi-definite) of the matrix is satisfied. Note this simplification does not affect the expressiveness of the model, as a GMM with a diagonal covariance matrix is also an universal approximator for densities and can approximate a GMM with a full covariance matrix [Benesty et al., 2008], given enough states. Under this definition, it is easy to show that $\prod_{i=1}^{l} f_{\tilde{A}}(\boldsymbol{x}_{\leq i})$ computes the density of the sequence $\boldsymbol{x}_{\leq l}$, where $\boldsymbol{x}_{\leq l}$ denotes $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_l$. We will refer to this NCWFA model with RNADE structure as RNADE-NCWFA of k states and m mixtures. Note although the definitions of $\boldsymbol{\beta}_i$, $\mathbf{M}_i$ and $\boldsymbol{\Sigma}_i$ takes specific forms, in practice, one can use any differentiable function of $\boldsymbol{h}_i$ to compute $\boldsymbol{\beta}_i$, $\mathbf{M}_i$ and $\boldsymbol{\Sigma}_i$, so long as $\boldsymbol{\beta}_i$ sums to one and $\boldsymbol{\Sigma}_i$ is positive.

One natural question to ask is how expressive this model is. We show in the following theorem that RNADE-NCWFA is strictly more expressive than Gaussian HMMs, which is well known for sequential modeling [Bilmes et al., 1998].

Theorem 12. *Given a Gaussian HMM with k states $\eta = \langle \boldsymbol{\mu}, \mathbf{T}, O \rangle$, where $O : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}^+$ is the Gaussian emission function, $\mu \in \mathbb{R}^k$ is the initial state distribution and $\mathbf{T} \in [0, 1]^k$ is the transition matrix, there exists a k states k mixtures RNADE-NCWFA $\tilde{A} = \langle \boldsymbol{\alpha}, \xi, \phi, \boldsymbol{\mathcal{A}} \rangle$ with full covariance matrices such that the density function over all possible trajectories generated by η can*

be computed by $\tilde{A}$: $p^{\eta}(\boldsymbol{o}_1 \cdots \boldsymbol{o}_n) = \prod_{i=1}^{n} f_{\tilde{A}}(\boldsymbol{o}_{\leq n})$ for any trajectory $\boldsymbol{o}_1 \cdots \boldsymbol{o}_n$. Moreover, there exists an RNADE-NCWFA $\tilde{A}$ such that no Gaussian HMM model can compute its density.

Proof. For the Gaussian HMM η, given an observation sequences $\boldsymbol{o}_1 \cdots \boldsymbol{o}_n$, its density under η is:

$$p^{\eta}(\boldsymbol{o}_1 \cdots \boldsymbol{o}_n) = O(\boldsymbol{m}^\top, \boldsymbol{o}_1) O(\boldsymbol{m}^\top \mathbf{T}, \boldsymbol{o}_2) \cdots O(\boldsymbol{m}^\top \mathbf{T}^{n-1}, \boldsymbol{o}_n),$$

where $O(\boldsymbol{h}, \boldsymbol{o}) = \sum_{i=1}^{k} \boldsymbol{h}_i \mathcal{N}(\boldsymbol{o} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$ for some mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. Let $\boldsymbol{\alpha} = \boldsymbol{m}$, $\mathscr{A}_{:,i,:} = \mathbf{T}$ for $i \in [k]$, $\phi(\boldsymbol{x}) = [\frac{1}{k}, \frac{1}{k}, \cdots, \frac{1}{k}]^\top$ and $\xi = O$. Note it reasonable to let $\xi = O$, since as long as we let $\boldsymbol{\beta}_i = \boldsymbol{\alpha}^\top \mathbf{T}^{i-1}$, $\boldsymbol{\beta}_0 = \boldsymbol{\alpha}^\top$, $\boldsymbol{\mu}_i = \boldsymbol{\mu}$ and $\boldsymbol{\Sigma}_i = \boldsymbol{\Sigma}$, following equations 7.2, then for any $\boldsymbol{h} \in \mathbb{R}^k, \boldsymbol{o} \in \mathbb{R}^d$, we have $\xi(\boldsymbol{h}, \boldsymbol{o}) = O(\boldsymbol{h}, \boldsymbol{o})$. Then under this parameterization, we have $\mathscr{A} \bullet_2 \phi(\boldsymbol{o}_j) = \mathbf{T}$. Then the RNADE-NCWFA computes the following function:

$$\begin{aligned} f_{\tilde{A}}(\boldsymbol{o}_1, \cdots, \boldsymbol{o}_i) &= \xi((\mathscr{A} \bullet_1 \boldsymbol{\alpha}^\top \bullet_2 \phi(\boldsymbol{o}_1))^\top (\mathscr{A} \bullet_2 \phi(\boldsymbol{o}_2)) \cdots (\mathscr{A} \bullet_2 \phi(\boldsymbol{o}_{i-1})), \boldsymbol{o}_i) \\ &= \xi(\boldsymbol{\alpha}^\top \mathbf{T}^{i-1}, \boldsymbol{o}_i) = O(\boldsymbol{m}^\top \mathbf{T}^{i-1}, \boldsymbol{o}_i) \end{aligned}$$

Therefore, we have:

$$\begin{aligned} p^{\eta}(\boldsymbol{o}_1 \cdots \boldsymbol{o}_n) &= O(\boldsymbol{m}^\top, \boldsymbol{o}_1) O(\boldsymbol{m}^\top \mathbf{T}, \boldsymbol{o}_2) \cdots O(\boldsymbol{m}^\top \mathbf{T}^{n-1}, \boldsymbol{o}_n) \\ &= f_{\tilde{A}}(\boldsymbol{o}_1) f_{\tilde{A}}(\boldsymbol{o}_1, \boldsymbol{o}_2) \cdots f_{\tilde{A}}(\boldsymbol{o}_1, \cdots, \boldsymbol{o}_n) = \prod_{i=1}^{n} f_{\tilde{A}}(\boldsymbol{o}_{\leq n}) \end{aligned}$$

For the proof of the second half of the theorem, consider a shifting Gaussian HMM, where the mean vector of the Gaussian emission is a function of the time steps, i.e., $\boldsymbol{\mu} = q(i)$, where $i = 1, 2, \cdots$. For simplicity, assume the shifting Gaussian HMM is for one dimensional sequences and has one mixture. In addition, let $q(i) = i$ and assume the variance is 1. Then the emission function can be written as $O^t(o) = \mathcal{N}(o | t, 1)$. Then the density of a sequence $o_1, \cdots, o_n$ under this shifting Gaussian HMM η_s is:

$$p^{\eta_s}(o_1, \cdots, o_n) = O^1(o_1) O^2(o_2) \cdots O^n(o_n).$$

We show that this density cannot be computed by a Gaussian HMM of finite states. If p^{η_s} can be computed by a Gaussian HMM, then for the mean vector $\boldsymbol{\mu}$ there exists an initial weight vector $\boldsymbol{m}$, a transition matrix $\mathbf{T}$ satisfying the following linear system:

$$\begin{cases} \boldsymbol{m}^\top \boldsymbol{\mu} & = 1 \\ \boldsymbol{m}^\top \mathbf{T} \boldsymbol{\mu} & = 2 \\ & \vdots \\ \boldsymbol{m}^\top \mathbf{T}^{n-1} \boldsymbol{\mu} & = n \\ & \vdots \end{cases}$$

This linear system is, however, overdetermined, as $\boldsymbol{\mu}$ is a vector of finite size, while there are infinite linearly independent equations to satisfy. Therefore, a Gaussian HMM of finite states cannot compute the density function of a shifting Gaussian HMM.

We now show such density can be computed by a RNADE-NCWFA. Let $\boldsymbol{\alpha}^\top = [1, 1]$, and $\mathscr{A}_{:,i,:} = \begin{bmatrix} 1 & 1 \\ & \\ 0 & 1 \end{bmatrix}$, $\boldsymbol{\mu}_i = \langle \boldsymbol{h}_{i-1}, [0, 1] \rangle$, $\phi(o)^\top = [0.5, 0.5]$, $\boldsymbol{\Sigma}_i = 1$. Then we have:

$$\begin{aligned} f_{\tilde{A}}(o_1, \cdots, o_i) &= \xi((\mathscr{A} \bullet_1 \boldsymbol{\alpha}^\top \bullet_2 \phi(o_1))^\top (\mathscr{A} \bullet_2 \phi(o_2)) \cdots (\mathscr{A} \bullet_2 \phi(o_{i-1})), o_i) \\ &= \xi(\boldsymbol{\alpha}^\top \mathbf{T}^{i-1}, o_i) = \xi([1, i], o_i) = \mathscr{N}(o_i | i, 1) \end{aligned}$$

Therefore:

$$\begin{aligned} p^{\eta_s}(o_1, \cdots, o_n) &= \mathscr{N}(o|1, 1)\mathscr{N}(o|1, 2) \cdots \mathscr{N}(o|1, n) \\ &= f_{\tilde{A}}(o_1) f_{\tilde{A}}(o_1, o_2) \cdots f_{\tilde{A}}(o_1, \cdots, o_n) = \prod_{i=1}^{n} f_{\tilde{A}}(\boldsymbol{o}_{\leq n}) \end{aligned}$$

Therefore, for the given shifting Gaussian HMM density, it can be computed by an RNADE-NCWFA, but cannot be computed by a Gaussian HMM with finite states. □

Note that a CWFA cannot compute the density function of a Gaussian mixture. Indeed, the function computed by a CWFA on a sequence of length 1 is linear in its input, whereas an RNADE-NCWFA associates such an input to a Gaussian density.

To learn RNADE-NCWFA, we want to maximize the likelihood given some training set $D_l = \{\boldsymbol{x}^1_{\leq l}, \cdots, \boldsymbol{x}^N_{\leq l}\}$ of length-l sequences of d dimensional vectors, i.e., $\boldsymbol{x}^n_{\leq l} = \boldsymbol{x}^n_1, \cdots, \boldsymbol{x}^n_l$, where each $\boldsymbol{x}^n_i \in \mathbb{R}^d$. More specifically, we want to minimize the negative log likelihood function: $\mathscr{L}(\tilde{A}, D) = -\sum_{i=1}^N \sum_{j=1}^l \log(f_{\tilde{A}}(\boldsymbol{x}^i_{\leq j}))$. One straightforward solution is to use gradient descent to optimize this loss function. However, as pointed out in [Bengio et al., 1994], due to repeated multiplication by the same transition tensor, gradient descent is prone to suffer from the vanishing gradient problem and to fail in capturing long-term dependencies. One alternative is the classic spectral learning algorithm for WFAs. Recall that the spectral learning method for CWFA requires first learning Hankel tensors of length L, $2L$, and $2L+1$ and then performing a rank factorization on the learned Hankel tensor to recover the CWFA parameters (see [Li et al., 2020b]). However, due to the nonlinearity added to the model, namely the feature map ϕ and the termination function ξ, spectral learning alone will not be enough. To circumvent this issue, we present an algorithm jointly leveraging gradient descent and spectral learning. The idea is to first learn the Hankel tensors of various lengths and the function ϕ and ξ using gradient descent. Then we use the spectral learning algorithm to recover the transition tensor and the initial weights.

Let δ and ω denote the parameters of the mappings ϕ and ξ, respectively (see Eq. 6.5-6.7), and let $\boldsymbol{\mathscr{H}}^{(l)}_{\tilde{A}} = [\![\boldsymbol{\mathscr{G}}^{(l)}_1, \cdots, \boldsymbol{\mathscr{G}}^{(l)}_l]\!]$ be the TT form of the Hankel tensor, where $\boldsymbol{\mathscr{G}}^{(l)}_1 \in \mathbb{R}^{d\times k}$ and $\boldsymbol{\mathscr{G}}^{(l)}_i \in \mathbb{R}^{k\times d\times k}$ for $i = 2, \cdots, l$. The spectral learning method for RNADE-NCWFAs first involves an approximation of the Hankel tensor via minimizing the following loss function:

$$\mathscr{L}(\delta, \omega, \boldsymbol{\mathscr{G}}^{(l)}_1, \cdots, \boldsymbol{\mathscr{G}}^{(l)}_l, D_l) = -\sum_{i=1}^N \sum_{j=1}^l \log\left[\xi\left(\psi(\boldsymbol{x}^i_{\leq j})^\top ([\![\boldsymbol{\mathscr{G}}^{(l)}_1, \cdots, \boldsymbol{\mathscr{G}}^{(l)}_j]\!])_{\langle\langle j,1\rangle\rangle}\right)\right] \tag{6.8}$$

where $\psi(\boldsymbol{x}_{\leq j}) = \phi(\boldsymbol{x}_1) \otimes \cdots \otimes \phi(\boldsymbol{x}_j)$. In this process, we have obtained the Hankel tensors and the parameters of the termination function and the feature map. Then, one can perform a rank factorization on the learned Hankel tensor and recover the rest of the parameters for the RNADE-NCWFA, namely $\boldsymbol{\alpha}, \mathscr{A}$. The detailed algorithm is presented in Algorithm 6.

6.4 Experiments

For the experiments, we conduct a synthetic experiment based on data generated from a random 10-states Gaussian HMM. We sample sequences of lengths 3, 6, and 7 from the HMM. To evaluate the model's performance on its generalization ability to an unseen length of sequences, we sample 1,000 sequences from length 8 to length 400 from the same HMM for the test data. To test the model's resistance to noise, we inject the training samples with Gaussian noise of different standard deviations (0.1 and 1.0) with 0 means.

For the baseline models, we have HMM learned with the expectation maximization (EM) method, as it can compute the density of sequences of any length by design. We also modified the RNADE model by replacing the hidden states update rule 6.2 with an LSTM structure to give the RNADE model the ability to generalize to sequences of arbitrary length, regardless of the length of the training sequences. We refer to this model as RNADE-LSTM.

Algorithm 6 `NCWFA-SL`: Spectral Learning of RNADE-NCWFA

Input: Three training datasets D_L, D_{2L}, D_{2L+1} with input sequences of length L, $2L$ and $2L+1$ respectively, an encoder ϕ_δ, a termination function ξ_ω and TT-parameterized Hankel tensors $\mathscr{H}_{\tilde{A}}^{(L)}$, $\mathscr{H}_{\tilde{A}}^{(2L)}$ and $\mathscr{H}_{\tilde{A}}^{(2L+1)}$, learning rate γ, desired rank R

1: **while** Model not converging **do**

2: **for** $l \in \{L, 2L, 2L+1\}$ **do**

3: Update $\delta, \omega, \mathscr{G}_1^{(l)}, \cdots, \mathscr{G}_l^{(l)}$ via gradient descent by minimizing the loss function 6.8

4: **for** $\theta \in \{\delta, \omega, \mathscr{G}_1^{(l)}, \cdots, \mathscr{G}_l^{(l)}\}$ **do**

5:

$$\theta \leftarrow \theta - \gamma \nabla_\theta \mathscr{L}(\delta, \omega, \mathscr{G}_1^{(l)}, \cdots, \mathscr{G}_l^{(l)}, D_l)$$

6: **end for**

7: **end for**

8: **end while**

9: Let $(\mathscr{H}_{\tilde{A}}^{(2L)})_{\langle\langle L, L+1 \rangle\rangle} = \mathbf{PS}$ be a rank R factorization.

10: Return the RNADE-NCWFA $\tilde{A} = \langle \boldsymbol{\alpha}, \xi_\omega, \phi_\delta, \boldsymbol{\mathcal{A}} \rangle$ where

$$\boldsymbol{\alpha} = (\mathbf{S}^\dagger)^\top (\mathscr{H}_{\tilde{A}}^{(L)})_{\langle\langle L+1 \rangle\rangle}, \qquad \boldsymbol{\mathcal{A}} = ((\mathscr{H}_{\tilde{A}}^{(2L+1)})_{\langle\langle L, 1, L+1 \rangle\rangle}) \times_1 \mathbf{P}^\dagger \times_3 (\mathbf{S}^\dagger)^\top$$

For our model, by following Algorithm 6, we have the method RNADE-NCWFA (spec). Alternatively, although we have mentioned that training the (RNADE-)NCWFA model through pure gradient descent method can have many issues, we also list this approach of training RNADE-NCWFA as one of the baselines, referred to as RNADE-NCWFA (sgd). For all the training processes, if gradient descent is involved, we always use Adam optimizer [Kingma and Ba, 2014] with 0.001 learning rate with early stopping. For HMM as well as RNADE-NCWFA models, we set the size of the model to be 10 (ground truth of the random HMM). For RNADE-LSTM, we set the size of the hidden

states to be 10. We present the trend of the averaged log likelihood ratio with the ground truth likelihood, i.e. $\log(\frac{\text{predicted likelihood}}{\text{ground truth}})$ w.r.t. length of the sequences over 10 seeds in Figure 6.1 and a snapshot of the log likelihood for each model of 400 length testing sequences in Table 6.1.

Table 6.1: Comparison of model performance in terms of average log likelihood (in NAT). Different models are compared under different training sizes and levels of noise injected. The reported likelihood (mean (standard deviation)) is evaluated on test sequence of length 400.

Training Size	100			500			1000		
Noise Std	0	0.1	1	0	0.1	1	0	0.1	1
HMM (EM)	-615.26 (2.57)	-616.88 (3.40)	-638.44 (7.50)	-601.15 (0.20)	-601.18 (0.17)	-628.75 (1.51)	-600.70 (0.12)	-600.69 (0.12)	-628.91 (1.13)
RNADE-LSTM	-604.71 (3.36)	-604.10 (2.29)	-641.72 (7.17)	-600.86 (0.15)	-600.85 (0.23)	-628.28 (3.56)	-601.01 (1.34)	-600.67 (0.24)	-628.45 (1.56)
RNADE-NCWFA (spec)	**-600.96 (0.42)**	**-601.06 (0.24)**	**-621.75 (2.71)**	**-600.73 (0.18)**	**-600.67 (0.067)**	**-622.10 (1.44)**	**-600.50 (0.49)**	-600.53 (0.07)	**-621.91 (1.13)**
RNADE-NCWFA (sgd)	-604.11 (2.13)	-603.29 (1.69)	-633.96 (14.6)	-600.81 (0.20)	-600.91 (0.46)	-631.80 (5.53)	-600.51 (0.06)	**-600.52 (0.08)**	-629.11 (1.65)
Ground Truth	-600.40	-600.40	-600.40	-600.40	-600.40	-600.40	-600.40	-600.40	-600.40

From the experiment results, we can see that RNADE-CWFA (spec) consistently has the best performance across all training sizes and levels of noise injected. More precisely, this advantage is more significant when given small training sizes and (or) the data is injected with high noise. Moreover, the spectral learning algorithm shows stable training results as the standard deviation of the log likelihood (ratio) is the lowest among all methods. This is especially the case when not enough training samples are provided. In addition, one can see that this advantage is consistent with all test sequence lengths we have experimented.

6.5 Conclusion and Future Work

In this chapter, we propose the RNADE-NCWFA model, an expressive and tractable WFA-based density estimator over sequences of continuous vectors. We extend the notion of continuous WFA to its nonlinear case by introducing a nonlinear feature mapping function as well as a nonlinear termination function. We then combine the idea from RNADE to propose our density estimation model RNADE-NCWFA and its spectral learning based learning algorithm. In addition, we show that theoretically, RNADE-NCWFA

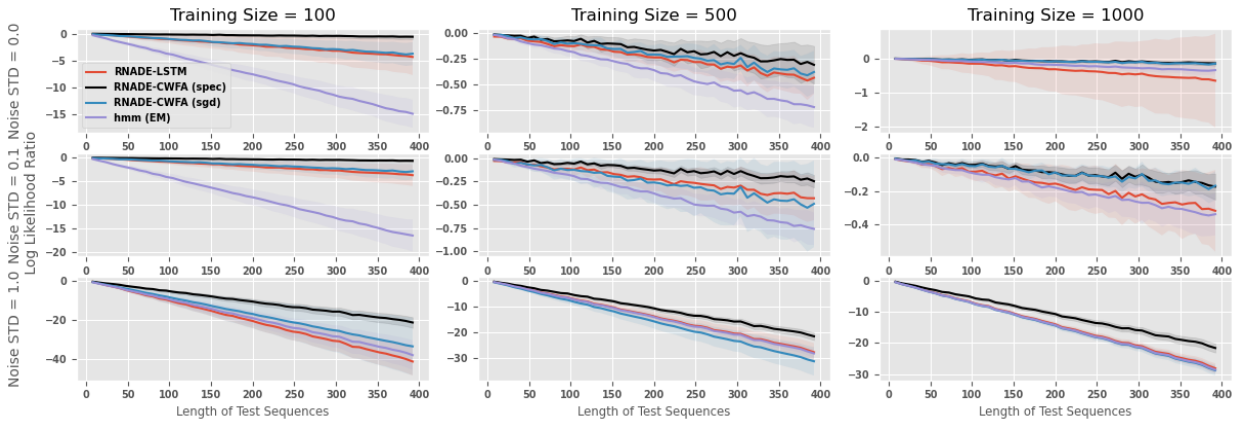


Figure 6.1: Log likelihood ratios between the tested models and the ground truth likelihood. We show the trend w.r.t. the length of the testing sequences under different sample sizes (columns) and standard deviations of the injected noise (rows).

is strictly more expressive than the Gaussian HMM model. We show that, empirically, our method has great capability of generalizing to sequences of varying length, which is potentially not the same as the training sequences. For future work, we are looking into more experiments on real datasets and comparing them with more baselines. Moreover, we did not add a nonlinear transition for the NCWFA model as it would imply that the Hankel tensor will be of infinite tensor train rank, hence making the spectral learning algorithm intractable. We will be looking into the possibilities of adding this nonlinearity into the NCWFA model and have a working algorithm for it. In addition, we would like to examine more closely in terms of the expressivity of the RNADE-NCWFA.

Chapter 7

Recurrent Real-valued Neural Autoregressive Density Estimator for Online Density Estimation and Classification of Streaming Data

In the previous chapter, we introduced the RNADE-NCWFA, a nonlinear extension of WFAs in the continuous domain that combines the NCWFA model with RNADE for sequence density estimation. In this chapter, we expand on this idea by replacing the NCWFA model with a variety of recurrent neural networks. Moreover, we shift our focus to a more practical task than the previous ones: online density estimation and classification for data streams.

Data streams have become increasingly popular in recent machine learning advancements, and they are characterized by being long-lasting, non-stationary, and subject to underlying distribution changes over time. To model such data with SSMs, the state representation needs to adapt to these changes in a timely fashion. To address this challenge, we propose the recurrent real-valued neural density estimator (RRNADE), a recurrent real-valued neural autoregressive density estimator that leverages RNNs and RNADE.

The RRNADE is a flexible method for online modeling of data streams that can accommodate per-sample updates and adapt to changes in the data stream distribution, providing an efficient state representation for these tasks.

7.1 Introduction

Many tasks in classic supervised machine learning, such as regression and classification, involve processing batched data in an offline fashion: the data, often coming as input-output pairs, is stored first and then used to learn a predictive model for future unseen data. However, many modern applications favor the form where the model update and predict while receiving new data entries. This form is often referred to as learning from data streams. The problem of learning from data streams is closely related to the problem of continual or incremental learning [Losing et al., 2018, Zenke et al., 2017, Lopez-Paz and Ranzato, 2017], which have recently received increasing interest in the machine learning community

There are three major issues when learning from data streams: memory constraints, concept drifts as well as temporal correlations. The sheer amount of data many modern applications process daily makes it infeasible to store all data and perform offline updates of the model [Naeem et al., 2022]. In addition, certain data sources do not allow the indefinite hold of the data due to potential privacy regulations [Forti, 2021]. Therefore, when learning data streams, it is often assumed that the model only has access to the recent history. Furthermore, concept drifts and temporal correlations are also common challenges when learning from data streams. Under the offline setting, data is often assumed to have the i.i.d. assumption, i.e. each data entry is independently drawn from the identical distribution. However, under the streaming data setting, the independent assumption can be violated, causing temporal correlations in the data, while the violation of the identical assumption can lead to the concept drift problem, which often refers to changes in the

underlying distribution. These issues often invalidate the model learned from historical data, resulting in further deterioration of its performance.

Density estimation is one of the core tasks in the field of unsupervised learning, branching out to many applications such as classification and clustering. Under the offline setting, Real-valued Neural Autoregressive Density Estimator (RNADE) leverages a neural network parameterized Gaussian mixture model to estimate the density function of real-valued vectors. It is then interesting to explore the possibility of extending RNADE to its online form, namely, the model needs to be updated as new data arrives and we only have a limited amount of history stored in memory. In this chapter, we show that the answer is in the positive. Concretely, our contributions are as follows:

1. We propose the Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE), a versatile density estimator for online learning of data streams.

2. Moreover, we propose an RRNADE based Bayes classifier for the online classification of streaming data.

Our model uses a recurrent module to maintain a set of sufficient statistics for the future and capture the potential temporal properties of the data. In addition, it also uses a neural network parameterized Gaussian mixture model as the density module to compute the conditional density function of the current input given the previous data. We theoretically show that RRNADE is strictly more expressive than Gaussian hidden Markov models [Bilmes et al., 1998]. We present empirical results demonstrating the ability of RRNADE to adapt to concept drifts and approximate density functions with sequential relations. Moreover, we conduct extensive experiments on various benchmarks of online classification and show that RRNADE outperforms all the compared methods on almost every dataset. In addition, we further demonstrate the importance of both the recurrent module and the density module in the ablation study.

Related Works For online density estimation on streaming data, many of the existing works focus on the adoption of the kernel density estimation (KDE) method [Procopiuc

and Procopiuc, 2005, Heinz and Seeger, 2008, Kristan et al., 2011, Boedihardjo et al., 2008]. These estimators often rely on maintaining and updating (though merging) a specific number of kernels while incorporating new instances, while in different fashions. In addition to these methods, KDE-Track [Qahtan et al., 2016] leverages an adaptive resampling strategy to deal with concept drifts and improve the estimation accuracy of the KDE-based methods. Another recent method, adaptive local online kernel density estimator (ALoKDE) [Chen et al., 2021], leverages a statistical test for concept drift detection to adapt fast to the concept drift. All these methods can be modified to a classification method via a Bayes classifier.

For online classification on streaming data, there are a number of methods that are direct adaptations of the original offline version to its online case. For example, the online SVM (OSVM) [Li and Yu, 2015], the adaptive random forest (ARF) [Gomes et al., 2017a], can be categorized to this type of methods. In addition, [Liang et al., 2006, Cauwenberghs and Poggio, 2000, Lu et al., 2014] also belong to this class of methods. Other methods like [Bifet and Gavalda, 2007, Bifet et al., 2013] leverage an adaptive window size of the past, [Losing et al., 2016] takes advantage of the short-term and long-term memories, while [Gomes et al., 2017b, Polikar et al., 2001] use ensemble method to further improve the results. Another large class of online classification methods is the prototype-based classifiers, such as incremental learning vector quantization (ILVQ) [Losing et al., 2015], generalized LVQ [Sato and Yamada, 1995], robust soft LVQ [Heusinger et al., 2019], and the sparse prototype online kernel density estimator (SPOK) [Coelho and Barreto, 2022].

7.2 Online learning for streaming data

In this chapter, we focus on online density estimation and classification for streaming data. Formally, for the density estimation task, let $S = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \cdots\}$ be a sequential data stream governed by some distribution $f_t(\cdot)$, where $\boldsymbol{x}_t \sim f_t$ and the subscript denotes the timestamp of the data entry, and $f_t : \mathbb{R}^d \to \mathbb{R}_0^+$ denotes the distribution at time step t.

We are then interested in finding an accurate approximation of the density function f at each time step. For a C classes classification task, let $y_t \in \{1, \cdots, C\}$ be the label at time t and $SL = \{(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n), \cdots\}$ be a sequence of input label pairs. Moreover, the label y_t is drawn from the distribution q_t while $\boldsymbol{x}_t$ is drawn from the distribution $f_t^{y_t}$, i.e. $\boldsymbol{x}_t \sim f_t^{y_t}$ where $f_t^{y_t}(\boldsymbol{x}_t) = p(\boldsymbol{x}_t|y_t)$ denotes the input distribution of class y_t at time step t. In this setting, we are interested in predicting the correct label at each time step t. As we are approaching these tasks in the streaming setting, it is infeasible to store all the data one have seen so far. Therefore, for both of these tasks, we constrain ourselves to only have access to a short window of data at each time step, e.g. $\boldsymbol{x}_{t-l}, \cdots, \boldsymbol{x}_t$, where l is the window size and controlled to be a relatively small number.

One of the most challenging problems in learning from data streams is concept drift. Concept drift occurs when the underlying distribution f_t changes over time. This change can be abrupt, happening when data changes significantly and occasionally [Iwashita and Papa, 2018]. Alternatively, distribution shifts can occur gradually, when the data values slowly but constantly change over time. For classification tasks, concept drifts can occur not only in $f_t^{y_t}$ but also in q_t. For example, in video frames classification, the goal is to classify different objects that appear in the current frame. In this case, a shift in the camera angle will result in a concept drift in the label's distribution, i.e., q_t. This shift can occur abruptly (due to a sudden movement of the camera) or gradually (due to a steady movement of the camera), resulting in different types of concept drift.

7.3 Methodology

In this section, we introduce the Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE): a versatile model for density estimation and classification of stream data.

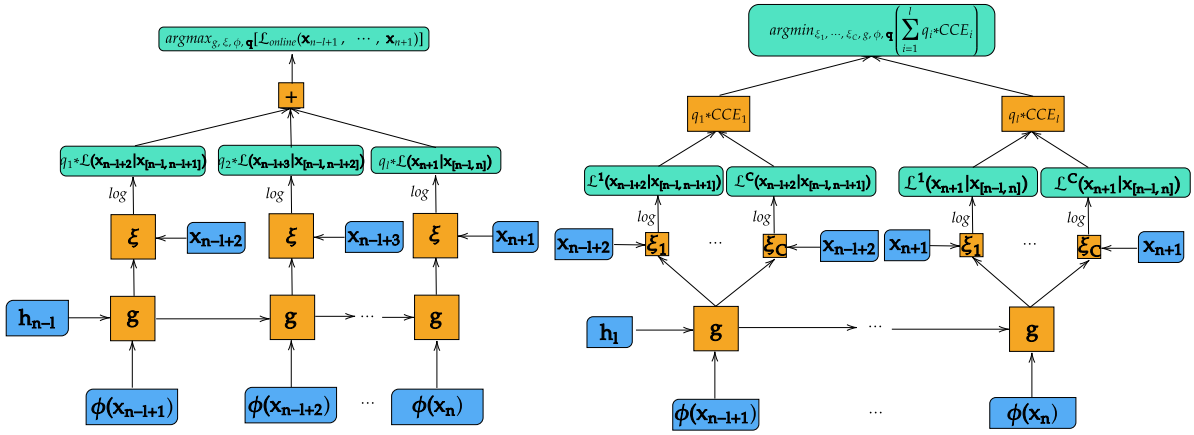


Figure 7.1: The Recurrent Real-valued Neural Autoregressive Density Estimator for online density estimation (left) and online classification (right), where the window size is l. Blue, orange and green boxes denote input data, functions and outputs, respectively.

7.3.1 Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE) for Online Density Estimation

It is natural to wonder if it is possible to use RNADE for *online* density estimation. There are two major issues for this adaptation: 1) the original RNADE model has one set of parameters per feature, which would lead to an infinite amount of parameters to estimate for indefinite lengths of the stream, and 2) the offline stochastic gradient descent routine needs to be adjusted to its online setting.

To tackle the first problem, instead of approximating the conditionals $p(\boldsymbol{x}_i|\boldsymbol{x}_{<i})$ via the classic RNADE treatment (see Eq. 6.2 and Eq. 6.3), we use a recurrent model $R = \langle g, \xi, \boldsymbol{h}_0 \rangle$ to model the conditionals: $p(\boldsymbol{x}_i|\boldsymbol{x}_{<i}) = f_R(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_i)$. In contrast with RNADE, using a recurrent model allows us to make the state update function independent of the time step, enabling RRNADE to generalize to sequences of arbitrary lengths. Inspired by RNADE, we constrain the output of the recurrent model to be a mixture of Gaussians with diagonal covariance matrices. We now formally introduce the Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE) model:

Definition 24. *A Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE) with k states and m components is a tuple $\mathscr{R} = \langle g, \phi, \boldsymbol{h}_0, \xi \rangle$, where $\boldsymbol{h}_0 \in \mathbb{R}^k$ is the initial state, $g : \mathbb{R}^k \times \mathbb{R}^{d'} \to \mathbb{R}^k$ is the recurrent module, $\xi : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}_0^+$ is the density module and*

$\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is the input encoder. An RRNADE computes a function $f_{\mathscr{R}} : (\mathbb{R}^d)^* \to \mathbb{R}_0^{+*}$. Given a sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$, $f_{\mathscr{R}}$ computes in the following fashion:

$$\boldsymbol{h}_i = g(\boldsymbol{h}_{i-1}, \phi(\boldsymbol{x}_i)), \qquad \boldsymbol{\beta}_i = \operatorname{softmax}(\mathbf{V}^\beta \boldsymbol{h}_{i-1} + \boldsymbol{b}^\beta) \tag{7.1}$$

$$\mathbf{M}_i = \boldsymbol{\mathcal{V}}^\mu \bullet_1 \boldsymbol{h}_{i-1} + \mathbf{B}^\mu, \quad \boldsymbol{\Sigma}_i = \exp(\boldsymbol{\mathcal{V}}^\sigma \bullet_1 \boldsymbol{h}_{i-1} + \boldsymbol{\mathcal{B}}^\sigma) \tag{7.2}$$

$$\xi(\boldsymbol{h}_{i-1}, \boldsymbol{x}_i) = \sum_{j=1}^{m} \boldsymbol{\beta}_i^j \mathcal{N}(\boldsymbol{x}_i | \mathbf{M}_i^j, \operatorname{diag}(\boldsymbol{\Sigma}_i^j)), \quad f_{\mathscr{R}}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \xi(\boldsymbol{h}_{n-1}, \boldsymbol{x}_n) \tag{7.3}$$

where $\boldsymbol{\mathcal{V}}^\mu \in \mathbb{R}^{k \times m \times d}, \boldsymbol{\mathcal{V}}^\sigma \in \mathbb{R}^{k \times m \times d}, \mathbf{B}^\mu \in \mathbb{R}^{m \times d}, \mathbf{B}^\sigma \in \mathbb{R}^{m \times d}$, $\mathbf{V}^\beta \in \mathbb{R}^{m \times k}$, $\boldsymbol{b}^\beta \in \mathbb{R}^m$, $\mathbf{M}_i^j = (\mathbf{M}_i)_{j,:} \in \mathbb{R}^d, \boldsymbol{\Sigma}_i^j = (\boldsymbol{\Sigma}_i)_{j,:} \in \mathbb{R}^d$, $\operatorname{diag}(\boldsymbol{\Sigma}_i^j)$ denotes the diagonal matrix having the components of $\boldsymbol{\Sigma}_i^j$ on the diagonal and $\boldsymbol{\mathcal{V}} \bullet_1 \boldsymbol{h}$ denotes the mode-1 product defined by $(\boldsymbol{\mathcal{V}} \bullet_1 \boldsymbol{h})_{j,k} = \sum_i \boldsymbol{\mathcal{V}}_{i,j,k} \boldsymbol{h}_i$.

For the simplicity of later sections, we denote $\mathscr{R}_{(\boldsymbol{h},\xi)}$ by the RRNADE with the initial vector $\boldsymbol{h}$ and the termination function ξ. More specifically, the notation $\boldsymbol{h}_t$ denotes the hidden state at time step t after observing the sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_t$ under the updating rule 7.1 of RRNADE. As outlined by Equation 7.3, RRNADE models the conditional density of the current input data, given the history, as a Gaussian mixture. Similar to the previous chapter, we let $d' = d$ and approximate each conditional via a mixture of Gaussian with a diagonal covariance matrix. Same as before, this can be changed to a full covariance matrix under appropriate assumptions and is still an universal approximator for density functions. Denote the sequence $\boldsymbol{x}_i, \cdots, \boldsymbol{x}_j$ by $\boldsymbol{x}_{[i,j]}$ with a special case $\boldsymbol{x}_{[i,i]} = \boldsymbol{x}_i$ and $j \geq i \in \mathbb{Z}^+$. With our aforementioned notations, $\Pi_{i=1}^{l} f_{\mathscr{R}_{(\boldsymbol{h}_t)}}(\boldsymbol{x}_{[t+1,t+i]})$ computes $p(\boldsymbol{x}_{t+1}, \cdots, \boldsymbol{x}_{t+l} | \boldsymbol{x}_{\leq t})$

We show in the following theorem that RRNADE is strictly more expressive than Gaussian HMMs, which are well known for sequential modeling [Bilmes et al., 1998].

Theorem 13. *Given a Gaussian HMM η with k states, there exists a k states k mixtures RRNADE $\mathscr{R}$ with full covariance matrices such that the density function over all possible trajectories sam-*

*$(\mathbb{R}^d)^*$ denotes the set of all possible sequences of arbitrary length constructed with d dimensional real-valued vector at each time step.

pled by η can be computed by $\mathscr{R}$: $p^{\eta}(\boldsymbol{x}_1 \cdots \boldsymbol{x}_n) = \prod_{i=1}^{n} f_{\mathscr{R}}(\boldsymbol{x}_{\leq n})$ for any trajectory $\boldsymbol{x}_1 \cdots \boldsymbol{x}_n$. Moreover, there exists an RRNADE $\mathscr{R}'$ such that no Gaussian HMM can compute its density.

Note that NCWFA is a special case of RRNADE, therefore the proof follows the same as the one of Theorem 12. For the second problem, by design, RRNADE approximates a conditional density function of the current input given the history. In the offline setting, one can learn such RRNADE via gradient ascent of the likelihood of the sequence, i.e, maximizing the likelihood function: $L_{\text{offline}}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = p(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \Pi_{i=1}^{n} f_{\mathscr{R}}(\boldsymbol{x}_{\leq i})$. In the online setting, we assume that we have access to the past data entries from a window of size l, i.e., $\boldsymbol{x}_{t-l}, \cdots, \boldsymbol{x}_t$, at each time step t. One solution would be to maximize the likelihood over the entire window, i.e. $L(\boldsymbol{x}_{t+1}, \cdots, \boldsymbol{x}_{t+l} | \boldsymbol{x}_{\leq t}) = \Pi_{i=1}^{l} f_{\mathscr{R}_{(\boldsymbol{h}_t)}}(\boldsymbol{x}_{[t+1,t+i]})$.

However, this procedure falls short when concept drift occurs. For example, if the data has abrupt drifts (infrequent), then choosing l could be a dilemma: to capture and adapt to this concept drift fast, l is preferable to be small to provide a significant gradient update, which, however, prevents the model from learning temporal dependencies in the data. To address this issue, we propose to optimize for the weighted sum of the loglikelihood, where the weights are adjusted after each time step with the gradient update. This way, not only can we capture temporal dependencies up to the window size l, but the weighted sum enables us to adjust how fast RRNADE adapts to potential concept drifts in the data. Formally, we maximize the following likelihood function:

$$L_{\text{online}}(\boldsymbol{x}_{t+1}, \cdots, \boldsymbol{x}_{t+l} | \boldsymbol{x}_{\leq t}) = \Pi_{i=1}^{l} f_{\mathscr{R}_{(\boldsymbol{h}_t)}}(\boldsymbol{x}_{[t+1,t+i]})^{q_i}, \tag{7.4}$$

where $-1 \leq q_i \leq 1$ are the weights. The core idea of this procedure is to obtain and maintain a set of sufficient statistics for the future. At the time $t = l$, the model starts with the hidden state $\boldsymbol{h}_0$. After updating the parameters by maximizing the likelihood $\Pi_{i=1}^{l} f_{\mathscr{R}(\boldsymbol{h}_0)}(\boldsymbol{x}_i)^{q_i}$, the internal state of RRNADE is then updated to $\boldsymbol{h}_1 = g(\boldsymbol{h}_0, \boldsymbol{x}_1)$. At the next time step, $t = l + 1$, the first observation $\boldsymbol{x}_1$ is discarded but $\boldsymbol{h}_1$ still represents sufficient statistics of *all* past observations, including $\boldsymbol{x}_1$. The parameters are then optimized to

maximize the likelihood $\Pi_{i=2}^{l+1} f_{\mathscr{R}(\boldsymbol{h}_1)}(\boldsymbol{x}_{[2,i]})^{q_i}$. This process then iterates over the future time steps, carrying the sufficient statistics forward (i.e., at a future time step t, even though only the past l observations are stored in memory, $\boldsymbol{h}_{t-l}$ captures sufficient statistics of all past observations). Note we only have l number of scalar q_i to optimize for. These weights are shared across all the time steps.

The training procedure is detailed in Algorithm 7, and a graphical illustration is presented on the left side of Figure 7.1. Note that except the weights assigned to each conditional, all the parameters are shared for each time step, therefore, the space complexity is $\mathcal{O}(kmd + l)$ where l is the desired window size. We use L and $\mathscr{L}$ to denote the likelihood function and the log likelihood function, respectively. In our experiments, we select l using validation over the first n data points. For practical online learning, this can be done by training several RRNADE models in parallel and selecting the one with the best overall performance after training and predicting the validation sequence.

7.3.2 RRNADE for Online Classification

One straightforward application of approximating densities is online classification. Recall that RRNADE approximates the conditional density of the current input given the history, i.e., $f_{\mathscr{R}}(\boldsymbol{x}_{[t-l,t]}) \simeq p(\boldsymbol{x}_t|\boldsymbol{x}_{<t})$. For the online classification problem, we are interested in the conditional probability of the current label given the history, i.e. $p(y_t|\boldsymbol{x}_1 \cdots \boldsymbol{x}_t)$. Using Bayes rule: $p(y_t|\boldsymbol{x}_1 \cdots \boldsymbol{x}_t) \propto p(y_t)p(\boldsymbol{x}_t|\boldsymbol{x}_{<t}, y_t) \simeq p(y_t) f_{\mathscr{R}}(\boldsymbol{x}_{[t-l,t]})$ One approach would be to train C different RRNADE models, one for each class: $f_{\mathscr{R}^c}(\boldsymbol{x}_{[t-l,t]}) \simeq p(\boldsymbol{x}_t|\boldsymbol{x}_{<t}, y_t = c)$, where $\mathscr{R}^c$ denotes the RRNADE model for class c. However, as we are learning online, each gradient update is often of high variance. This approach introduces too many model parameters, which will further increase the model's variance, resulting in a suboptimal performance. To reduce the number of parameters, we propose to share the recurrent module of all RRNADE models, while keeping the density module specific to each class.

RRNADE's prediction at time t is thus given by

$$\hat{y}_t = \arg\max_c [p(y_t = c) f_{\mathscr{R}^c}(\boldsymbol{x}_{[t-l,t]})] \tag{7.5}$$

For the choice of the prior distribution $p(y_t)$, we recommend using a uniform distribution as extra effort needs to be taken to mitigate the shift in the true prior distribution caused by concept drifts in the data. We defer the study of estimating proper prior for data streams with concept drifts to the future work. We present RRNADE for online classification in Algorithm 7 and a graphical illustration of the model is presented on the right side of Figure 7.1.

7.4 Experiments

In this section, we present empirical results on RRNADE for online density estimation and classification. For density estimation, we experiment with synthetic data to evaluate RRNADE's ability of adapting to concept drifts and to verify Theorem 13. For classification, we conduct experiments on both synthetic as well as real world streaming data and compare them with multiple density based and non-density based online classification methods. Finally, we show an ablation study to further showcase the significance of both the density module and the recurrent module of the RRNADE. We experimented with three different variants of the RRNADE model. By using LSTM, GRU, and 2RNN for the recurrent module, we have RRNADE-LSTM, RRNADE-GRU, and RRNADE-2RNN, respectively. In all experiments, we use Adam optimizer [Kingma and Ba, 2014].

Algorithm 7 RRNADE for Online Density Estimation and Classification

INPUT: Input data stream $S = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \cdots\}$ for density estimation, or $SL = \{(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n), \cdots\}$ for C classes classification; window size l; a randomly initialized RRNADE: $\mathscr{R}(\boldsymbol{h}_0, \xi_1)$; for classification, extra $C-1$ density modules $\xi_2, \cdots, \xi_C$.

for $t = l, l+1, \cdots, n, \cdots$ **do**

  Compute the log likelihood for each input (and for each class):

$$\mathscr{L}_j^c(\boldsymbol{x}_j|\boldsymbol{x}_{[t-l,j-1]}) = \log(f_{\mathscr{R}(\boldsymbol{h}_{t-l},\xi_c)}(\boldsymbol{x}_{[t-l,j]})) \text{ for } j = t-p+1, \cdots, t$$

  if running density estimation task **then**

    Compute the weighted sum of the log likelihood:

$$\mathscr{L}_{online}(\boldsymbol{x}_{t-l+1}, \cdots, \boldsymbol{x}_t) = \sum_{i=t-l+1}^{t} q_i \mathscr{L}_i^1(\boldsymbol{x}_i|\boldsymbol{x}_{[t-l,i-1]})$$

    Perform gradient ascent update to $\mathscr{R}(\boldsymbol{h}_{t-l}, \xi_1)$, w.r.t. $\mathscr{L}_{online}$.

    Obtain the conditional density estimation at time step t:

$$p(\boldsymbol{x}_t|\boldsymbol{x}_{<t}) \simeq f_{\mathscr{R}(\boldsymbol{h}_{t-l})}(\boldsymbol{x}_{t-l+1}, \cdots, \boldsymbol{x}_t)$$

  else if running classification task **then**

    Compute the predicted class distribution:

$$P_C^j(y_t = c) = \frac{\exp(\mathscr{L}_j^c)}{\sum_{i=1}^{C} \exp(\mathscr{L}_j^i)}$$

    Obtain the predicted label at the current time step:

$$\hat{y}_t = \arg\max_c P_C^t(y_t = c)$$

    Perform gradient descent update to $\mathscr{R}(\boldsymbol{h}_{t-l}, \xi_1), \xi_2, \cdots, \xi_C$ w.r.t. the categorical cross entropy:

$$\frac{1}{l} \sum_{j=t-l+1}^{t} q_j \text{CCE}(P_C^j, y_t)$$

  end if

  Update $\boldsymbol{h}_{t-l}$ to $\boldsymbol{h}_{t-l+1}$ via the transition function g of $\mathscr{R}(\boldsymbol{h}_{t-l})$:

$$\boldsymbol{h}_{t-l+1} = g(\boldsymbol{h}_{t-l}, \boldsymbol{x}_{t-l+1})$$

end for

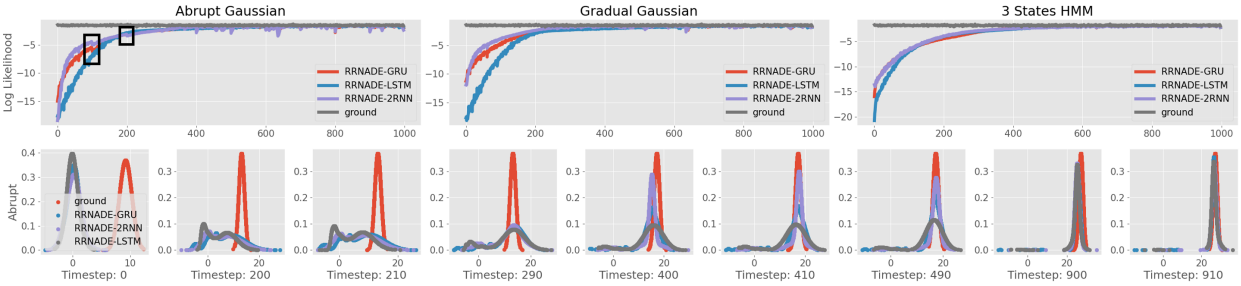


Figure 7.2: Top row (left to right): Log likelihood of (1a) Gaussian with abrupt drift. (1b) Gaussian with gradual drift. (1c) Gaussian HMM. **Bottom row**: Snapshots of learned density at corresponding time steps of Gaussian with abrupt drift, the red curve represents the ground truth density function.

Table 7.1: Averaged online AUC score of RRNADE over 5 seeds (standard deviation in the brackets), compared with [Chen et al., 2021]

AUC	ALoKDE	oKDE	odKDE	LAIM	KDE-Track	RRNADE-2RNN	RRNADE-LSTM	RRNADE-GRU	RNADE
Sea	83.11	78.74	75.13	77.17	81.29	88.12 (1.9)	87.42 (3.6)	**88.88 (2.6)**	83.31 (5.1)
Hyperplane	83.44	82.36	82.36	75.56	82.8	93.12 (2.9)	93.14 (4.2)	**94.32 (3.1)**	88.45 (3.8)
Mixed drift	90.12	88.16	70.43	55.65	88.48	**98.13 (1.2)**	97.66 (1.1)	96.11 (1.9)	83.4 (6.5)
Transient chessboard	79.6	77.4	77.08	77.18	77.98	89.16 (3.6)	86.39 (2.8)	**91.34 (3.2)**	79.21 (5.4)
Weather	76.81	68.43	66.26	73.47	76.1	85.23 (6.4)	**86.67 (3.3)**	84.14 (2.2)	71.42 (7.3)
Electricity	52.51	51.57	42.45	43.25	44.06	**92.14 (4.5)**	90.68 (5.3)	91.01 (6.7)	61.53 (6.8)
Cover type	97.31	57.86	96.67	93.04	96.04	**98.01 (1.1)**	96.13 (1.8)	90.79 (2.7)	57.13 (6.7)
Poker hand	91.01	88.36	82.92	82.19	91.01	94.39 (2.8)	**96.24 (3.4)**	93.15 (2.9)	80.31 (5.6)
Rialto	92.67	70.55	87.71	83.49	82.37	**99.11 (0.9)**	98.13 (1.5)	94.35 (0.8)	70.21 (6.9)

7.4.1 Density Estimation

To evaluate the performance on density estimation, we first conduct experiments on learning drifting Gaussian to examine RRNADE's ability of adapting to concept drifts. We generate samples from a shifting Gaussian with random initial mean and variance 1. Recall there are two major types of concept drifts, abrupt and gradual drift. For the abrupt drift, the mean is increased by 2 every 100 time steps, while for the gradual drift, the mean is increased by 0.01 every time step. For the model hyperparameters of RRNADE, we set the number of components to 10, the window size is set to 1, the number of hidden states of the recurrent module is set to 5 and we use a one layer fully connected neural network with 5 neurons to be the input encoder ϕ.

The results are displayed in Figure 7.2. In Figure 7.2 (1a) and (1b), we show the log likelihood of the Gaussian density function with abrupt and gradual drift on its mean. From the figures, we can see that all three different variants of RRNADE are able to learn

Table 7.2: Averaged online accuracy of RRNADE over 5 seeds with standard deviations, compared with [Losing et al., 2018]

Accuracy	ISVM	LASVM	ILVQ	SGD	NB	RRNADE-2RNN	RRNADE-LSTM	RRNADE-GRU	RNADE
Electricity	0.843(0.02)	0.771 (0.03)	0.732(0.02)	0.826(0.02)	0.613(0.02)	0.864 (0.02)	**0.878(0.01)**	0.858 (0.01)	0.615 (0.05)
Inter RBF	0.754(0.03)	0.512 (0.03)	0.772(0.02)	0.415(0.03)	0.301(0.01)	0.901 (0.02)	0.922 (0.02)	**0.924(0.03)**	0.452 (0.06)
Moving RBF	0.699(0.02)	0.348 (0.02)	**0.771(0.01)**	0.415(0.02)	0.182(0.01)	0.744 (0.01)	0.739 (0.02)	0.772(0.01)	0.301 (0.08)
Cover Type	0.936(0.01)	0.886 (0.03)	0.876(0.02)	0.942(0.01)	0.552(0.01)	**0.960(0.01)**	0.929 (0.02)	0.951 (0.02)	0.736 (0.07)
Border	**0.982(0.01)**	0.976 (0.03)	0.927(0.03)	0.358(0.03)	0.935(0.01)	0.955 (0.05)	0.957 (0.02)	0.943 (0.03)	0.711 (0.05)
Overlap	**0.820(0.01)**	0.788 (0.03)	0.810(0.01)	0.685(0.02)	0.682(0.01)	0.754 (0.01)	0.734 (0.01)	0.770 (0.01)	0.475 (0.03)
Outdoor	0.843(0.02)	0.823 (0.02)	0.832(0.02)	0.18(0.01)	0.652(0.01)	0.958 (0.03)	**0.963(0.02)**	0.942 (0.02)	0.703 (0.08)
COIL	0.755(0.01)	0.663 (0.02)	0.795(0.01)	0.091(0.00)	0.711(0.02)	0.851(0.03)	**0.859 (0.01)**	0.832 (0.02)	0.647 (0.04)

Table 7.3: Averaged online accuracy of RRNADE over 5 seeds with standard deviations, compared with [Coelho and Barreto, 2022]

Accuracy	L++.NSE	DACC	LVGB	KNNs	KNNwa	SAM	SPOK	RRNADE -2RNN	RRNADE -LSTM	RRNADE -GRU	RNADE
CoverType	0.850	0.899	0.909	0.958	0.932	0.952	0.883	**0.960 (0.01)**	0.929 (0.02)	0.951 (0.02)	0.736 (0.07)
Electricity	0.728	0.831	0.832	0.713	0.739	0.825	0.742	0.864 (0.02)	**0.878 (0.01)**	0.858 (0.01)	0.615 (0.05)
Outdoor	0.422	0.644	0.601	0.86	0.837	0.888	0.810	0.958 (0.03)	**0.963 (0.02)**	0.942 (0.02)	0.703 (0.08)
Poker Hand	0.779	0.790	**0.864**	0.829	0.721	0.816	0.731	0.847 (0.02)	0.851 (0.02)	0.840 (0.01)	0.707 (0.05)
Rialto	0.596	0.711	0.604	0.772	0.750	0.814	0.618	0.887 (0.05)	**0.936 (0.04)**	0.915 (0.04)	49.66 (0.02)
Weather	0.771	0.732	0.781	0.785	0.769	0.783	0.741	0.786 (0.00)	**0.790 (0.01)**	0.783 (0.01)	70.12 (0.03)

the density function and adapt to both of these drifts. The black boxes in (1a) indicate the first two abrupt drifts (time steps 100 and 200) of the Gaussian distribution, where visible drops of model performance are observed. Moreover, the adaptation speed increases w.r.t. the time step. The bottom row of Figure 7.2 shows the learned Gaussian at various time steps. We observe that, at time step 210 the mean of the mixture model has not been correctly adjusted after 10 time steps of adaptation, while at 410, the model has already adapted to the drift that occurred at time step 400.

To verify Theorem 13 and show RRNADE can approximate density functions of data streams with sequential features. We generate 1,000 examples from a random Gaussian HMM of 3 states. In this experiment, we set the hyperparameters to be the same as in the above experiment except for $l = 5$. In Figure 7.2 (1c), we show the learning curves w.r.t. log likelihood on all three variants. Here we can see all three variants are able to approximate the density function that the HMM emits at each time step. Note there is also a gradual concept drift with the HMM data, as the density function at each time is a mixture of a set of Gaussians, where the mixing weights are the state distributions that the HMM maintains.

7.4.2 Classification

In this subsection, we present experimental results for online classification on various classic benchmarks of online classification and compare RRNADE to both density-based and non-density-based methods. The results of these methods are obtained from the corresponding papers, and to ensure fairness, we conduct data preprocessing and evaluation procedures in the same way as mentioned in each of these papers.

We validate on the first 1,000 examples or the first 10% of the data (whichever is smaller) to select the number of mixture components, the number of hidden states, and the window size l, where l is set to be no larger than 10. The input encoder is still a one-layer fully connected neural network with the number of neurons being one of the hyperparameters as well. This validation routine is consistent with all three papers we compare with.

First, we compare with several density-based classifiers listed in [Chen et al., 2021]. Note that "Cover type," "Poker hand," "Transient chessboard," "Rialto," and "Mixed drift" are originally multi-class data streams. Following the same procedure as in [Chen et al., 2021], we generate their binary versions by extracting the two largest classes from each data stream, respectively. The AUC scores on various datasets are presented in Table 7.1. From this table, we can see that all RRNADE variants consistently outperform the other methods. In addition, in many datasets, such as "Electricity," "Hyperplane," etc., we outperform the best compared method by a significant margin.

Second, we compare with multiple non-density-based classification methods in both [Coelho and Barreto, 2022] and [Losing et al., 2018]. The running average of the classification accuracy is reported in Table 7.3 and Table 7.2, respectively. We can see that in almost every dataset, we achieve competitive results, if not significantly better. For synthetic datasets, "Inter RBF" has various Gaussians replacing each other every 3000 samples, representing an abrupt concept drift, while "Moving RBF" is constructed such that Gaussian distributions with random initial positions, weights, and standard deviations are moved with constant speed, representing a gradual concept drift. Here we can see that on both of

Table 7.4: Properties for all experimented datasets.

Data stream	Data stream type	#inst.	#feat.	#class.	Drift type
Sea	Artificial	50,000	3	2	Abrupt
Hyperplane	Artificial	200,000	10	2	Gradual
Mixed Drift	Artificial	177,117	2	15	Mixed
Transient Chessboard	Artificial	50,994	2	8	Reoccurring
Moving RBF	Artificial	200,000	10	2	Gradual
Electricity	Real World	45,312	5	2	Unknown
Cover Type	Real World	495,141	54	7	Unknown
Poker Hand	Real World	765,952	10	4	Unknown
Rialto	Real World	16,450	27	10	Unknown
Border	Real World	5,000	2	3	Unknown
Overlap	Real World	4,950	2	4	Unknown
Outdoor	Real World	4,000	21	40	Unknown
COIL	Real World	7,200	21	100	Unknown
Weather	Real World	18,159	8	2	Unknown

these datasets, we outperform other methods, further showcasing RRNADE's ability to adapt to concept drift. For real-world data, "Weather," "Electricity," "Outdoor," "COIL," and "Rialto" are all data streams with sequential dependencies. Here we can also see that on these datasets, RRNADE outperforms other compared methods. A summarization of all datasets can be found in Table 7.4

Last but not least, in addition to the compared baselines in the corresponding papers, we have also included the original RNADE method as a baseline. As can be observed from the tables, RNADE performs poorly under the online streaming setting. This further strengthens our motivation of incorporating recurrent units into the RNADE model for it to handle online streaming tasks.

7.4.3 Ablation Study

In this ablation study, we investigate the significance of RRNADE's two components, namely the density module ξ and the recurrent module g. We compare RRNADE against three baselines: RRNADE without the recurrent module (NR), RRNADE without the density module (ND), and RRNADE without both modules (NRND). For NR, we re-

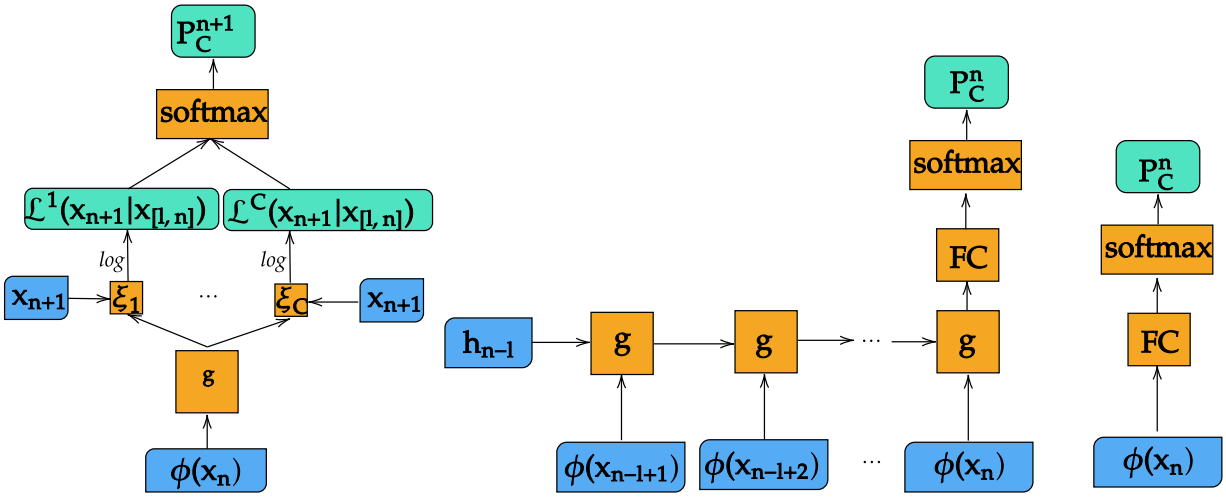


Figure 7.3: The three baseline models NR (left), ND (middle), NRND (right).

place RRNADE's recurrent module with a two-layer fully connected neural network that only takes the data at the current and previous time steps as input. For ND, we replace RRNADE's density module with a two-layer fully connected neural network, while for NRND, a two-layer neural network is used to map from the current input step to its label. To correspond to the three variants of RRNADE, ND also has three different recurrent units, namely GRU, LSTM, and 2RNN. A graphical illustration of the baseline models can be found in Figure 7.3. We select the hyperparameters using validation in the same way as mentioned before, and present the results in Table 7.5.

The results show that RRNADE outperforms all baselines on all examined datasets. In some datasets, such as "Moving RBF" and "Inter RBF", both the density module ξ and the recurrent module g are required for the model to achieve the best performance. However, on other datasets such as "Rialto", the absence of the recurrent module alone is detrimental, while for datasets like "Overlap", the density module is of great importance.

The recurrent module captures the temporal relations in the data, which are key to predicting the correct label in some cases. The top figure of Figure 7.4 shows the learning curves of the three variants of RRNADE and NR on the "Rialto" dataset. We can see that NR converges slower and to a worse solution compared to RRNADE, due to the fact that the label has a specific ordering that is hard to capture without the recurrent module.

Accuracy	RRNADE -2RNN	RRNADE -LSTM	RRNADE -GRU	NR	NRND	ND -LSTM	ND -2RNN	ND -GRU
Border	0.96 (0.05)	**0.96 (0.02)**	0.94 (0.03)	0.90 (0.03)	0.80 (0.02)	0.81 (0.02)	0.80 (0.02)	0.81 (0.01)
Overlap	0.75 (0.01)	0.73 (0.01)	**0.77 (0.01)**	0.75 (0.03)	0.65 (0.02)	0.64 (0.01)	0.65 (0.02)	0.66 (0.02)
Inter RBF	0.90 (0.02)	0.92 (0.02)	**0.92 (0.03)**	0.87 (0.02)	0.46 (0.05)	0.87 (0.09)	0.66 (0.08)	0.79 (0.04)
Moving RBF	0.74 (0.01)	0.74 (0.02)	**0.77 (0.01)**	0.51 (0.02)	0.42 (0.01)	0.59 (0.03)	0.44 (0.02)	0.58 (0.04)
Outdoor	0.96 (0.03)	**0.96 (0.02)**	0.94 (0.02)	0.95 (0.01)	0.31 (0.02)	0.42 (0.02)	0.45 (0.03)	0.39 (0.02)
Weather	0.79 (0.00)	**0.79 (0.01)**	0.78 (0.01)	0.78 (0.01)	0.78 (0.01)	0.79 (0.01)	0.78 (0.01)	0.79 (0.0)
Rialto	0.89 (0.05)	**0.94 (0.04)**	0.92 (0.04)	0.82 (0.04)	0.73 (0.01)	0.92 (0.04)	0.83 (0.05)	0.93 (0.03)

Table 7.5: Comparison of RRNADE with three baselines NR, ND, NRND.

In addition, RRNADE-LSTM seems to be converging slower compared to the other two variants. This could be because LSTM is more complex than 2RNN and GRU therefore needs additional iteration to converge.

On the other hand, the use of the density module provides a proper inductive bias in some cases, which accelerates the convergence and improves the final solution. For example, both "Inter RBF" and "Moving RBF" are generated from Gaussians with concept drifts, so they benefit greatly from the density module. We also train the NRND model on the "Overlap" dataset offline using gradient descent with batch size 1, without random permutation of the data. Note that under this training routine, the first epoch is equivalent to learning under the streaming setting. The bottom figure of Figure 7.4 shows the learning curves of the first 10 epochs for this trained NRND model, as well as for NR and RRNADE-GRU. As the number of epochs increases, NRND slowly reaches the performance of NR. However, in the first epoch, i.e., the online setting, we can see that methods with the density module improve much faster.

7.5 Conclusion

In this chapter, we propose the Recurrent Real-valued Neural Autoregressive Density Estimator (RRNADE), which is an extension of the classic RNADE model to its online setting. The core idea behind RRNADE is to use a recurrent function (recurrent module) to maintain a set of sufficient statistics for the future and approximate the conditional density function using a mixture of Gaussian (density module) that is parameterized by

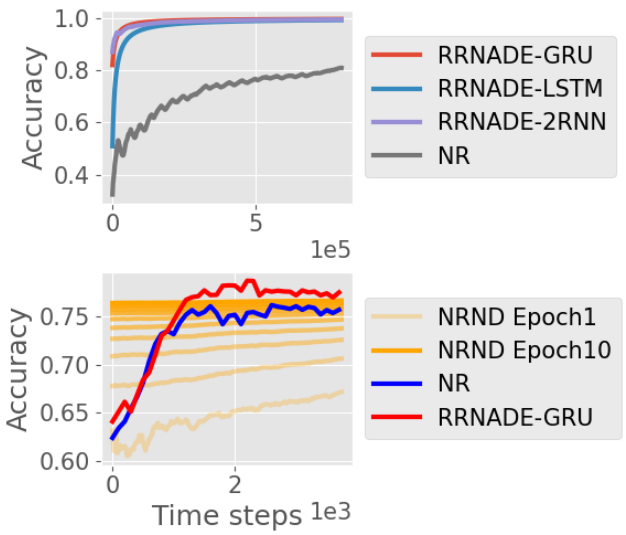


Figure 7.4: Comparison of RRNADE with three baselines NR, ND, NRND.

neural networks. We prove that RRNADE is strictly more expressive than the Gaussian hidden Markov model, which is a classic model for learning sequential data. To use RRNADE on online density estimation and classification of data streams, we propose learning algorithms. In our empirical studies, we conduct experiments on synthetic data to show that RRNADE can learn the density function parameterized by a Gaussian HMM and efficiently adapt to concept drifts. For classification tasks, we compare RRNADE with various methods on multiple synthetic and real-world datasets, and our results demonstrate that RRNADE outperforms all other methods on almost every dataset. In our ablation study, we further showcase the importance of both the recurrent module and the density module, where the recurrent module helps capture the sequential dependencies of the data stream, while the density module aids in online optimization. For future work, we would like to investigate a more adaptive way of estimating the prior, since we use a uniform distribution as the prior for RRNADE on classification tasks. Additionally, as our model is a density model, it would be interesting to investigate the possibility of online clustering.

Chapter 8

Discussion and Conclusion

In this thesis, we explore various approaches to enhance the effectiveness of state representations in state space models (SSMs) under different settings. The thesis begins by introducing some well-known SSMs, including finite state automata and recurrent neural networks. We focus on weighted finite automata (WFAs) and the classic spectral learning algorithm for estimating them from data. Building upon this, we extend WFAs to operate in continuous input spaces and unveil relationships between WFAs, second-order RNNs, and tensor networks from different fields. We show that WFAs are equivalent to linear 2-RNNs and uniform matrix product states in quantum physics when continuous input variables are given. Furthermore, we propose a spectral learning algorithm to recover the corresponding WFAs, which preserves some of the key characteristics of the original method. To achieve a more expressive and compact state representation, we introduce nonlinearities to WFAs with both discrete and continuous input variables. For classic WFAs, we use a nonlinear encoder-decoder framework to replace the low-rank decomposition of the Hankel matrix and solve the transition function, parameterized with a neural network, as a regression task. For WFAs with continuous input (i.e., linear 2-RNNs), we employ a nonlinear feature mapping and a nonlinear output function to obtain nonlinear continuous WFAs. The spectral learning algorithm then involves estimating the corresponding Hankel tensor via gradient descent and recovering the transition tensor of the

nonlinear continuous WFAs using tensor algebra. Note that besides the fact that NL-WFA deals with discrete input variables while NCWFA deals with continuous ones, the difference between NL-WFA and NCWFA also lies in other aspects. Although the learning algorithm of NL-WFA, i.e., the rank factorization on the Hankel matrix of finite rank, constrains the learned NL-WFA to be a more compact representation of another WFA, the definition of NL-WFA is quite general. In our formulation, NL-WFA is a nonlinear RNN for discrete variables while NCWFA is a CWFA (linear 2-RNN) with a nonlinear output function. Aside from WFAs, we also explored other forms of SSMs in this thesis. In Chapter 3, we introduced the unnormalized Q function as a tool for combining the learning and planning phases of POMDPs, using a spectral learning algorithm to obtain an informative state representation for the given task. In the final chapter, we proposed the recurrent real-valued neural autoregressive density estimator (RRNADE), which utilizes recurrent neural networks and RNADE [Uria et al., 2013] to compute density functions. The design of the learning algorithm for RRNADE allows the state representation to adapt effectively to account for concept shifts in the data stream setting.

8.1 Limitations

The spectral learning algorithm typically involves two stages: constructing and decomposing the Hankel tensor and recovering the model parameters via regression. However, these stages can pose significant challenges, particularly with respect to the curse of dimensionality. In this thesis, we explored several techniques to address these issues. For example, in UQF, we applied compressed sensing to reduce the size of the Hankel matrix, while for WFAs with continuous input space, we leveraged the tensor train format of the Hankel tensor to significantly reduce its size and complexity of the decomposition and regression steps in the spectral learning algorithm. Nonetheless, some challenges remain unsolved. For instance, what if the ground truth SSM has a nonlinear transition function? Although our nonlinear WFA can solve this problem by using a nonlinear

encoder-decoder to decompose the Hankel matrix and a neural network to parameterize the transition function, this approach may not work if the ground truth transition function is highly nonlinear, resulting in a Hankel matrix (tensor) of infinite (TT) rank. Therefore, the rank of the Hankel matrix (tensor) we constructed with limited prefixes and suffixes will never be the same as the true rank, hence breaking the fundamental theorem of the spectral learning algorithm.

Another issue in learning WFAs and related linear/multi-linear SSMs is numerical stability. In practice, compounding the transition matrix/tensor can lead to vanishing or exploding state representations, making learning and inference challenging, especially for WFAs with continuous inputs. Although constraining the norm of the state in gradient-based methods can help, empirical results suggest that such constraints can hurt the performance of the learned model. A more systematic approach may be necessary to address this issue.

8.2 Future Directions

In this section, we will outline some interesting problems to work in the future. We will also briefly describe some ideas for these problems.

8.2.1 Hankel Function

One of the future directions, following the previous subsection, is to investigate if it is possible to perform spectral learning like algorithms for SSMs with nonlinear transition functions. One interesting angle might be rethinking the Hankel structures. For the classic WFAs with discrete inputs, we use prefixes and suffixes to index the rows and columns and construct the Hankel matrix. For WFAs with continuous inputs, we construct the high-dimensional Hankel tensor. The one-on-one correspondence is that the size of WFAs is the same as the rank of the Hankel matrix or the tensor train rank of the Hankel tensor. For SSMs with nonlinear transition functions, these constructions fall short due to the

potential infinite matrix (TT) rank. How to resolve this issue could be an interesting future direction. Below we will outline the sketch of one potential solution for this problem, which is the concept of Hankel function.

In the spectral learning for classic WFAs, the algorithm starts with the construction of the Hankel matrix, i.e. a matrix that is indexed by prefixes and suffixes. We can view this matrix as a mapping, i.e.: $\mathscr{H} : \Sigma^* \times \Sigma^* \to \mathbb{R}^p$. In the Hankle tensor construction of the linear 2-RNN in section 4.5.3, we leveraged the TT form of the Hankel tensor and use the spectral learning method to recover the transition tensor. In this process, we are essentially distilling a uniform transition tensor from the L different tensor cores of the TT representation of the Hankel tensor, where L is the length of the sequences. These observations inspire us to extend the notion of Hankel matrix to Hankel function, which maps sequences of vectors (strings under the discrete setting) to the output space, with the use of L different transition functions.

Assume we want to estimate an NCWFA with a nonlinear transition function. Recall the definition for NCWFA mentioned in Definition 23, an NCWFA with a linear transition function is defined by $\tilde{A} = \langle \boldsymbol{\alpha}, \xi, \phi, \mathscr{A} \rangle$, where $\boldsymbol{\alpha} \in \mathbb{R}^k$ is the initial vector, $\xi : \mathbb{R}^k \to \mathbb{R}^p$ is the output function, $\phi : \mathbb{R}^d \to \mathbb{R}^{d*}$ is the encoding function and $\mathscr{A} \in \mathbb{R}^{k \times d \times k}$ is the transition tensor. For an NCWFA with a nonlinear transition function, instead of the tensor $\mathscr{A}$, we have the recurrent function $g : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}^k$. Formally, an NCWFA with a nonlinear transition function is a tuple $\tilde{A}' = \langle \boldsymbol{\alpha}, \xi, \phi, g \rangle$, which computes the function

$$f_{\tilde{A}'}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \xi(g(\phi(\boldsymbol{x}_n), \boldsymbol{h}'_{n-1})),$$

where $\boldsymbol{h}'_t = \phi(\boldsymbol{x}_t, \boldsymbol{h}'_{t-1})$ and $\boldsymbol{h}'_0 = \boldsymbol{\alpha}$, for a sequence $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$. Assume we are given data generated from $\tilde{A}'$, namely $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \boldsymbol{y})$ and $\boldsymbol{y} = f_{\tilde{A}'}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$. Let $g_t : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}^k$ be the transition function at time step t, we want to compute the function $\mathscr{H}^n : (\mathbb{R}^d)^n \to \mathbb{R}^p$ such that $\mathscr{H}^n(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \xi(g_n(\phi(\boldsymbol{x}_n), \boldsymbol{h}_{n-1})) = \boldsymbol{y}$, where $\boldsymbol{h}_t = g_t(\phi(\boldsymbol{x}_t), \boldsymbol{h}_{t-1})$ and the

*For simplicity we set the encoding dimension to be d as well

initial weight vector $\boldsymbol{h}_0 = \boldsymbol{\alpha} \in \mathbb{R}^k$. We refer to the function $\mathscr{H}^n$ as the Hankel function of length n, mapping from the space of sequences of vectors $(\mathbb{R}^d)^n$ to some output space $\mathbb{R}^p$. Note when $g_t = g$ for $t = 1, \cdots, n$, we recover exactly the form of $f_{\tilde{A}'}$.

The learning of the Hankel function can then leverage the classic gradient descent method to minimize the distance between the output of the model and the true target. For an input sequence and output vector pair: $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \boldsymbol{y})$, we want to minimize the following loss function:

$$\mathscr{L}_H(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \boldsymbol{y}) = \|\mathscr{H}^n(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) - \boldsymbol{y}\|^2 \tag{8.1}$$

$$= \|\mathscr{H}^n(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) - f_{\tilde{A}'}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)\|^2 \tag{8.2}$$

In the above optimization, we solve for $\hat{\phi}$, $\hat{\boldsymbol{\alpha}}$, $\hat{\xi}$ and g_t for $t = 1, 2, \cdots, n$. What is left to be recovered for the NCWFA is the nonlinear transition function g. We want to solve for this function such that for an input-output pair $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, \boldsymbol{y})$, we have

$$\hat{\xi}(\hat{g}(\hat{\phi}(\boldsymbol{x}_n), \boldsymbol{h}'_{n-1})) = \boldsymbol{y}.$$

Notice that given $\mathscr{H}^n$, we can obtain an encoding of any prefix up to length n of the input sequences, namely, we have $\Phi_p^m(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_m) = g_m(\phi(\boldsymbol{x}_m), \boldsymbol{h}_{m-1})$, for any $m \leq n$. Recall the encoder-decoder perspective of the spectral learning algorithm we mentioned in Chapter 5 at section 5.2.2, during the regression step of the spectral learning algorithm, we are essentially solving for a function $\hat{g} : \mathbb{R}^k \times \mathbb{R}^d \to \mathbb{R}^k$ such that it minimizes the following loss function:

$$\mathscr{L}_R(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n) = \sum_{m=1}^{n-1} \|\hat{g}(\Phi_p^m(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_m), \hat{\phi}(\boldsymbol{x}_{m+1})) - \Phi_p^{m+1}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{m+1})\|^2 \tag{8.3}$$

Our idea then is to minimize the above loss function to solve for the transition function g of the NCWFA. To conclude, we obtain estimations for the encoder ϕ, initial vector $\boldsymbol{\alpha}$ and the output function ξ in the step of learning the Hankel function by minimizing the loss

in Eq 8.1. Then we learn an estimate of the transition function g by minimizing the loss in Eq 8.3. Some interesting further directions stemming from this idea could be: theoretically, assuming all minimization reaches to zero training error, could we still obtain the results of minimality and consistency that the classic spectral learning algorithm enjoys? Empirically, the above routine can be applied to other SSMs as well, such as various types of RNNs. Evaluating this alternative method of learning or initializing an RNN could also be an interesting direction of research.

8.2.2 Transition for Multimodal Data

In the linear 2-RNN model, the transition tensor is a third-order tensor where one of the dimensions is for the input variables at each time step. This construction potentially faces a scalability issue where we are dealing with multimodal data, namely for data with different data types. For example, video data can be considered as an example of sequential multimodal data, where each time step consists of two types of input data: sound and image, each representing a modality. The scalability issue arises especially when we are interested in the multiplicative relations of the different modalities of the inputs. In this scenario, a simple concatenation of the data on different modalities will fail to capture the multiplicative structure between these modalities. Often, one straightforward solution is to use the outer product of the input variables among all the modalities. This solution, however, will suffer from the curse of dimensionality due to the outer product, which leads to the exponential size of the input vector w.r.t. the number of modalities. Take multi-agent reinforcement learning as an example, each agent has its observations and actions as its own modality, assume to be both of size d. Then in an n-agents multiagent system, the transition tensor is of the size $\mathcal{O}(d^n)$, which makes it difficult for both learning and inference. To solve this issue, one idea could be leveraging various tensor decomposition forms of the transition tensor to directly learn and infer with the decomposed form of the transition tensor, which circumvents the scalability issue. One future direction of this

could be investigating the impact of different tensor decomposition forms on the model performance. Let us briefly investigate some possible choices and their formulations.

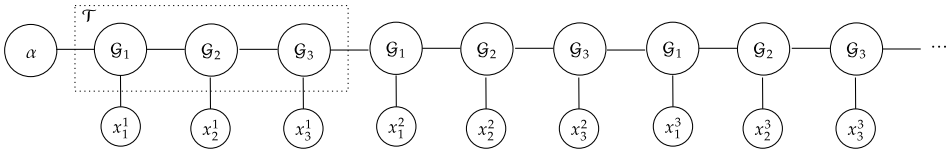


Figure 8.1: Tensor train representation of the transition tensor $\mathcal{T}$

As we have discussed in previous chapters, the key idea of tensor train decomposition is to decompose the tensor into an ordered contraction of a series of small tensor cores. An important property is that, for higher-order tensors, TT decomposition provides a space-saving representation while preserving the expressivity. In Figure 8.1, we present the tensor train representation of the transition tensor $\mathcal{T}$ for an SSM with 3 modalities in the input data, where the initial state vector is α, x_i^j denotes the input of the i-th modality at time step j. In this formulation, the size of the transition tensor $\mathcal{T}$ is linear w.r.t. the number of modalities, which significantly reduces the memory cost. One thing important is that, by using this form of decomposition, we enforce a notion of distance among all the modalities as the contraction of the TT form has to follow the order of the TT cores. As a result, we might enforce an incorrect inductive bias on the correlations between different modalities.

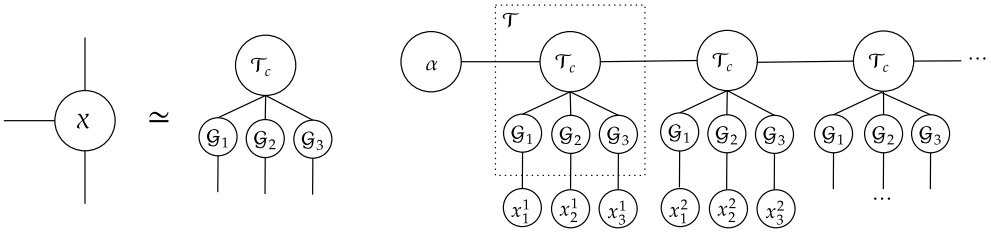


Figure 8.2: Left: Tensor diagram representation of tucker decomposition of a third order tensor $\mathcal{X}$. **Right**: Tucker representation of the transition tensor $\mathcal{T}$.

Tucker decomposition [Tucker, 1966] is a multilinear extension of the matrix singular value decomposition (SVD) that factorizes a higher-order tensor into a core tensor multi-

plied by a matrix along each mode. Figure 8.2, we present a tensor diagram illustration of the Tucker decomposition of a third-order tensor $\mathcal{X}$. The core tensor $\mathcal{T}_c$ encodes the higher-order relationships among the modes. The matrix factors $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ capture the mode-specific information, which can be interpreted as the projection of the original tensor onto each mode. Tucker decomposition has been widely used in the machine learning community, for its ability to reveal underlying patterns and reduce the dimensionality of high-dimensional data. Figure 8.2 shows the Tucker decomposition of the transition tensor $\mathcal{T}$. The intuition here is different from the tensor train decomposition in the sense that the Tucker core $\mathcal{T}_c$ in this formulation can capture all the correlations of each modality through contractions, given enough size. Therefore, there is no sense of distance, as the contractions between the matrix factors and the core tensor are permutation invariant, which allows for all possible interactions among all the modes. One drawback of such a structure is that the size of the Tucker core $\mathcal{T}_c$ is exponential to the number of modalities, which is infeasible when dealing with data with many modalities.

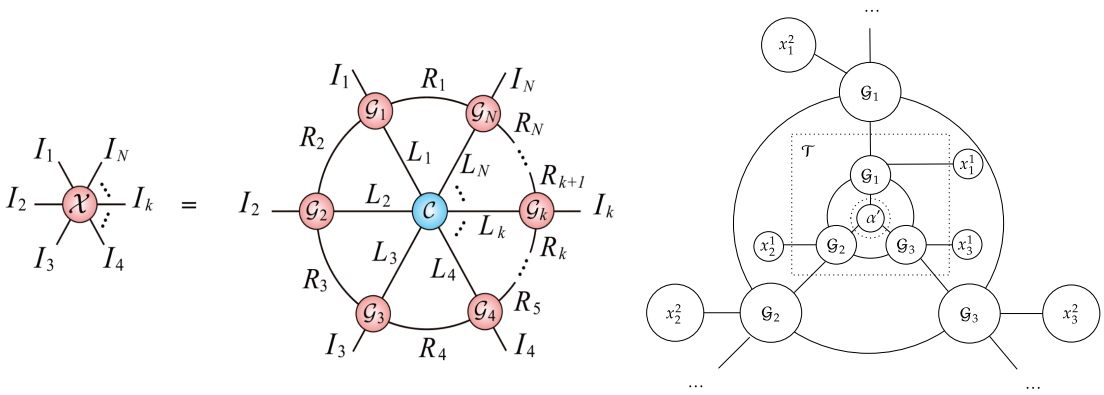


Figure 8.3: Left: Tensor diagram representation of tensor wheel decomposition of the tensor $\mathcal{X}$. Figure credit [Wu et al., 2022]. **Right**: Tensor wheel representation of the transition tensor $\mathcal{T}$.

Another interesting form of decomposition is the tensor wheel decomposition [Wu et al., 2022]. A tensor diagram representation of the tensor wheel decomposition is shown in Figure 8.3, where $\mathcal{C}$ is referred to as the core factor and the remaining tensor cores are

referred to as the ring factors. Building upon tensor ring decomposition [Zhao et al., 2016], tensor wheel decomposition enjoys the nice property of core-connected invariance, which means that tensor wheel topology can comprehensively establish all possible mode interactions of a high-order tensor through the connection of the core factor. Figure 8.3 shows the decomposition of the transition tensor $\mathcal{T}$ in its tensor wheel format. The system starts with the initial state distribution $\boldsymbol{\alpha}' = \boldsymbol{\alpha} \otimes \boldsymbol{\alpha} \otimes \boldsymbol{\alpha}$. Combining with the ring factors $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ and the core factor $\boldsymbol{\alpha}'$, assuming proper contractions between the input variables and the ring factors are done, we have a tensor wheel decomposition. As the tensor wheel topology allows contraction in any order, due to the core-connected invariance, this format, compared to the previous tensor train format, can capture the correlation between arbitrary pairs of modalities (factors). However, similar to the Tucker decomposition, the core factor is of exponential size w.r.t. the number of modalities, which could be the bottleneck. Nevertheless, because of the connections in the neighboring ring factors, which establish a connection for a higher characterization capacity and reduce the loadings of the core factor, the core factor of the tensor wheel decomposition tends to be smaller compared to the one in Tucker [Wu et al., 2022]. Thus, this construction could be smaller than the Tucker structure, and to a certain degree, alleviate the curse of dimensionality in the core factor.

These are three variants of potential decompositions and the structure of the transition tensor when dealing with multimodal data. As we have shown, each has its own advantages and disadvantages. To investigate which form of decomposition to use for what type of data could be an interesting future topic. Moreover, here we are just presenting the potential structures of the decomposed transition tensor, how to learn these forms efficiently, particularly using a spectral learning algorithm, could also be very interesting. Last but not least, we only considered three specific forms of decomposition of the transition tensor, are these predefined topologies optimal? Is there a way to automatically discover the optimal decomposition structure of the transition tensor?

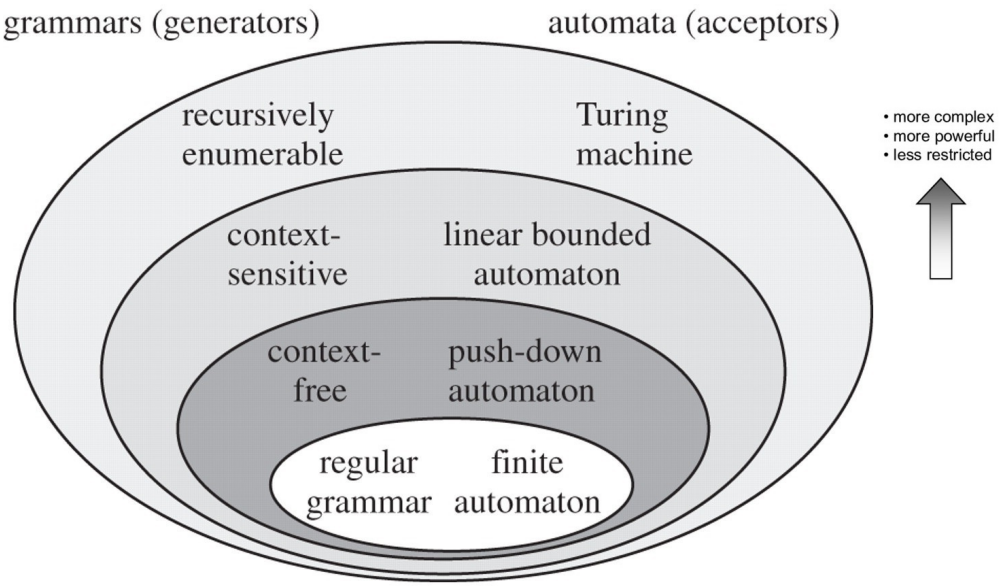


Figure 8.4: The Chomsky Hierarchy. (Figure credit [Fitch, 2014])

8.2.3 Continuous UQF

One last future direction that builds on Chapter 3 is to extend the notion of unnormalized Q functions to environments with continuous observation and action spaces. Briefly, this can be achieved by first obtaining the density of the action-observation sequences using any density estimation method, multiplying it with the immediate rewards, and learning the continuous WFA that computes the function $p(x)r(x)$ using the spectral learning algorithm. The challenge lies in converting this WFA to obtain the UQF, as the summations in the discrete setting become integrals. Finally, planning with the learned UQF under continuous action space can be accomplished with linear programming. This extension can have important applications in various fields, such as robotics and autonomous vehicles.

8.3 Concluding Remarks

Recently, transformers [Vaswani et al., 2017] have emerged as a highly efficient architecture for processing large volumes of sequential data. While originally designed for natural language processing, transformers have since been applied to a diverse range of tasks

and demonstrated potential in areas such as computer vision and speech recognition. It seems that the numerous successes of transformers have rendered SSMs irrelevant in the community. However, as a fundamental tool for extracting temporal relations in sequential data, SSMs still play an important role in modern machine learning. For example, automata, as a well-studied model, provide a convenient tool for evaluating the expressivity of the model, such as the Chomsky hierarchy [Chomsky, 1956] (Fig 8.4). Studies have been conducted to comprehend the expressivity of transformers by leveraging the theory of automata [Liu et al., 2022, Bhattamishra et al., 2020]. Additionally, recent research on state space models, such as [Gu et al., 2020, 2022], utilizing a specific structure of the transition matrices, have managed to achieve superior results compared to transformers in the domains studied. All these recent advances shed a light on the importance of SSMs. Moving on forward, further understanding and establishing the relationships between various types of SSMs from different fields, i.e. automata, RNNs, differential equations, tensor networks, etc, should be a fundamental step to further advance the field of SSMs.

Bibliography

Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of computer and System Sciences*, 66(4):671–687, 2003.

Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

Enes Avcu, Chihiro Shibata, and Jeffrey Heinz. Subregular complexity and deep learning. *arXiv preprint arXiv:1705.05940*, 2017.

Stéphane Ayache, Rémi Eyraud, and Noé Goudian. Explaining black boxes on sequential data using weighted automata. In *Proceedings of ICGI*, pages 81–103, 2018.

Raphaël Bailly, François Denis, and Liva Ralaivola. Grammatical inference as a principal component analysis problem. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 33–40. ACM, 2009.

Raphaël Bailly, Amaury Habrard, and François Denis. A spectral approach for probabilistic grammatical inference on trees. In *Algorithmic Learning Theory*, pages 74–88. Springer, 2010.

Borja Balle and Mehryar Mohri. Spectral learning of general weighted automata via constrained matrix completion. In *Advances in Neural Information Processing Systems*, pages 2159–2167, 2012.

Borja Balle, Xavier Carreras, Franco M Luque, and Ariadna Quattoni. Spectral learning of weighted automata. *Machine learning*, 96(1-2):33–63, 2014a.

Borja Balle, William Hamilton, and Joelle Pineau. Methods of moments for learning stochastic languages: Unified presentation and empirical comparison. In *International Conference on Machine Learning*, pages 1386–1394, 2014b.

Borja Balle, Rémi Eyraud, Franco M Luque, Ariadna Quattoni, and Sicco Verwer. Results of the sequence prediction challenge (spice): a competition on learning the next symbol in a sequence. In *International Conference on Grammatical Inference*, pages 132–136, 2017.

Borja de Balle Pigem, Xavier Carreras Pérez, Franco M Luque, and Ariadna Julieta Quattoni. Spectral learning of weighted automata: a forward-backward perspective. *Machine learning*, (October):1–31, 2013.

Richard G Baraniuk and Michael B Wakin. Random projections of smooth manifolds. *Foundations of computational mathematics*, 9(1):51–77, 2009.

Leonard E Baum and John Alonzo Eagon. An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology. *Bull. Amer. Math. Soc*, 73(3):360–363, 1967.

Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563, 1966.

Richard Bellman. Dynamic programming. *Princeton University Press*, 1957a.

Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957b.

Jacob Benesty, M Mohan Sondhi, Yiteng Huang, et al. *Springer handbook of speech processing*, volume 1. Springer, 2008.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Adam L Berger, Peter F Brown, Stephen A Della Pietra, Vincent J Della Pietra, John R Gillett, John D Lafferty, Robert L Mercer, Harry Printz, and Lubo$v$s Ure$v$s. The candide system for machine translation. In *Proceedings of the workshop on Human Language Technology*, pages 157–162. Association for Computational Linguistics, 1994.

James O Berger. Some recent developments in bayesian analysis, with astronomical illustrations. In *Statistical Challenges in Modern Astronomy II*, pages 15–48. Springer, 1997.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Jacob Biamonte and Ville Bergholm. Tensor networks in a nutshell. *arXiv preprint arXiv:1708.00006*, 2017.

Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

Albert Bifet, Bernhard Pfahringer, Jesse Read, and Geoff Holmes. Efficient data stream classification via probabilistic adaptive windows. In *Proceedings of the 28th annual ACM symposium on applied computing*, pages 801–806, 2013.

Jeff A Bilmes et al. A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. *International Computer Science Institute*, 4(510):126, 1998.

Arnold P Boedihardjo, Chang-Tien Lu, and Feng Chen. A framework for estimating complex probability density structures in data streams. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 619–628, 2008.

Byron Boots, Sajid M Siddiqi, and Geoffrey J Gordon. Closing the learning-planning loop with predictive state representations. *The International Journal of Robotics Research*, 30(7): 954–966, 2011.

Byron Boots, Arthur Gretton, and Geoffrey J Gordon. Hilbert space embeddings of predictive state representations. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, pages 92–101. AUAI Press, 2013.

John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

T Tony Cai, Anru Zhang, et al. Rop: Matrix recovery via rank-one projections. *The Annals of Statistics*, 43(1):102–138, 2015.

Emmanuel J Candes and Yaniv Plan. Tight oracle inequalities for low-rank matrix recovery from a minimal number of noisy random measurements. *IEEE Transactions on Information Theory*, 57(4):2342–2359, 2011.

Jack W. Carlyle and Azaria Paz. Realizations by stochastic finite automata. *Journal of Computer and System Sciences*, 5(1):26–40, 1971.

Richard Caron and Tim Traynor. The zero set of a polynomial. *WSMR Report*, pages 05–02, 2005.

Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *Association for the Advancement of Artificial Intelligence*, 1994.

Gert Cauwenberghs and Tomaso Poggio. Incremental and decremental support vector machine learning. *Advances in neural information processing systems*, 13, 2000.

Yining Chen, Sorcha Gilroy, Andreas Maletti, Jonathan May, and Kevin Knight. Recurrent neural networks as weighted language recognizers. In *Proceedings of NAACL-HLT*, pages 2261–2271, 2018.

Zhong Chen, Zhide Fang, Victor Sheng, Jiabin Zhao, Wei Fan, Andrea Edwards, and Kun Zhang. Adaptive robust local online density estimation for streaming data. *International Journal of Machine Learning and Cybernetics*, 12(6):1803–1824, 2021.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.

David N Coelho and Guilherme A Barreto. A sparse online approach for streaming data classification via prototype-based kernel models. *Neural Processing Letters*, pages 1–28, 2022.

Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. In *Conference on Learning Theory*, pages 698–728, 2016.

Jerome T Connor, R Douglas Martin, and Les E Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

Andrew Critch. *Algebraic geometry of hidden Markov and related models*. PhD thesis, University of California, Berkeley, 2013.

Andrew Critch and Jason Morton. Algebraic geometry of matrix product states. *SIGMA*, 10(095):095, 2014.

François Denis and Yann Esposito. On rational stochastic languages. *Fundamenta Informaticae*, 86(1, 2):41–77, 2008.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Carlton Downey, Ahmed Hefny, Boyue Li, Byron Boots, and Geoffrey Gordon. Predictive state recurrent neural networks. *arXiv preprint arXiv:1705.09353*, 2017.

Pierre Dupont, François Denis, and Yann Esposito. Links between probabilistic automata and hidden markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38(9):1349–1371, 2005.

Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

RJ Elliott, WP Malcolm, and A Tsoi. Hmm volatility estimation. In *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, volume 1, pages 398–404. IEEE, 2002.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556, 2005.

Herbert Federer. *Geometric measure theory*. Springer, 2014.

W. Tecumseh Fitch. Toward a computational framework for cognitive biology: Unifying approaches from cognitive neuroscience and comparative cognition. *Physics of Life Reviews*, 11(3):329–364, 2014. ISSN 1571-0645. doi: https://doi.org/10.1016/j.plrev.2014.04.005. URL https://www.sciencedirect.com/science/article/pii/S157106451400058X.

Michel Fliess. Matrices de hankel. *Journal de Mathématiques Pures et Appliquées*, 53(9): 197–222, 1974.

Mirko Forti. The deployment of artificial intelligence tools in the health sector: privacy concerns and regulatory answers within the gdpr. *Eur. J. Legal Stud.*, 13:29, 2021.

Patrick Gelß. *The Tensor-Train Format and Its Applications: Modeling and Analysis of Chemical Reaction Networks, Catalytic Processes, Fluid Flows, and Brownian Dynamics*. PhD thesis, 2017.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

C Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher order recurrent networks and grammatical inference. In *Proceedings of NIPS*, pages 380–387, 1990.

C Lee Giles, Clifford B Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.

Hadrien Glaude and Olivier Pietquin. PAC learning of probabilistic automaton based on the method of moments. In *Proceedings of ICML*, pages 820–829, 2016.

Heitor M Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9):1469–1495, 2017a.

Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM Computing Surveys (CSUR)*, 50(2):1–36, 2017b.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Proceedings of ICASSP*, pages 6645–6649. IEEE, 2013.

Albert Gu, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. Hippo: Recurrent memory with optimal polynomial projections. *Advances in neural information processing systems*, 33:1474–1487, 2020.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *International conference on representation learning*, 2022.

William Hamilton, Mahdi Milani Fard, and Joelle Pineau. Efficient learning and planning with compressed predictive states. *The Journal of Machine Learning Research*, 15(1):3395–3439, 2014.

William L Hamilton, Mahdi Milani Fard, and Joelle Pineau. Modelling sparse dynamical systems with compressed predictive state representations. In *ICML (1)*, pages 178–186, 2013.

Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. Unsupervised generative modeling using matrix product states. *Physical Review X*, 8(3):031012, 2018.

Ahmed Hefny, Carlton Downey, and Geoffrey J Gordon. Supervised learning for dynamical system learning. In *Advances in neural information processing systems*, pages 1963–1971, 2015.

Christoph Heinz and Bernhard Seeger. Cluster kernels: Resource-aware kernel density estimators over streaming data. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):880–893, 2008.

Moritz Heusinger, Christoph Raab, and Frank-Michael Schleif. Passive concept drift handling via momentum based robust soft learning vector quantization. In *International Workshop on Self-Organizing Maps*, pages 200–209. Springer, 2019.

Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):45, 2013.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimization in the tensor train format. *SIAM Journal on Scientific Computing*, 34(2):A683–A713, 2012.

Daniel Hsu, Sham M Kakade, and Tong Zhang. A spectral algorithm for learning hidden markov models. *Journal of Computer and System Sciences*, 78(5):1460–1480, 2012.

Daniel J. Hsu, Sham M. Kakade, and Tong Zhang. A spectral algorithm for learning hidden markov models. In *Proceedings of COLT*, 2009.

Xuedong D Huang, Yasuo Ariki, and Mervyn A Jack. Hidden markov models for speech recognition. 1990.

Adriana Sayuri Iwashita and Joao Paulo Papa. An overview on concept drift learning. *IEEE access*, 7:1532–1547, 2018.

Masoumeh T Izadi and Doina Precup. Point-based planning for predictive state representations. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 126–137. Springer, 2008.

Herbert Jaeger. Observable operator models ii: Interpretable models and model induction. 1997.

Herbert Jaeger. *Discrete Time, Discrete Valued Observable Operator Models: A Tutorial*. GMD-Forschungszentrum Informationstechnik Darmstadt, Germany, 1998.

Herbert Jaeger. Observable operator models for discrete stochastic time series. *Neural Computation*, 12(6):1371–1398, 2000.

Prateek Jain, Raghu Meka, and Inderjit S Dhillon. Guaranteed rank minimization via singular value projection. In *Proceedings of NIPS*, pages 937–945, 2010.

William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

Biing Hwang Juang and Laurence R Rabiner. Hidden markov models for speech recognition. *Technometrics*, 33(3):251–272, 1991.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. 1960.

Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2019.

Valentin Khrulkov, Alexander Novikov, and Ivan Oseledets. Expressive power of recurrent neural networks. In *Proceedings of ICLR*, 2018.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Stefan Klus, Patrick Gelß, Sebastian Peitz, and Christof Schütte. Tensor-based dynamic mode decomposition. *Nonlinearity*, 31(7):3359, 2018.

Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

Matej Kristan, Ale$v$s Leonardis, and Danijel Sko$v$caj. Multivariate online kernel density estimation with gaussian kernels. *Pattern Recognition*, 44(10-11):2630–2642, 2011.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

YC Lee, Gary Doolen, HH Chen, GZ Sun, Tom Maxwell, HY Lee, and C Lee Giles. Machine learning using a higher order correlation network. *Physica D: Nonlinear Phenomena*, 22(1-3):276–306, 1986.

Tianyu Li, Guillaume Rabusseau, and Doina Precup. Nonlinear weighted finite automata. In *Proceedings of AISTATS*, pages 679–688, 2018.

Tianyu Li, Bogdan Mazoure, Doina Precup, and Guillaume Rabusseau. Efficient planning under partial observability with unnormalized q functions and spectral learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2852–2862. PMLR, 2020a.

Tianyu Li, Doina Precup, and Guillaume Rabusseau. Connecting weighted automata, tensor networks and recurrent neural networks through spectral learning. *arXiv preprint arXiv:2010.10029*, 2020b.

Tianyu Li, Bogdan Mazoure, and Guillaume Rabusseau. Sequential density estimation via ncwfas sequential density estimation via nonlinear continuous weighted finite automata. *LearnAut workshop at ICALP 2022*, 2022a.

Tianyu Li, Doina Precup, and Guillaume Rabusseau. Connecting weighted automata, tensor networks and recurrent neural networks through spectral learning. *Machine Learning*, pages 1–35, 2022b.

Xiaoou Li and Wen Yu. Data stream classification for structural health monitoring via on-line support vector machines. In *2015 IEEE first international conference on big data computing service and applications*, pages 400–405. IEEE, 2015.

Yue Li, Dorothy Yanling Zhao, Jack F Greenblatt, and Zhaolei Zhang. Ripseeker: a statistical package for identifying protein-associated transcripts from rip-seq experiments. *Nucleic acids research*, 41(8):e94–e94, 2013.

Nan-Ying Liang, Guang-Bin Huang, Paramasivan Saratchandran, and Narasimhan Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Transactions on neural networks*, 17(6):1411–1423, 2006.

Qin Lin, Christian Hammerschmidt, Gaetano Pellegrino, and Sicco Verwer. Short-term time series forecasting with regression automata. 2016.

Michael L Littman and Richard S Sutton. Predictive representations of state. In *Advances in Neural Information Processing Systems*, pages 1555–1561, 2002.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. *Advances in neural information processing systems*, 30, 2017.

Viktor Losing, Barbara Hammer, and Heiko Wersing. Interactive online learning for obstacle classification on a mobile robot. In *2015 international joint conference on neural networks (ijcnn)*, pages 1–8. IEEE, 2015.

Viktor Losing, Barbara Hammer, and Heiko Wersing. Knn classifier with self adjusting memory for heterogeneous concept drift. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 291–300. IEEE, 2016.

Viktor Losing, Barbara Hammer, and Heiko Wersing. Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing*, 275:1261–1274, 2018.

Yanyun Lu, Khaled Boukharouba, Jacques Boonært, Anthony Fleury, and Stéphane Lecoeuche. Application of an incremental svm algorithm for on-line human recognition from video surveillance using texture and color features. *Neurocomputing*, 126: 132–140, 2014.

Christian Lubich, Thorsten Rohwedder, Reinhold Schneider, and Bart Vandereycken. Dynamical approximation by hierarchical tucker and tensor-train tensors. *SIAM Journal on Matrix Analysis and Applications*, 34(2):470–494, 2013.

Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. In *Advances in Neural Information Processing Systems*, pages 2229–2239, 2019.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A Smith, and Eran Yahav. A formal hierarchy of rnn architectures. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 443–459, 2020.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *Proceedings of ICASSP*, pages 5528–5531. IEEE, 2011.

Jacob Miller, Guillaume Rabusseau, and John Terilla. Tensor networks for language modeling. *arXiv preprint arXiv:2003.01039*, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Cristopher Moore. Dynamical recognizers: Real-time language recognition by analog computers. In *Foundations of Computational Mathematics*, pages 278–286. Springer, 1997.

Shunji Mori, Ching Y Suen, and Kazuhiko Yamamoto. Historical review of ocr research and development. *Proceedings of the IEEE*, 80(7):1029–1058, 1992.

Muhammad Naeem, Tauseef Jamal, Jorge Diaz-Martinez, Shariq Aziz Butt, Nicolo Montesano, Muhammad Imran Tariq, Emiro De-la Hoz-Franco, and Ethel De-La-Hoz-Valdiris. Trends and future perspective challenges in big data. In *Advances in intelligent data analysis and applications*, pages 309–325. Springer, 2022.

Alexander Novikov, Anton Rodomanov, Anton Osokin, and Dmitry Vetrov. Putting mrfs on a tensor train. In *International Conference on Machine Learning*, pages 811–819, 2014.

Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015.

Christian W Omlin and C Lee Giles. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM (JACM)*, 43(6):937–972, 1996.

Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33 (5):2295–2317, 2011.

Hao Peng, Roy Schwartz, Sam Thomson, and Noah A Smith. Rational recurrences. In *Proceedings of EMNLP*, pages 1203–1214, 2018.

Roberto Pieraccini, Esther Levin, and Enrique Vidal. Learning how to understand language. In *Third European Conference on Speech Communication and Technology*, 1993.

Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *International Joint Conference on Artificial Intelligence*, volume 3, pages 1025–1032, 2003.

Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Anytime point-based approximations for large pomdps. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

Robi Polikar, Lalita Upda, Satish S Upda, and Vasant Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 31(4):497–508, 2001.

Jordan B Pollack. The induction of dynamical recognizers. *Machine learning*, 7(2):227–252, 1991.

Cecilia M Procopiuc and Octavian Procopiuc. Density estimation for spatial data streams. In *International Symposium on Spatial and Temporal Databases*, pages 109–126. Springer, 2005.

Abdulhakim Qahtan, Suojin Wang, and Xiangliang Zhang. Kde-track: An efficient dynamic density estimator for data streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(3):642–655, 2016.

Ariadna Quattoni, Xavier Carreras, and Matthias Gallé. A maximum matching algorithm for basis selection in spectral learning. In *Artificial Intelligence and Statistics*, pages 1477–1485. PMLR, 2017.

Guillaume Rabusseau. *A Tensor Perspective on Weighted Automata, Low-Rank Regression and Algebraic Mixtures*. PhD thesis, Aix-Marseille Université, 2016.

Guillaume Rabusseau, Borja Balle, and Joelle Pineau. Multitask spectral learning of weighted automata. In *Proceedings of NIPS*, pages 2585–2594, 2017.

Guillaume Rabusseau, Tianyu Li, and Doina Precup. Connecting weighted automata and recurrent neural networks through spectral learning. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1630–1639. PMLR, 2019.

Holger Rauhut, Reinhold Schneider, and Željka Stojanac. Low rank tensor recovery via iterative hard thresholding. *Linear Algebra and its Applications*, 523:220–262, 2017.

Adria Recasens and Ariadna Quattoni. Spectral learning of sequence taggers over continuous sequences. In *Proceedings of ECML*, pages 289–304, 2013.

Benjamin Recht, Maryam Fazel, and Pablo A Parrilo. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. *SIAM review*, 52(3): 471–501, 2010.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

Matthew Rosencrantz, Geoff Gordon, and Sebastian Thrun. Learning low dimensional predictive representations. In *Proceedings of the twenty-first international conference on Machine learning*, page 88. ACM, 2004.

Matthew R Rudary and Satinder P Singh. A nonlinear predictive state representation. In *Advances in Neural Information Processing Systems*, pages 855–862, 2004.

Claude Sammut and Geoffrey I. Webb, editors. *Sequential Data*, pages 902–902. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_754. URL https://doi.org/10.1007/978-0-387-30164-8_754.

Atsushi Sato and Keiji Yamada. Generalized learning vector quantization. *Advances in neural information processing systems*, 8, 1995.

Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of Physics*, 326(1):96–192, 2011.

Hanie Sedghi and Anima Anandkumar. Training input-output recurrent neural networks through spectral methods. *arXiv preprint arXiv:1603.00954*, 2016.

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10 (Nov):2615–2637, 2009.

Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of COLT*, pages 440–449. ACM, 1992.

David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Satinder Singh, Michael R James, and Matthew R Rudary. Predictive state representations: A new theory for modeling dynamical systems. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 512–519. AUAI Press, 2004.

Satinder P Singh, Michael L Littman, Nicholas K Jong, David Pardoe, and Peter Stone. Learning predictive state representations. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 712–719, 2003.

Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934, 2013a.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic composition-

ality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013b.

Edward J Sondik. The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operations research*, 26(2):282–304, 1978.

Edwin Stoudenmire and David J Schwab. Supervised learning with tensor networks. In *Advances in Neural Information Processing Systems*, pages 4799–4807, 2016.

Wen Sun, Arun Venkatraman, Byron Boots, and J Andrew Bagnell. Learning to filter with predictive state inference machines. In *International Conference on Machine Learning*, pages 1197–1205, 2016.

Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of ICML*, pages 1017–1024, 2011.

Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

Binhua Tang, Zixiang Pan, Kang Yin, and Asif Khateeb. Recent advances of deep learning in bioinformatics and computational biology. *Frontiers in genetics*, 10:214, 2019.

Ann Taylor, Mitchell Marcus, and Beatrice Santorini. The penn treebank: an overview. In *Treebanks*, pages 5–22. Springer, 2003.

Michael Thon and Herbert Jaeger. Links between multiplicity automata, observable operator models and predictive state representations: a unified learning framework. *The Journal of Machine Learning Research*, 16(1):103–147, 2015.

Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. In *Proceedings of IJCNN*, pages 4451–4458. IEEE, 2017.

Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

Benigno Uria, Iain Murray, and Hugo Larochelle. Rnade: The real-valued neural autoregressive density-estimator. *Advances in Neural Information Processing Systems*, 26, 2013.

Benigno Uria, Marc-Alexandre Côté, Karol Gregor, Iain Murray, and Hugo Larochelle. Neural autoregressive distribution estimation. *The Journal of Machine Learning Research*, 17(1):7184–7220, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A monte-carlo aixi approximation. *Journal of Artificial Intelligence Research*, 40:95–142, 2011.

Arun Venkatraman, Nicholas Rhinehart, Wen Sun, Lerrel Pinto, Martial Hebert, Byron Boots, Kris M Kitani, and J Andrew Bagnell. Predictive-state decoders: Encoding the future into recurrent networks. *arXiv preprint arXiv:1709.08520*, 2017.

Sicco Verwer, Rémi Eyraud, and Colin De La Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Machine learning*, 96(1-2):129–154, 2014.

Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

Wenqi Wang, Vaneet Aggarwal, and Shuchin Aeron. Efficient low rank tensor ring completion. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5697–5705, 2017.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pages 5247–5256. PMLR, 2018.

Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Proceedings of NIPS*, pages 2856–2864, 2016.

Zhong-Cheng Wu, Ting-Zhu Huang, Liang-Jian Deng, Hong-Xia Dou, and Deyu Meng. Tensor wheel decomposition and its tensor completion application. In *Advances in Neural Information Processing Systems*, 2022.

Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3891–3900. JMLR. org, 2017.

Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. Long-term forecasting using tensor-train rnns. *arXiv preprint arXiv:1711.00073*, 2017.

Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR, 2017.

Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.