UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Sede Amministrativa: Università degli Studi di Padova

Dipartimento di Matematica

CORSO DI DOTTORATO DI RICERCA IN: Scienze Matematiche
CURRICOLO: Informatica
CICLO: XXIX

LINEAR MODELS AND DEEP LEARNING: LEARNING IN SEQUENTIAL DOMAINS

Coordinatore: Ch.mo Prof. Pierpaolo Soravia
Supervisore: Ch.mo Prof. Alessandro Sperduti

Dottorando: Luca Pasa

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Alessandro Sperduti for the continuous support of my Ph.D study, for his patience, motivation and enthusiasm.

Besides my advisor, I would like to thank the rest of my thesis committee for their insightful and valuable comments and suggestions.

I would like to thank Dr. Alberto Testolin for all his help and for his guidance before and during my Ph.D studies. Also, I would like to thank Prof. Peter Tino for the meaningful discussions and inputs he provided during my visit to University of Birmingham.

I would like to thank all my friends, colleagues and officemates (Moreno, Riccardo, Alberto, Daniele, Hossein and Ding Ding) here at the University of Padua for the stimulating discussions, and for all the fun we have had in the last three years.

Last but not the least, a huge thank goes to my family, that always supported me in these years. Words cannot express how grateful I am to my mother, and father for all of the sacrifices that they have made on my behalf.

Luca Pasa
Padova, Jan 31, 2017

Abstract

With the diffusion of cheap sensors, sensor-equipped devices (e.g., drones), and sensor networks (such as *Internet of Things*), as well as the development of inexpensive human-machine interaction interfaces, the ability to quickly and effectively process sequential data is becoming more and more important. There are many tasks that may benefit from advancement in this field, ranging from monitoring and classification of human behavior to prediction of future events. Most of the above tasks require pattern recognition and machine learning capabilities.

There are many approaches that have been proposed in the past to learn in sequential domains, especially extensions in the field of Deep Learning. Deep Learning is based on highly nonlinear systems, which very often reach quite good classification/prediction performances, but at the expenses of a substantial computational burden. Actually, when facing learning in a sequential, or more in general structured domain, it is common practice to readily resort to nonlinear systems. Not always, however, the task really requires a nonlinear system. So the risk is to run into difficult and computational expensive training procedures to eventually get a solution that improves of an epsilon (if not at all) the performances that can be reached by a simple linear dynamical system involving simpler training procedures and a much lower computational effort. The aim of this thesis is to discuss about the role that linear dynamical systems may have in learning in sequential domains. On one hand, we like to point out that a linear dynamical system (LDS) is able, in many cases, to already provide good performances at a relatively low computational cost. On the other hand, when a linear dynamical system is not enough to provide a reasonable solution, we show that it can be used as a building block to construct more complex and powerful models, or how to resort to it to design quite effective pre-training techniques for nonlinear dynamical systems, such as Echo State Networks (ESNs) and simple Recurrent Neural Networks (RNNs).

Specifically, in this thesis we consider the task of predicting the next event into a sequence of events. The datasets used to test various discussed models involve polyphonic music and contain quite long sequences. We start by introducing a simple state space LDS. Three different approaches to train the LDS are then considered. Then we introduce some brand new models that are inspired by the LDS and that have the aim to increase the prediction/classification capabilities of the simple linear models.

We then move to study the most common nonlinear models. From this point of view, we considered the RNN models, which are significantly more computationally demanding. We experimentally show that, at least for the addressed prediction task and the considered datasets, the introduction of pre-training approaches involving linear systems leads to quite large improvements in prediction performances. Specifically, we introduce pre-training via linear Autoencoder, and an alternative based on Hidden Markov Models (HMMs).

Experimental results suggest that linear models may play an important role for learning in sequential domains, both when used directly or indirectly (as basis for pre-training approaches): in fact, when used directly, linear models may by themselves return state-of-the-art performance, while requiring a much lower computational effort with respect to their nonlinear counterpart. Moreover, even when linear models do not perform well, it is always possible to successfully exploit them within pre-training approaches for nonlinear systems.

Contents

Chapter 1

Introduction

The research work presented in this thesis deals with the problem of learning in sequential domains. Before diving into the content of the thesis and its contributions, in this chapter we firstly introduce the problem and discuss why the tasks and challenges in this field are interesting. Additionally, in what follows we unveil the additional issues related to the treatment of time and computational burden of the common models used to perform learning in sequential domains.

1.1 Learning in Sequential Domains

A broad range of real-world applications involve learning over sequential data, e.g. classifying time series of heart rates (ECG) to decide if data come from a patient with heart disease, predicting the future value of a company stock traded on an exchange, interpreting a sequence of utterances in speech understanding, predicting the secondary or tertiary protein structure from its DNA sequence, and so on. Performing learning in sequential domains usually involves long sequences that have different lengths. This makes the sequence learning a hard task, and for this reason, different approaches, tailored to specific data and task features (e.g., discrete vs. continuous valued sequences, classification vs. prediction tasks [1], etc.), have been developed. All these approaches can be grouped into three main categories: *i)* feature-based approaches, which transform a sequence into a feature vector and then apply conventional vectorial-based methods (e.g., [25]); *ii)* distance-

[1]The main difference between classification and prediction task regards the output. Indeed, for what concerns the classification task, the output will be the class of a given input. Therefore, the output space is typically smaller than the input space. While in prediction task the output space coincides with the input space.

based approaches, which employ a distance function measuring the similarity between sequences, e.g. Euclidean distance (e.g., [91]), edit-distance (e.g., [64]), dynamic time warping distance (e.g., [94]), or a kernel function (e.g., [58, 28]); *iii)* model-based approaches, such as using Hidden Markov Models (e.g., [74, 95]), or Recurrent Neural Networks (e.g., [32, 34]), to process sequences. Methods falling into the first category are successfully only if apriori knowledge on the application domain can be used to select the most relevant sequence features for the task at hand. A notable example of these approaches, in the case of discrete valued sequences, is the use of short sequence segments of k consecutive symbols (k-grams) as features; a sequence is represented as a vector of the presence/absence/frequency of k-grams. The obtained vectors can then be fed into conventional learning machines, such as Decision Trees [72], Support Vector Machines [29], feed-forward neural networks [20], for any kind of learning task (classification, prediction, ranking, etc.). The drawback of these approaches is that the number of features to consider easily grows exponentially and this will also lead to increased memory consumption (e.g., the size of k in k-grams). If apriori knowledge is not available for pruning the feature space, feature selection strategies (see [42]) need to be used. Moreover, *ad-hoc* strategies, such as discretization, are needed to deal with continuous valued sequences (e.g., [96]). Distance-based approaches treat each sequence as a single entity and exploit a sequence similarity function to determine how similar two sequences are. This information can then be used within an instance-based approach for learning (e.g., k-Nearest Neighbor [30]), or directly inside a kernel method if the used similarity function is a proper kernel. These approaches tend to be expensive from a computational point of view since computing the sequence similarity function usually involves a relevant computational burden (e.,g. edit-distance [76]). Moreover, these approaches usually have problems to extrapolate the learned function to sequences that are longer than the ones used for training.

Finally, model-based approaches assume that the observed sequences, as well as the function to learn, have been generated by a law (or model). Because of that, they aim at reconstructing such model, with the goal of successfully extrapolating the learned function to the whole sequence domain. Model-based approaches are typically computationally demanding, however, if a good approximation of the target model is learned, very good performances on the whole sequence domain can be obtained. Graphical models [56], and in particular Hidden Markov Models (HMMs), are often used as learning models. HMMs assume that each sequence item has been generated by hidden variables that are not directly observable. The way each sequence item observed at time t is generated is described by a (parametric) probability distribution which depends on the *state* at time t of the HMM, i.e. the values taken by the hidden variables at time t; moreover, another (parametric) probability distribution drives the way the values as-

signed to hidden variables change through time. Learning aims at tuning these probability distributions in order to make the observed sequences more likely to be generated when sampling from the model. A deterministic alternative to HMMs is given by Recurrent Neural Networks (RNNs), which can be understood as nonlinear dynamical systems where learning is performed by using gradient-based approaches [20]. From an abstract computational point of view, given a graphical model for sequences, it is possible to state that an RNN constitutes a specific deterministic implementation of that graphical model [35]. Due to their nonlinearity, RNNs are potentially very expressive and powerful. However, they are also difficult to train, mainly because temporal dependencies introduce constraints that limit the efficacy of gradient-based learning algorithms [12] as well as the parallelization of computation. Despite old [49] and recent developments [62, 13], the computational burden to train RNNs still remains very high.

It is worth to note that models can also be used to define kernels (e.g., [50, 7]). One very general way to exploit a generative model for defining a kernel is given by the Fisher kernel approach, originally proposed by [50]. The underpinning idea of Fisher kernel is to use the training data to create a generative model, e.g. an HMM, and then to define a kernel on sequences from the Fisher score vectors extracted from the generative model. Besides being computationally very expensive, Fisher kernel may suffer from quite bad feature representation due to maximum likelihood training which leads to develop a large number of very small gradients (data with high probability under the model) and a few very large ones (data with low probability under the model) [88]. Because of that, Fisher kernel may suffer when used for discriminative tasks. A technique that tries to correct this problem has been proposed in [88]. The basic idea is to learn the generative model parameters in such a way that the resulting embedding has a low nearest-neighbor error. This approach improves the discriminative performance at the expenses of an increased computational cost, which makes Fisher kernel very computational demanding when considering long sequences.

1.1.1 Learning in Sequential Domains and Temporal Constraints

It is important to notice that the computational burden problem of the nonlinear methods reported above is one of the biggest problems when applying one of these methods to real world scenarios. For instance nowadays with the diffusion of cheap sensors, sensor-equipped devices (e.g., drones), and sensor networks (such as *Internet of Things* [8]), as well as the development of inexpensive human-machine interaction interfaces, the ability to quickly and effectively process sequential data is becoming more and more

important. Many are the tasks that may benefit from advancement in this field, ranging from monitoring and classification of human behavior to prediction of future events. Most of the above tasks require pattern recognition and machine learning capabilities. Many are the approaches that have been proposed in the past few years to learn in sequential domains (e.g., [83]). A special mention goes to recent advancements involving Deep Learning [39, 71, 41]. Deep Learning is based on very nonlinear systems, which reach quite good classification/prediction performances but very often at the expenses of a **very high computational burden**.
It is common practice, when facing learning in a sequential, or more in general structured, domain to readily resort to nonlinear systems. In this thesis we wonder if it is always necessary to use this kind of complex models, or if it is possible to exploit simpler models in order to obtain good results but trying to limit the computational burden of the used method in order to make feasible to apply it to some of the real world tasks reported above. Indeed not always the task really requires a nonlinear system. So the risk is to run into difficult and computationally expensive training procedures to eventually get a solution that improves of an epsilon (if not at all) the performances that can be reached by a simple linear dynamical system involving simpler training procedures and a much lower computational effort.

1.2 Contribution

The aim of this thesis is to explore how linear models could be used to perform learning in sequential domains. We firstly explore how a very simple model, the Linear Dynamic System (LDS) can be applied in this context. We introduce three different training methods that can be used in order to perform learning on large datasets. Then we explore if it is possible to create a more powerful model by using LDSs as a building block. The idea is to combine several LDSs to build a more complex nonlinear model. Finally, we study how a linear model can be used in order to help nonlinear models to simplify the learning phase.

1.2.1 Linear Dynamic System and Learning in Sequential Domains

We start by focusing on a particularly simple linear models: the Linear Dynamic System (LDS). LDS is particularly interesting, because for some tasks it is able, in many cases, to already provide good performances at a relatively low computational cost. On the other hand, when a linear dynamical system is not enough to provide a reasonable solution it could be used as a starting point to train more complex models. The first part of this thesis reports our contribution in applying LDS on machine learning tasks. Specifically, here we consider the task of predicting the next event

into a sequence of events. Our first contribution consists in introducing methods that allow to perform training on LDS models. Three different approaches to train the LDS are developed. The first one is based on random projections and it is particularly efficient from a computational point of view. The second, computationally more demanding approach, projects the input sequences onto an approximation of their spanned sub-space obtained via a linear autoencoder [82] naturally associated to the LDS. For both approaches, the computation of the output weights involves the computation of a pseudo-inverse. Finally, we consider a refinement via stochastic gradient descent of the solution obtained by the autoencoder-based training scheme. Of course, this last approach requires additional computational resources. The application of simple LDSs on prediction tasks highlighted the strengths and the weaknesses of this model. In particular, experimental results on benchmark datasets show the importance of performing a wise tuning of the parameters of the model. The comparison among the LDS and the most common nonlinear models on benchmarks datasets has proven that the use of linear models for learning sequential data is certainly useful. Specifically, the comparison involves the Echo State Networks, that can be considered a natural extension of the first linear approach, since nonlinear random projections are used to define a coding of input sequences, and the pseudo-inverse is exploited to estimate the output weights. In addition, these are the less computationally demanding models in the nonlinear models arena. The second considered family of nonlinear models is given by simple RNNs, which computationally are significantly more demanding.

The second step that we made in studying how to apply LDSs in learning in sequential domains, has been exploring if it is possible to combine more LDSs in order to increase the capability of the resulting model. Firstly, our study drove us to wonder whether it is possible to obtain nonlinearity by combining several LDSs. More precisely, our idea is to wisely compose the representation computed by several LDSs fed with the same input, in order to obtain a nonlinear representation of it. The Linear System Network (LSN) is the model that we have developed in order to implement this intuition. The LSN model is attractive thanks to a simple and efficient training procedure. Unfortunately, the obtained experimental results are not much different than the results obtained by the LDS model, that turns out to be easier to train and much more efficient.

Another explored idea in the same line was to apply co-Learning techniques, that consist in exploiting an external model in order to perform training on the feature space of the LDS. Our study uncovered (and formally proved) some critical limitations of the application of Co-learning techniques in the LDS context. The empirical poor results obtained by these models confirmed the correctness of our theoretical results.

1.2.2 Linear models to for simplifying learning in sequential domains

A serious problem that afflicts the existent approaches for sequential data is their difficulties in learning over long sequences: feature-based approaches typically use features associated to the occurrence of short temporal/positional sub-sequences, i.e., *local* features, which fail to capture long-term dependencies. Distance-based approaches typically select a subset of the training sequences as reference to perform the desired computation[2]; because of that, the learned function typically has difficulties to deal with sequences that are longer than the ones used for training. Model-based approaches also tend to have problems capturing long-term dependencies, either because they have discrete finite memory (i.e., the number of different states in which an HMM can be), such as in the most commonly used versions of HMMs, or because learning algorithms fail to find the "right" setting for the parameters, such as in RNNs. Notwithstanding the difficulties in training RNNs, their computational power is so high (see, for example [19]) that it is worth to study new approaches to improve learning. If we turn our attention to static data, recent advances in training *deep* neural networks, i.e. networks composed of many layers of nonlinear processing units, now allow to reach state-of-the-art performance in complex machine learning tasks, such as image classification [57], speech recognition [5] and natural language processing [26]. One reason for this progress is due to the possibility of learning algorithms to enlarge the exploration of the parameter space thanks to the advent of new, high-performance parallel computing architectures, which exploit powerful graphic processors to significantly speed-up learning [75]. However, the breakthrough that allowed to effectively train large-scale "deep" networks has been the introduction of an unsupervised pre-training phase [45], in which the network is trained to build a generative model of the data, which can be subsequently refined using a supervised criterion (fine-tuning phase). Pre-training initializes the weights of the network in a region where optimization is somehow easier, thus helping the fine-tuning phase to reach better local optima. It might also perform some form of regularization, by introducing a bias towards good configurations of the parameter space [33]. The importance to start gradient-based learning from a good initial point in the parameter space has also been pointed out in [84].

Going back to sequences, an interesting research question is whether the benefits of pre-training could also be extended to the temporal domain, and whether it is possible to perform it by using a simple linear models. Up to now, the most popular approaches to pre-train sequential models do not

[2]For example, k-Nearest Neighbor exploits all training sequences or a subset obtained by editing the training set (e.g., [43]); also Support Vector Machines use a subset of the training set, i.e. the support sequences.

take into account temporal dependencies (e.g., [5, 17]) and only pre-train input-to-hidden connections by considering each item of the sequence as independent from the others. This pre-training strategy is clearly unsatisfactory, because by definition the items belonging to the same sequence are dependent from each other, and this information should be exploited also during pre-training.

In this thesis, we propose two alternative pre-training methods. The first one, instead of using the same dataset for both the pre-training and the fine-tuning phases, uses a linear model, such as HMM with a limited number of states, to generate a new dataset, which represents an approximation of the target probability distribution. This simpler, "smoothed" dataset is then used to pre-train a more powerful nonlinear model, which is subsequently fine-tuned using the original sequences. Importantly, this method does not require to develop any *ad-hoc* pre-training algorithm: we can adopt standard gradient descent learning, and apply it first on the approximate distribution and then on the original dataset. We first applied the HMM pre-training on a recently proposed recurrent model [17] that has been shown to obtain state-of-the-art performance on a prediction task for the considered dataset. We then assessed the robustness and the generality of the method by applying it also to a classic recurrent neural network. Our results confirm the value of the proposed pre-training strategy, which allows to learn an accurate model of the data in a significantly shorter time, sometimes also leading to improvements in prediction accuracy.

The second proposed pre-training method is focused on RNN models. One of the hardest problems that makes it difficult to train this model is the **vanishing gradient problem** [45, 44, 31], that makes it unfeasible to deal with long-term temporal dependencies. In that context, there is a growing evidence that effective learning procedures should be based on relevant and robust internal representations developed in autonomy by the learning system. This is usually achieved in vectorial spaces by exploiting nonlinear autoencoder networks to learn rich internal representations of input data which are then used as input to shallow neural classifiers or predictors (see, for example, [11]). The relationship between autoencoder networks and Principal Component Analysis (PCA) [53] is well known since late '80s, especially in the case of linear hidden units [18, 9]. More recently, linear autoencoder networks for structured data have been studied in [80, 65, 81], where an exact closed-form solution for the weights is given in the case of a number of hidden units equal to the rank of the full data matrix. The second proposed pre-training method exploits the conceptual framework presented in [80, 82] to devise an effective pre-training approach, based on linear autoencoder networks for sequences, to get a good starting point into the weight space of an RNN, which can then be successfully trained even in presence of long-term dependencies. Specifically, we revise the theoretical approach

presented in [80] by: *i)* giving a simpler and direct solution to the problem of devising an exact closed-form solution (full rank case) for the weights of a linear autoencoder network for sequences, highlighting the relationship between the proposed solution and PCA of the input data; *ii)* introducing a new formulation of the autoencoder learning problem able to return an optimal solution also in the case of a number of hidden units which is less than the rank of the full data matrix; *iii)* proposing a procedure for approximate learning of the autoencoder network weights under the scenario of very large sequence datasets. More importantly, we show how to use the linear autoencoder network solution to derive a good initial point into an RNN weight space. The tests performed on RNNs pre-trained by using this autoencoder based method returned quite impressive empirical results when applied to prediction tasks.

1.3 Outline of the Thesis

This thesis is composed of four logical parts. In the first part, that consists in Chapter 2, the most common models used to perform training on sequential data are presented. The second part regards the original contributions presented in this thesis. In particular, Chapter 3 regards the study of the application of LDS model to Machine Learning tasks; Chapter 4 introduces some brand new models that are based on LDS. Finally, Chapter 5 discusses about how a linear model can be used in order to preform a pre-training phase of more complex nonlinear models, and how it can help during the training phase. The third part of the thesis (Chapter 6) reports all the experimental results obtained by testing the models presented in the second part. Specifically, the performed tests concern the application of the various models on prediction tasks involving complex musical sequences. Finally Chapter 7 contains conclusion and final remarks.

1.4 Publications

Part of the research presented in this thesis and developed during my Ph.D. program produced peer-reviewed workshop, conference and journal publications. The complete list of published works is listed in the following:

[C1] Luca Pasa, Alberto Testolin, and Alessandro Sperduti: A HMM-based Pre-training Approach for Sequential Data. In: 22th European Symposium on Artificial Neural Networks, 2014.

[C2] Luca Pasa and Alessandro Sperduti: Pre-training of recurrent neural networks via linear autoencoders. In: Advances in Neural Information Processing Systems (NIPS), 2014.

[C3] Luca Pasa, Alessandro Sperduti: Learning Sequential Data with the Help of Linear Systems. In: IAPR Workshop on Artificial Neural Networks in Pattern Recognition, 2016.

[J1] Luca Pasa, Alberto Testolin, Alessandro Sperduti: Neural Networks for Sequential Data: a Pretraining Approach based on Hidden Markov Models. In: Neurocomputing (169), 2015.

Chapter 2

Deep Learning for sequences

In this chapter we introduce and briefly review the most interesting Deep Learning models, developed to deal with sequential data. Our literature review regards the models that reached the best results in different areas and tasks concerning learning in sequential domains. In the first part of the chapter the most common Deep Learning models for sequences are introduced and discussed. These models are widely used in many real world problems, and constitute an essential background to our contribution. In the second part of this chapter we list and briefly explain the brand new models introduced in the last few years, that have been developed for facing some interesting problems in sequential domains. These models are the ones that achieved the most interesting results on some very specific tasks. Some of these models will be later compared to models proposed in this thesis in Chapter 6.

2.1 Base models and theoretical tools

In this section some fundamental models developed to perform learning in sequential domains are introduced. In particular, we start by explaining the **Linear Dynamic System** that is a really simple model that can be used to model sequential phenomena. Then, we introduce another powerful linear model: the **Hidden Markov Model**. Finally, the two of the most common nonlinear models for sequences, **Recurrent Neural Network and Echo State Network**, are discussed.

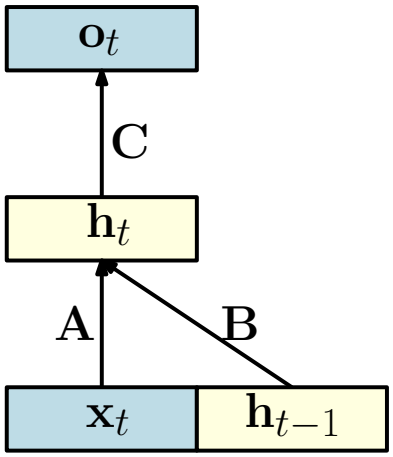

Figure 2.1: Schematic representation of the Linear Dynamic System.

2.1.1 Linear Dynamic System

A **Linear Dynamic System (LDS)** (Figure 2.1) is a mathematical model that allows to represent sequential events and phenomena, widely used in Physics and in Engineering. The discrete time LDS model has an input $\mathbf{x}_t$, an internal state $\mathbf{h}_t$, and an output $\mathbf{o}_t$, at each time step t. The model uses three matrices in order to compute the internal state and the output value, and it is able to describe the time dynamic of a phenomenon, thanks to recurrent connections on its internal state $\mathbf{h}_t$. The simpler linear model is a discrete-time dynamical system defined as:

$$\mathbf{h}_t = \mathbf{A}\,\mathbf{x}_t + \mathbf{B}\,\mathbf{h}_{t-1}, \tag{2.1}$$

$$\mathbf{o}_t = \mathbf{C}\,\mathbf{h}_t, \tag{2.2}$$

where $\mathbf{h}_t \in \mathbb{R}^m$ is the state of the system at time t and, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{m \times m}$, $\mathbf{C} \in \mathbb{R}^{s \times m}$ are the input matrix, the state matrix and the output matrix, respectively. In addition, we assume $\mathbf{h}_0 = \mathbf{0}$, i.e. the null vector. In order to describe a phenomenon, which is characterized by the input data and the output data, the LDS has to be "identified", that consists in finding the model parameters that maximize the model likelihood given the data. In this thesis, we focus on discrete time LDS. Identifying an LDS consists in computing the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ that map the input into the output. The most common techniques to identify an LDS are know as *sub-space identification methods* [59]. Such methods, however, present some important limitations. Indeed, they try to compute an *exact* solution, and diverge in case such solution does not exist. Moreover, these methods are developed to compute the parameters that allow to model the behavior of a physical system that is usually described by a single very long sequence. This limitation makes it difficult to apply them to machine learning tasks.

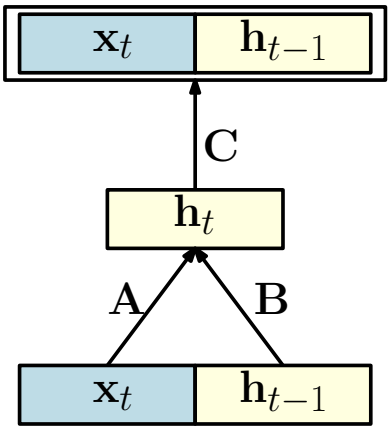

Figure 2.2: Schematic representation of the Autoencoder.

In fact, Machine learning tasks typically use large datasets, and finding an exact solution is sometime impossible. Moreover, in machine learning, the aim is to learn a function that does not overfit the training data, therefore performing training by using system identification methods turns out not to be useful.
A particular type of LDS is the Linear Autoencoder (Figure 2.2) for sequences. The specific feature of the Autoencoder is that the size of the output space is the same as the size of the input space. The definition of Linear Autoencoder is obtained by substituting equation (2.2) with

$$\begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} = \mathbf{C}\,\mathbf{h}_t, \tag{2.3}$$

therefore, the idea is to learn a hidden representation $\mathbf{h}_t$ of the input $\mathbf{x}_t$. It is possible to have also a nonlinear Autoencoder where equations (2.1) and (2.2) are substituted by the following equations:

$$\mathbf{h}_t = \mathcal{F}(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}\,\mathbf{h}_{t-1}), \tag{2.4}$$

$$\begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} = \mathcal{G}(\mathbf{C}\,\mathbf{h}_t), \tag{2.5}$$

where $\mathcal{F}$ and $\mathcal{G}$ are two nonlinear functions (e.g. component-wise sigmoid functions).

2.1.2 Probabilistic models: Hidden Markov Models

The **Hidden Markov Model (HMM)** is a probabilistic linear model that has been developed in order to deal with sequential data. Most real-world information sources emit, at each time step t, observable events which are

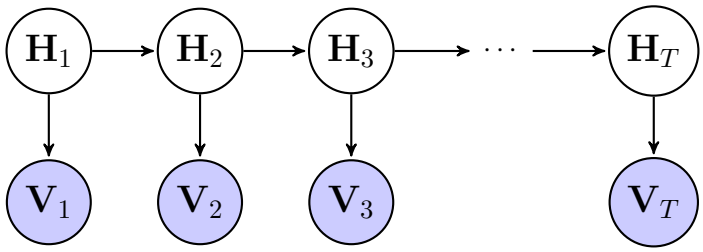

Figure 2.3: The graphical model of a first-order hidden Markov model with hidden states $\mathbf{H}_t$. The observable variables $\mathbf{V}_t$ are driven by the hidden states $\mathbf{H}_t$.

correlated with the internal state of the generating process. More importantly, the only available information is the outcome of the stochastic process at each time step t, i.e. event $\mathbf{v}^{(t)}$, while the state of the system is unobservable, i.e. *hidden*. Hidden Markov Models allow modeling general stochastic processes where the state transition dynamics is disentangled from the observable information generated by the process. The state-transition dynamics, which is non-observable, is modeled by a Markov chain of discrete and finite latent variables, i.e. the *hidden states*.

The dependency relationships among the different involved variables are typically represented by a graphical model, as exemplified for the HMM in Figure 2.3: the hidden states are latent variables $\mathbf{H}_t$, while the sequence elements $\mathbf{V}_t$ are observed. The conditional dependence represented by the arrow $\mathbf{H}_t \rightarrow \mathbf{V}_t$ indicates that the observed element at time t of the sequence is generated by the corresponding hidden state $\mathbf{H}_t$ through the *emission distribution*

$$\mathbf{b}_{\mathbf{h}^{(t)}}(\mathbf{v}^{(t)}) = P(\mathbf{V}_t = \mathbf{v}^{(t)} | \mathbf{H}_t = \mathbf{h}^{(t)}).$$

The joint distribution of the observed sequence $\mathbf{v} = \mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(T)}$ and associated hidden states $\mathbf{h} = \mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(T)}$, can be written as

$$P(\mathbf{V} = \mathbf{v}, \mathbf{H} = \mathbf{h}) = P(\mathbf{h}^{(1)}) \prod_{t=2}^{T} P(\mathbf{h}^{(t)} | \mathbf{h}^{(t-1)}) P(\mathbf{v}^{(t)} | \mathbf{h}^{(t)}). \tag{2.6}$$

The actual parametrization of the probabilities in eq. (2.6) depends on the form of the observation and hidden states variables. A stationary hidden states chain, with N states, is regulated by the $N \times N$ matrix of *state-transitions* $\mathbf{A}_{ij} = P(\mathbf{H}_t = i | \mathbf{H}_{t-1} = j)$ and by the N-dimensional vector of *initial state* probabilities $\pi_i = P(\mathbf{H}_t = i)$, where i, j are drawn from $\{1, \ldots, N\}$. Moreover, discrete sequence observations $\mathbf{v}_t \in \{1, \ldots, M\}$ (which is the case we are interested in here), the emission distribution is an $M \times N$ emission matrix $\mathbf{B}$ with elements

$$\mathbf{b}_i(k) = \mathbf{B}_{ki} = P(\mathbf{V}_t = k | \mathbf{H}_t = i). \tag{2.7}$$

The most common tasks performed by using an HMM are: *i)* to compute the most likely sequence of states given an observed sequence; *ii)* to train a model, which consists in finding the parameters (emission probabilities, transition probabilities, and initial state probability) that maximize the probability of the observed sequences contained in a training set, given the model. The first task is achieved by using the Viterbi algorithm [74], while training an HMM is usually performed by the Baum-Welch algorithm [92].

2.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) (Figure 2.4) are a particular type of neural networks designed to model sequential data. They are composed of three different layers of units: input layer, hidden layer, and output layer. At each time step, the corresponding elements of the sequence are given in input to the network through the input layer. The input layer is connected to the (first) hidden layer by a set of weighted connections. One or more hidden layers are used to encode, after training, the latent features of the data. The (last) hidden layer is then connected to the output layer, which is used for prediction of the desired output, e.g. in this thesis we are interested into the prediction, at time t, of the elements at time $t+1$ of the sequence. Let us consider a network with just one hidden layer. Let $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_l \in \mathbb{R}^n$ be the sequence of input vectors, $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_l \in \mathbb{R}^m$ the sequence of hidden states and $\mathbf{o}_1, \mathbf{o}_2, ..., \mathbf{o}_l \in \mathbb{R}^k$ the sequence of output states. The computation performed by the RNN at time t is described by the following equations:

$$\begin{aligned} \mathbf{h}_t &= \mathcal{F}(\mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{h}_{t-1} + \mathbf{b}_h), \\ \mathbf{o}_t &= \mathcal{F}(\mathbf{C}\mathbf{h}_t + \mathbf{b}_o), \end{aligned}$$

where $\mathcal{F}$ is a component-wise activation function, $\mathbf{b}_h$ and $\mathbf{b}_o$ are the biases of hidden and output units, and $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are the input-to-hidden, hidden-to-hidden and hidden-to-output weights, respectively. The standard way to perform training of RNNs parameters is via the back-propagation through time (BPTT) algorithm [93], either in batch mode or online via a stochastic gradient descent (SGD) approach. Unfortunately, RNNs encounter difficulties in learning when trained by the BPTT algorithm because of long-term temporal dependencies which lead to a vanishing gradient [12]. Two main possible remedies to this problem have been proposed: the first one is called Long Short-Term Memory (LSTM) [49] and it consists in extending the model by using special linear memory units, while a more recent proposal relies on an Hessian-Free optimization algorithm (HF) [62].

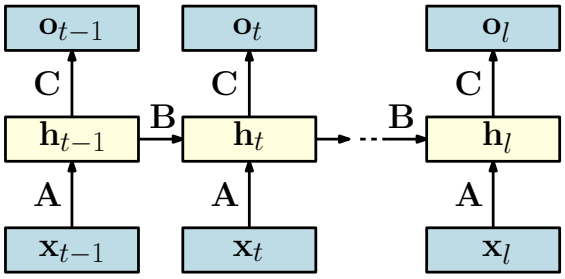

Figure 2.4: Schematic representation of the RNN.

2.1.4 Long Short-Term Memory

Long Short-Term Memory (LSTM) [49] is an extension of RNN network that has the ability of deal with long-term temporal dependencies, without incurring in the problem of vanishing gradient. The LSTM is designed in order to provide a mechanism that allows the networks to "remember" relevant information for a long period of time. This behavior is obtained by adding gate units to the structure of a simple RNN. The idea is to create a "cell state" where it is possible to add or remove information by using gates. Each gate cell consists in a sigmoidal function that regulates the information flow in the memory cell. In LSTM there are 3 gates. More formally an LSTM network is defined as follows:

$$\mathbf{g}_i = \sigma(\mathbf{A}^{(\mathbf{g}_i)}\mathbf{x}_t + \mathbf{B}^{(\mathbf{g}_i)}\mathbf{h}_{t-1}, \tag{2.8}$$

$$\mathbf{g}_f = \sigma(\mathbf{A}^{(\mathbf{g}_f)}\mathbf{x}_t + \mathbf{B}^{(\mathbf{g}_f)}\mathbf{h}_{t-1}, \tag{2.9}$$

$$\mathbf{g}_o = \sigma(\mathbf{A}^{(\mathbf{g}_o)}\mathbf{x}_t + \mathbf{B}^{(\mathbf{g}_o)}\mathbf{h}_{t-1}, \tag{2.10}$$

$$\mathbf{g} = \mathcal{F}(\mathbf{A}^{(\mathbf{g})}\mathbf{x}_t + \mathbf{B}^{(\mathbf{g})}\mathbf{h}_{t-1}, \tag{2.11}$$

$$\mathbf{c}_t = \mathbf{g}_f \cdot \mathbf{c}_{t-1} + \mathbf{g}_i \cdot \mathbf{g}, \tag{2.12}$$

$$\mathbf{h}_t = \mathbf{g}_o \cdot \mathcal{F}(\mathbf{c}_t), \tag{2.13}$$

where:

- $\mathbf{g}_i$, $\mathbf{g}_f$ and $\mathbf{g}_o$ are the input gate, the forget gate and the output gate, respectively. The equations that define them are very similar. Indeed, each of them uses different matrices. During the training, each gate is specialized to perform a special task. All gates use the same element-wise function: the sigmoid that narrows the real value, that a gate can take, in the range from 0 to 1. The values of these gates are multiplied by the vector that represents the input, the previous state and the candidate state in equations (2.11), (2.12) and (2.13). Therefore, the gate matrices regulate the quantity of information that passes through the memory cell. More precisely, the input gate defines how much information from the current input will be "saved" in the memory

cell. Indeed, in equation (2.12) its output is multiplied by $\mathbf{g}$ that is the candidate hidden state. The forget gate allows to define the quantity of information of previous state id used to compute the current state. Its output is multiplied by $\mathbf{c}_{t-1}$ (eq.(2.12)), that represents the state of the memory cell at the previous time step. Finally, the output gate $\mathbf{g}_o$ defines how much information stored in the cell is used to compute the current hidden state of the network (eq. (2.13)).

- $\mathbf{g}$ is computed by the same equation that is used by classic RNN to compute the hidden state. It may be considered as the candidate hidden state. Indeed, this hidden state is used to compute the LSTM hidden state, by mitigating it with the information coming from the memory cell.

- $\mathbf{c}_t$ is the memory cell that uses the information coming from the previous step and the information of the "base" hidden state filtered by the input and the forget gates, respectively.

- $\mathbf{h}_t$ is the hidden state of LSTM, that is computed by multiplying the value contained in the memory cell by the forget gate's value.

A simpler variant of LSTM is the Gated Recurrent Unit (GRU) [22], that merges the input and the forget gates in a single gate, called update gate. This gate uses two reset gates to manage how to combine the new input with the previous memory. The GRU has only two gates meaning that this model has fewer parameters than LSTM, but GRU does not have an internal memory cell. In literature, the comparison between these two models is discussed in [24] and [54], and the results show that the two models are comparable. Since the GRU has fewer parameters than LSTM, it has better performance in terms of time required for training. On the other hand, the greater expressive capability of LSTM allows it to better cope with complex tasks.

2.1.5 Restricted Boltzmann Machines (RBM)

Restricted Boltzmann Machines [79][46] (Figure 2.5) is a particular type of Markov Random field [55]. In particular, it consists of a twofold graphical model that contains two types of units: hidden units and visible units. This model is bipartite, indeed, there are no connections among visible units and among hidden units. This feature makes it different from the Boltzmann Machine (BM), that consists of a fully-connected probabilistic graphical model. The RBM weighted connections between the units and its biases, define a probability distribution by using an Energy-Based function. In general, Energy-Based Models use a scalar value, called Energy, that is associated with each variable configuration. Performing learning on this type of models

consists in "modifying" the energy function by tuning the model parameters. The Energy function modeled by an RBM is defined as follows:

$$E(\mathbf{v}, \mathbf{h}|\theta) = -\sum_{i=1}^{n_v}\sum_{j=1}^{n_h} \mathbf{w}_{ij}\mathbf{v}_i\mathbf{h}_j - \sum_{i=1}^{n_v} \mathbf{b}_{v_i}\mathbf{v}_i - \sum_{j=1}^{n_h} \mathbf{b}_{h_j}\mathbf{h}_j, \tag{2.14}$$

where $\mathbf{v}$ and $\mathbf{h}$ are respectively the vectors of visible and hidden units that have size n_v and n_h respectively. While $\mathbf{b}_v$ and $\mathbf{b}_h$ are the bias vectors of the visible and hidden layer, θ is the set of model parameters (weights and bias). The probability associated with visible units can be computed as follows:

$$P(\mathbf{v}|\theta) = \frac{\sum_h e^{-E(\mathbf{v},\mathbf{h})}}{\sum_u \sum_h e^{-E(\mathbf{u},\mathbf{h})}}. \tag{2.15}$$

The denominator of equation (2.15) is called partition function and it is defined in order to normalize to 1 the summation of the probability over all visible units. Since there are no connections between the hidden and the visible units, the conditional probability can be factorized as follows:

$$P(\mathbf{h}_j = 1|\mathbf{v}, \theta) = \sigma(-\mathbf{b}_{h_j} + \sum_{i=1}^{n_v} \mathbf{W}_{ij}\mathbf{v}_i) \tag{2.16}$$

$$P(\mathbf{v}_i = 1|\mathbf{h}, \theta) = \sigma(\mathbf{b}_{v_i} + \sum_{j=1}^{n_h} \mathbf{W}_{ij}\mathbf{h_j}). \tag{2.17}$$

The RBM turns out to be quite complex to train, mainly because during the training phase the goal is to maximize the model likelihood, and it is highly computational demanding. Indeed, it involves the computation of the function derivative. In order to solve this problem it is possible to exploit the following equation:

$$\frac{\partial \log P(\mathbf{v})}{\partial \mathbf{w}_{ij}} = < \mathbf{v}_i\mathbf{h}_j >_{data} - < \mathbf{v}_i\mathbf{h}_j >_{model}, \tag{2.18}$$

where $< \cdot >_d$ is the expected value over a probability distribution d. By using a gradient descent we can define the equation that will be used in order to update the model weights during the training phase:

$$\Delta\mathbf{w}_{ij} = \eta(< \mathbf{v}_i\mathbf{h}_j >_{data} - < \mathbf{v}_i\mathbf{h}_j >_{model}), \tag{2.19}$$

where η is called learning rate, that is typically set to a value in range from 0 to 1. In equation (2.19) it is easy to compute $< \mathbf{v}_i\mathbf{h}_j >_{data}$, since it can be obtained by considering the probability distribution of the training data. For what concerns $< \mathbf{v}_i\mathbf{h}_j >_{model}$, computation is more complex because it directly derives from the model, and therefore it requires to execute a Monte

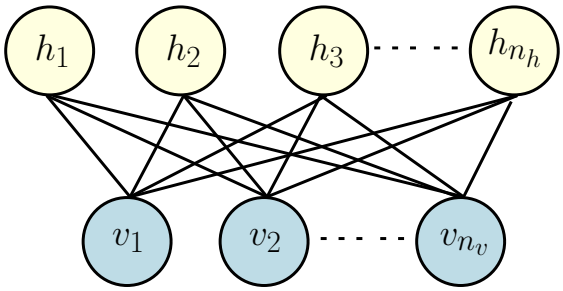

Figure 2.5: Schematic representation of the RBM.

Carlo Markov Chain [38] until the model converges. That happens because in this case we do not have any given data, and so research has to start from random data assigned to visible units, that is used to compute the hidden variables. The process has to be iterated until the model achieves stability. A more efficient alternative is called Contrastive Divergence (CD_k) [47]. This technique allows to compute an approximation of $< \mathbf{v}_i\mathbf{h}_j >_{model}$. The idea is to perform the sampling of hidden units values starting from the visible units values that directly derive from the data contained into the training set. Therefore, given the visible units values the idea is to use equation (2.17), in order to sample the hidden units values and then recompute the visible units values that maximize the probability of them given the computed hidden values. That technique allows to compute $< \mathbf{v}_i\mathbf{h}_j >_{reconstruction}$ that may be used in place of $< \mathbf{v}_i\mathbf{h}_j >_{model}$. The Contrastive Divergence is called CD_k because the sequence of operations presented above has to be performed k times. The interesting aspect is that empirical experiments show that it is possible to obtain good results with k equal to 1 [11].

2.1.6 Recurrent Neural Networks with Restricted Boltzmann Machines

The Recurrent Neural Network - Restricted Boltzmann Machine (RNN-RBM) [17] is a sequential neural network that combines the best features of RNNs, which are particularly effective in learning temporal dependencies, within an RBM, which can model complex and multi-modal distributions. The RNN-RBM network is similar to the RTRBM [85]. However, instead of exploiting a simple connection between the RBM hidden units of two contiguous time-steps, it hinges on the hidden units of an RNN to keep track of the relevant temporal information, thus allowing the encoding of long-term temporal dependencies.

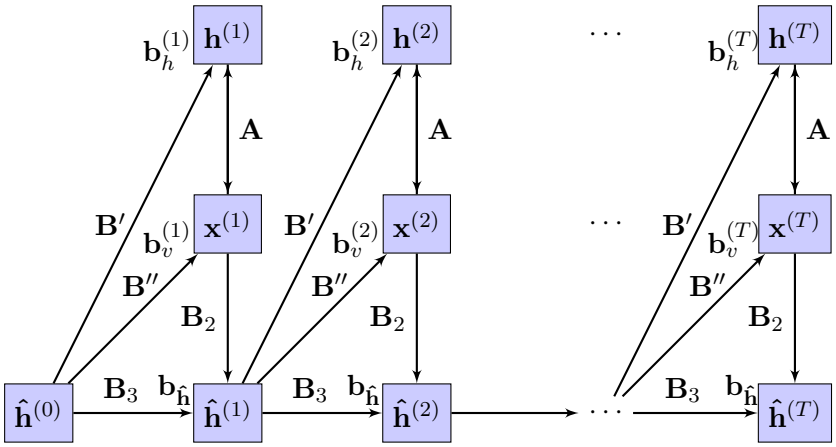

Figure 2.6: Schematic representation of the RNN-RBM (see [17] for details).

RNN-RBMs are nonlinear stochastic models, for which the joint probability distribution of hidden and input units is defined as:

$$P(\mathbf{x}_t, \mathbf{h}_t) = \prod_{t=1}^{T} P(\mathbf{x}_t, \mathbf{h}_t | \mathbf{x}_{t-1}, \mathbf{x}_{t-2}, ..., \mathbf{x}_1, \hat{\mathbf{h}}_{t-1}, \hat{\mathbf{h}}_{t-2}, ..., \hat{\mathbf{h}}_1)$$

where $\hat{\mathbf{h}}_t = \sigma(\mathbf{B}_2\mathbf{v}_t + \mathbf{B}_3\hat{\mathbf{h}}_{t-1} + \mathbf{b}_{\hat{\mathbf{h}}})$ and $\mathbf{x}_t$, $\mathbf{h}_t$ and $\hat{\mathbf{h}}_t$ represent, respectively, the input units, the RBM-hidden units and the RNN-hidden units, whereas $\mathbf{b}_{\hat{\mathbf{h}}}$ represents the RNN-hidden unit biases (see Figure 2.6). This type of networks are more complex than RNN and RTRBM networks, so they require an *ad-hoc* training algorithm. The idea is to propagate the value of hidden units $\hat{\mathbf{h}}_t$ in the RNN-part of the network and then to use the $\hat{\mathbf{h}}_t$ value to obtain the value of some of the parameters of the RBM-part. Specifically, time-variant biases for RBM are derived by the hidden units of the RNN, according to the following equations:

$$\begin{aligned} \mathbf{b}_{h_t} &= \mathbf{b}_{\hat{\mathbf{h}}} + \mathbf{B}'\hat{\mathbf{h}}_{t-1}, \\ \mathbf{b}_{v_t} &= \mathbf{b}_{\hat{\mathbf{h}}} + \mathbf{B}''\hat{\mathbf{h}}_{t-1}. \end{aligned}$$

The RBM-part of the network can then be trained by using Gibbs sampling and Contrastive Divergence [47]. In this way, the log-likelihood gradient of the RBM-part of the network can be estimated and propagated to all time steps by using the BPTT algorithm [93] in order to estimate the gradient with respect to the RNN-part parameters.

To obtain a good model of the data, a pre-training phase has to be performed. In particular, the authors of the RNN-RBM network developed an approach which consists in a separate pre-training for RBM and RNN over the training set. Specifically, pre-training of the RBM-part of the model is

performed by using the Contrastive Divergence algorithm [45] on information associated to single time steps, while the RNN-part of the model can be performed either by using SGD or Hessian-Free optimization, with the aim to better capture temporal dependencies.

2.1.7 Echo State Network

The **Echo State Network (ESN)** [52] is a nonlinear recurrent model that belongs to Reservoir Computing (RC) [61] framework. The model definition is similar to the normal RNN model [27] and formalized as:

$$\mathbf{h}_t = \mathcal{F}(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}\,\mathbf{h}_{t-1} + \mathbf{D}\,\mathbf{o}_{t-1}), \tag{2.20}$$

$$\mathbf{o}_t = \mathbf{C}\,\mathbf{h}_t. \tag{2.21}$$

The main difference w.r.t. RNN, regards how the models are trained. The idea behind the ESN model is to have a supervised and fast learning method, which exploits the RC principles: the first step is to randomly initialize input, hidden and feedback weights (matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$). In order to obtain good results [1], and ensure that the effect of initial conditions should vanish as time passes, these matrices must be initialized according to the Echo state Properties (ESP) [97]. ESP impose some rules on the weights and parameters initialization. In particular, they state that the spectral radius (ρ) of the matrices has to be ≤ 1. By tuning the value of ρ, it is possible to vary the memory length of the network reservoir. Moreover, other parameters need to be set in order to obtain good results with this particular type of network. For instance the range of values that can appear in the matrices (the maximum value and minimum value). By tuning these values the dynamic of the network can be changed and optimized. The second step in training phase consists in harvest all the reservoir states $\mathbf{h}_t$ computed on the input contained in the training set, and use them to compute the output weights $\mathbf{C}$ by performing a Linear Regression that minimizes the Mean Squared Error (MSE). The most stable technique is ridge regression also known as regression with *Tikhonov* regularization [60]. Let $\mathbf{H}$ be the matrix that contains all the $\mathbf{h}_t$ computed for each input $\mathbf{x}_t$, and $\mathbf{O}$ the matrix that collects all the corresponding supervised output. $\mathbf{C}$ is computed as:

$$\mathbf{C} = \mathbf{O}\mathbf{H}^{\mathsf{T}}(\mathbf{H}\mathbf{H}^{\mathsf{T}} + \beta\mathbf{I})^{-1}, \tag{2.22}$$

where $\mathbf{I}$ is the identity matrix and β is a regularization coefficient.Another common technique used in order to compute the output weights is the pseudoinverse:

$$\mathbf{C} = \mathbf{H}^{+}\mathbf{O}. \tag{2.23}$$

[1] The model is usually evaluated on the accuracy performance, whose evaluation is typically task specific. As an example, in order to perform prediction on polyphonic music, in this thesis we use the method presented in Section 6.1.3.

The direct pseudoinverse calculation typically exhibits high numerical stability. As a downside, it is expensive memory-wise for large matrices **H**.
Another important parameter that may be used in order to optimize the ESN model is the leaky-integrator. The ESN model that uses this parameter is defined by substituting equation (2.20) with:

$$\mathbf{h}_t = (1-a)\mathbf{h}_{t-1} + a\mathcal{F}(\mathbf{A}\,\mathbf{x}_t + \mathbf{B}\,\mathbf{h}_{t-1} + \mathbf{D}\,\mathbf{o}_{t-1}), \tag{2.24}$$

where a is the leaky-parameter. This parameter is related with the speed of reservoir dynamic [36]. A larger value of a allows the model to react in a faster way when the input changes.

2.2 The State-of-the-art in learning sequences

In this section we give a brief review of the most powerful Deep Learning models for sequences. These models are all able to deal with complex sequential data, and most of them are developed to solve a specific task. In particular, we consider the tasks related with the cognitive process. This choice stems from the fact that data and tasks related with this particular scope appear to be more complex to manage. Therefore, they allow to better study and highlight both strengths and weaknesses of various models.
The models that we present in this section derive from the models presented in Section 2.1. Indeed, models like LDS, RNN, and HMM, achieve good results in many complex tasks; however, recent results show that defining architectures that combine two or more of these models, or that extend them, allows to obtain better results in complex tasks.

2.2.1 Temporal RBM

An interesting model (that is also used as a building block to develop new architectures) is the **Temporal Restricted Boltzmann Machine** (**TRBM)** [85], an extension of Restricted Boltzmann Machine. The advantage of TRBM, respect to the normal RBM, is that it can deal with sequential data. The TRBM can be defined as a probabilistic graphical model that is composed of two types of units: visible units and hidden units. TRBM network defines a joint probability distribution over these units:

$$P(\mathbf{v}_{[1,\dots,T]}, \mathbf{h}_{[1,\dots,T]}) = \prod_{t=2}^{T} P(\mathbf{v}_t, \mathbf{h}_t|\mathbf{h}_{t-1})P(\mathbf{v}_1, \mathbf{h}_1), \tag{2.25}$$

where $\mathbf{v}_t$ and $\mathbf{h}_t$ are respectively the values of visible units and hidden units at time t, and $v_{[1,\dots,T]}$ is the vector that contains all values that $\mathbf{v}$ takes from time step 1 to T. The conditional distribution that derives from (2.25) can

be computed as follows:

$$P(\mathbf{v}_t, \mathbf{h}_t | \mathbf{h}_t - 1) = \frac{\exp(\mathbf{v}_t^\top \mathbf{b}_V + \mathbf{v}_t^\top \mathbf{W}_{vh}\mathbf{h}_t + \mathbf{h}_t(\mathbf{b}_H + \mathbf{W}_{hh}\mathbf{h}_{t-1}))}{\mathcal{Z}(\mathbf{h}_{t-1})}, \quad (2.26)$$

where $\mathbf{b}_V$ and $\mathbf{b}_H$ are respectively the visible and hidden bias, $\mathbf{W}_{vh}$ is the weights matrix that connects the visible and the hidden units and, $\mathbf{W}_{hh}$ is the weights matrix that connects the hidden units at time step t ($\mathbf{h}_t$) and the hidden units of the previous time step ($\mathbf{H}_{t-1}$). $\mathcal{Z}$ is called partition function and it is defined as in normal RBM [85]. In other words, the TRBM is a direct graphical model that has an undirected graphical model (RBM) at each time step. The training phase of this model consists in an inference problem that could be solved with Contrastive Divergence [87]. An extension of TRBM is the **Recursive TRBM (RTRBM)** [85] that allows to perform the inference in a more efficient way. Indeed, the RTRBM computes the values of $\mathbf{h}_t$ by sampling from the marginal distribution of Boltzmann Machine [6], which is involved in the exact ratio of two RBM partition functions. In order to solve this complexity issue the RTRBM uses an heuristic inference procedure which is based on a value $\mathbf{h}'_t$, which is not the exact result of sampling from the probability distribution $P(\mathbf{H}_t|\mathbf{v},_t, \mathbf{h}_{t-1})$, but a real value, so this is a "mean-field" update. Hence, the main difference between the TRBM and RTRBM is in the second step of inference, which makes the inference easier. Indeed, in RTRBM, there is only one unique hidden value that has a nonzero posterior probability, given the visible unit values. The main weakness of this model, however, is that it is not able to deal with long-term temporal dependencies in a satisfactory way.

2.2.2 DBN-based Model

Deep Belief Network (DBN) [48] is a generative graphical model composed of visible end hidden units. The DBN is a deep model composed of many hidden layers. This model is defined as a stack of RBM [11]. The most interesting feature of this model is its capability of extract different representations of the input at each layer. The higher is the deepness of the layer the higher the level of abstraction of the computed representation. The most common issue with this model is the complexity of the training phase performed by using the Back-propagation algorithm [20]. This problem is due to the deepness of the model. Indeed, if the DBN has many hidden layers, the back propagation turns out to be ineffective. The breakthrough that allows to train DBN in a reasonable time and with good results is the introduction of greedy layer-wise pre-training phase. The idea is to train each couple of DBN layers as if they were an RBM. For the first couple of layers (starting from the lowest one) the layer-wise pre-training technique considers them as an RBM and uses the training set as input in order to train the weights between the layers. For the subsequent couples of layers,

the output of previous RBM is used as input.
The DBN can represent the following probability distribution:

$$P(\mathbf{v}, \mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_l) = P(\mathbf{h}_{l-1}, \mathbf{h}_l) \cdot \prod_{k=0}^{l-2} P(\mathbf{h}_k | \mathbf{h}_{k+1}), \tag{2.27}$$

where $\mathbf{v}$ is the input layer, $\mathbf{h}_i$ is the i-th hidden layers of the model. Moreover in the second part of the equation $\mathbf{h}_0 = \mathbf{v}$.
A possible solution that exploits DBN to perform learning in sequential domains is to use a static network, and divide the input sequence into parts, i.e., performing the so-called "windowing". The idea is to split the input into two contiguous parts, in a way that they are overlapped for a small region, and use the obtained parts as input of a DBN. A more powerful technique is the combination of DBN models with a sequential model such as the Conditional Random field (CRF) [98]. An interesting model, derived by the application of the DBN model to sequential domains, is the new model called Temporal Sigmoid Belief Network (TSBL) [37]. This particular model consists in a DBN where the activation function is a sigmoid function, and where recurrent connections are inserted on the hidden layer:

$$P(\mathbf{V}, \mathbf{H}) = P(\mathbf{h}_1) P(\mathbf{v}_1 | \mathbf{h}_1) \cdot \prod_{t=2}^{T} P(\mathbf{h}_t | \mathbf{h}_{t-1}, \mathbf{v}_{t-1}) \cdot P(\mathbf{v}_t | \mathbf{h}_t, \mathbf{v}_{t-1}). \tag{2.28}$$

This model has been applied to many different prediction tasks, e.g., prediction on motion capture dataset (that contains several measures of joint angles for different motion types), prediction on polyphonic music dataset, prediction on bouncing ball dataset [37].

2.2.3 Bidirection-RNN

An interesting RNN based model is the **Bidirectional RNN (BRNN)** [78]. This model has been developed for a particular task: off-line text recognition. In this particular case, all sequences are visible during each time step. BRNN exploits also the future context in order to compute the output, by processing the data in both directions with two separate hidden layers that are connected to the same output layer. The two following equations define the forward and the backward hidden layers:

$$\mathbf{h}_t^F = \mathcal{F}_F(\mathbf{A}^F \mathbf{x}_t + \mathbf{B}^F \mathbf{h}_{t-1}^F + \mathbf{b}_h^F), \tag{2.29}$$

$$\mathbf{h}_t^B = \mathcal{F}_B(\mathbf{A}^B \mathbf{x}_t + \mathbf{B}^B \mathbf{h}_{t+1}^B + \mathbf{b}_h^B), \tag{2.30}$$

where $\mathbf{A}^F$ and $\mathbf{A}^B$ are the weights that connect the input layer respectively to Forward and Backward hidden layers, while $\mathbf{B}^F$ and $\mathbf{B}^B$ are the recurrent

weights for the two hidden layers. $\mathbf{b}_h^F$ and $\mathbf{b}_h^B$ are the bias.
The output is computed by using the results of both hidden layers:

$$\mathbf{o}_t = \mathcal{F}_o(\mathbf{C}^F\mathbf{h}_t^F + \mathbf{C}^B\mathbf{h}_t^B + \mathbf{b}_o), \tag{2.31}$$

where C^F and C^B are the output weights that respectively connect the forward hidden layer and the backward hidden layer to the output layer, and $\mathbf{b}_o$ is the output layer bias. In these equations $\mathcal{F}_F$, $\mathcal{F}_B$ and $\mathcal{F}_O$ are the activation function of the various layers, and they usually use logistic function or the hyperbolic tangent function. BRNN share the problem of vanishing gradient with the standard RNN. In order to solve this problem in [40] Graves et al. developed a Bidirectional version of the Long Short-Term Memory network that allows to deal with long sequences without the risk of vanishing/exploding gradient. Unfortunately, the bi-directionality of this type of network introduces issues in the training phase: even when the network processes the first inputs of the sequence, the backward layer has potentially to manage long-term dependencies. Therefore, solving these problem turns out to be crucial in order to apply BRNN in real world scenario.
BRNN networks are developed to deal with complex data like text or audio, therefore, it is important to ensure that the model is enough powerful to extract the relevant information from these types of data. For this reason in [39] a version that exploits a deep RNN is developed. In this case, the model will have several hidden layers. Equations (2.29) and (2.30) are substituted by:

$$\mathbf{h}_t^{F(i)} = \mathcal{F}_F(\mathbf{A}^{F(i)}\mathbf{h}_t^{F(i)} + \mathbf{B}^{F(i)}\mathbf{h}_{t-1}^{F(i)} + \mathbf{b}_h^{F(i)}), \tag{2.32}$$

$$\mathbf{h}_t^{B(i)} = \mathcal{B}_F(\mathbf{A}^{F(i)}\mathbf{h}_t^{B(i)} + \mathbf{B}^{B(i)}\mathbf{h}_{t+1}^{B(i)} + \mathbf{b}_h^{B(i)}), \tag{2.33}$$

where i is the considered layer and $\mathbf{h}_t^{F(0)} = \mathbf{h}_t^{B(0)} = \mathbf{x}_t$. The BRNN network is trained by using the BPTT algorithm.

2.2.4 Multiplicative-RNN

Another model that directly derives from RNN is the **Multiplicative-RNN (MRNN)** [86]. This model has been developed in order to deal with tasks where the input has a great influence on the hidden state values. For instance, in text recognition/prediction, where the input is a single character, a single input could change a lot the hidden representation. For instance, consider a RNN that already had as input the three characters: "f","i", and "x". Now, the output of the network that has to predict the next char, has to model a probability distribution where "fix" is considered a root of a verb (e.g. fixing, fixed, etc.) or a world itself. It is easy to understand that the next input character will greatly vary the probability distribution of the network output. Indeed, if the next char is "i", the probability that

the next output will be "n" have to dramatically increase. Otherwise, if the next character is "e", the probability of "d" will increase a lot. Therefore, it is obvious that a single char has the capability to deeply modify the probability distribution of the output.
The output of the network is based on the hidden representation, which in turn depends on the input and on the previous hidden state. The main problem is that, as can be noticed in equation (2.8) the input is projected in the state space by using the matrix $\mathbf{A}$ and the result is added to the previous hidden state transformed by the matrix $\mathbf{B}$. MRNN aim is to create a multiplicative interaction between the input and the hidden state. In order to implement this idea, a matrix $\mathbf{B}^{(x_t)}$ is introduced. The weights represented by the matrix $\mathbf{B}^{(x_t)}$ are influenced by the input $\mathbf{x}_t$. In order to compute $\mathbf{B}^{(x_t)}$ in an efficient way 3 matrices are introduced: $\mathbf{B}_x$, $\mathbf{B}_h$, and $\mathbf{B}_f$. Now we can define $\mathbf{B}^{(x_t)}$:

$$\mathbf{B}^{(x_t)} = \mathbf{B}_h \cdot diag(\mathbf{B}_x \cdot \mathbf{x}_t) \cdot \mathbf{B}_f, \tag{2.34}$$

Where $diag(\cdot)$ is the function that compute a square diagonal matrix with the elements of a vector. In equation (2.8) $\mathbf{B}$ is substituted by $\mathbf{B}^{(x_t)}$:

$$\mathbf{h}_t = \sigma(\mathbf{A}\mathbf{x}_t + \mathbf{B}^{(x_t)}\mathbf{h}_{t-1} + \mathbf{b}_h). \tag{2.35}$$

The equation shows that MRNN has two steps of nonlinear processing in its hidden states for every input. The main drawback of this type of network is that the multiplicative contribution of the input $\mathbf{x}_t$ in computing the $\mathbf{h}_t$ makes gradient descent learning complicated. This problem is well handled by the Hessian-Free optimization. This model obtained impressive results on text prediction, and achieves also very good results as a generative model on text generation tasks [86].

Chapter 3

Linear Dynamic System for Sequence Prediction

In the last few years several different models able to face the problem of performing prediction on sequential data were introduced. The largest part of these models is based on RNN or other Deep Learning models. The considered sequential data are usually very complex. In particular, performing prediction tasks on cognitive data (e.g. speech, text, video, etc.), has been proven to be a tough challenge. Thanks to the capability of Deep Learning models of dealing with complex data, performing prediction on complex data has become possible and many models obtain very good results. Unfortunately, the use of models inspired by Deep Learning have also some drawbacks. In particular, training and application of these models require a great computational and time effort. For this reason, we decided to explore how to solve these problems by using simpler models. Our exploration starts by considering the most simple model for dealing with sequences: the Linear Dynamic System (defined in Section 2.1.1). In this chapter, we firstly define the problem of sequence prediction and then we explain how it is possible to use a LDS in order to deal with complex data, by introducing three different training techniques.

3.1 Prediction Task on Sequential Data

Prediction tasks for sequences usually consist in predicting, at time t, the input at time step $t + i$, $i \geq 1$ given all previous time step inputs of a sequence. More formally, we would like to learn a function $\mathcal{F}(\cdot)$ from multivariate bounded length input sequences to desired output values. Specifically, given a training set $\mathcal{T} = \{(\mathbf{s}^q, \mathbf{d}^q) \mid q = 1, \ldots, N,\ \mathbf{s}^q \equiv (\mathbf{x}_1^q, \mathbf{x}_2^q, \ldots, \mathbf{x}_{l_q}^q),$

$\mathbf{d}^q \equiv (\mathbf{d}_1^q, \mathbf{d}_2^q, \ldots, \mathbf{d}_{l_q}^q),\ \mathbf{x}_t^q \in \mathbb{R}^n,\ \mathbf{d}_t^q \in \mathbb{R}^s\}$, we wish to learn a function $\mathcal{F}(\cdot)$ such that $\forall q, t\ \mathcal{F}(\mathbf{s}^q[1,t]) = \mathbf{d}_t^q$, where $\mathbf{s}^q[1,t] \equiv (\mathbf{x}_1^q, \mathbf{x}_2^q, \ldots, \mathbf{x}_t^q)$. Notice that the prediction task we are interested in is a special case in which $\mathbf{d}_k^q = \mathbf{x}_{k+1}^q$.

3.2 Training Method for LDS

Since we want to deal with complex, and usually high dimensional data, we firstly explore how to tune a LDS in order to make it capable to perform the prediction task. As we already said in Section 2.1.1 this particular model is already used to perform a task similar to prediction, and the **Identification System Method** [59] could be applied in order to tune the parameters of the system in order to make it capable to output the desired results given an input. But these methods are developed in order to compute the parameters that allow to model the behavior of a physical system (that is usually described by a single very long sequence), and compute an exact solution. So they are not able to deal with a complete dataset, and moreover the Identification System methods do not have the capability to generalize, that is a fundamental part of a learning task. For these reasons we propose here three alternative approaches, to train the LDS model. All these methods allow to train the model over a dataset of sequences. Two of these approaches are inspired by pre-existing training methods for common models like **ESN** or **RNN**, the other is a method that allows to compute a closed form, approximated solution for a particular instance of LDS. In each case, the aim is to minimize the error function $E_{\mathcal{T}}$

$$E_{\mathcal{T}} = \frac{1}{NL} \sum_{q=1}^{N} \sum_{j=1}^{l_q} (\mathbf{d}_j^q - \mathbf{o}_j^q)^2, \tag{3.1}$$

where $L = \sum_{q=1}^{N} l_q$. In the following these methods, that we named $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ are briefly summarized:

- $\mathcal{L}_1$: Adopt the Echo State Network-like training procedure that randomly initializes matrices $\mathbf{A}$ and $\mathbf{B}$ according to some property that ensure the effectiveness of the initialization, and only trains the output weights using pseudo-inverse of the hidden representations.

- $\mathcal{L}_2$: Perform the training by considering the LDS as an Autoencoder, initializing matrices $\mathbf{A}$ and $\mathbf{B}$ according to the procedure that allows to compute a closed form solution for the linear auto-encoder, and $\mathbf{C}$ with $\mathbf{D}\tilde{\mathbf{H}}^+$, where $\tilde{\mathbf{H}}^+$ is the pseudo-inverse of matrix $\tilde{\mathbf{H}}$ and $\tilde{\mathbf{H}}$ is obtained by first running equation (4.1) with initialized matrices $\mathbf{A}$ and $\mathbf{B}$.

- $\mathcal{L}_3$: Perform SGD with respect to the regularized error function $\tilde{E}_{\mathcal{T}}$ (eq. 3.2), with standard random initialization for matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$.

This corresponds to a standard RNN training procedure. the regularized error function $\tilde{E}_{\mathcal{T}}$ is defined as follows:

$$\tilde{E}_{\mathcal{T}} = \frac{1}{NL}\sum_{q=1}^{N}\sum_{j=1}^{l_q}(\mathbf{d}_j^q - \mathbf{o}_j^q)^2 + R_1 + R_2, \tag{3.2}$$

where $L = \sum_{q=1}^{N} l_q$, and

$$R_1 = \sum_{i}^{m}\sum_{j}^{n}|\mathbf{A}_{ij}| + \sum_{i}^{m}\sum_{j}^{m}|\mathbf{B}_{ij}| + \sum_{i}^{s}\sum_{j}^{m}|\mathbf{C}_{ij}|,$$
$$R_2 = \sum_{i}^{m}\sum_{j}^{n}\mathbf{A}_{ij}^2 + \sum_{i}^{m}\sum_{j}^{m}\mathbf{B}_{ij}^2 + \sum_{i}^{s}\sum_{j}^{m}\mathbf{C}_{ij}^2.$$

3.2.1 Method $\mathcal{L}_1$

This method consists in initializing matrices $\mathbf{A}$ and $\mathbf{B}$ as in ESN training. A relevant issue is how to generate matrices $\mathbf{A}$ and $\mathbf{B}$. This issue has already be addressed in ESN. Indeed, in order to avoid problems in computing the system state and ensure good results, a set of rules to follow for random matrix initialization has been proposed. This set of rules is called Echo State Property [51], and in particular, they prescribe to ensure that the randomly initialized matrices have *spectral radius* ρ less than or equal to 1. Unfortunately computing the spectral radius of large matrices is computationally demanding, so we use a much faster approach where we require $\mathbf{A}$ and $\mathbf{B}$ to have norm $\|\cdot\|$ (either L1-norm or L2-norm) less than or equal to 1. Since for any symmetric matrix $\mathbf{M}$, $\rho(\mathbf{M}) \leq \|\mathbf{M}\|$, in this way the Echo State Property is preserved.

Theorem 1. *Let's* $\mathbf{M}$ *be a random symmetric matrix. If* $\|\mathbf{M}\| \leq 1$ *then* $\rho(\mathbf{M})$ *will be* ≤ 1.

Proof. let's consider an eigenvalue λ of $\mathbf{M}$; then exists a not null vector $\mathbf{f}$ s.t. $\mathbf{Mf} = \lambda\mathbf{f}$. Let's consider the matrix $\mathbf{F} = [\mathbf{f}, \mathbf{f}, \ldots, \mathbf{f}]$; then $\mathbf{MF} = \lambda\mathbf{F}$, and so

$$|\lambda|\,\|\mathbf{F}\| = \|\lambda\mathbf{F}\| = \|\mathbf{MF}\| \leq \|\mathbf{M}\|\,\|\mathbf{F}\|\,. \tag{3.3}$$

Since $\mathbf{F}$ is not null we can obtain $|\lambda| \leq \|\mathbf{M}\|$, and since $\mathbf{f}$ is an arbitrary eigenvector of $\mathbf{M}$, we can conclude that $\rho(\mathbf{M}) \leq |\lambda| \leq \|\mathbf{M}\|$. □

In practice, experimental results have shown that it is not necessary to use symmetric matrices.
$\mathbf{C}$, the output matrix, is defined by using the pseudo-inverse. The method consists in computing all hidden representations $\mathbf{h}_t^q$ for each input sequence $\mathbf{s}^q$ contained in the training set $\mathcal{T}$ by using eq. (2.1). The aim is to create the

matrix $\mathbf{H} = [\mathbf{h}_1^1, \mathbf{h}_2^1, \ldots, \mathbf{h}_{l_N}^N]$ that is the matrix that collects all the hidden representations obtained by running the system, with random matrices $\mathbf{A}$ and $\mathbf{B}$, over all sequences in the training set. The matrix $\mathbf{C}$ is computed as follows:

$$\mathbf{C} = \mathbf{D}\mathbf{H}^+, \tag{3.4}$$

where $\mathbf{D}$ is the matrix containing all targets occurring in the training set: $\mathbf{D} = [\mathbf{d}_1^1, \mathbf{d}_2^1, \ldots, \mathbf{d}_{l_N}^N]$.

3.2.2 Method $\mathcal{L}_2$

This training method exploits the similarity of the structure of a LDS and a Linear Autoencoder. Indeed, the idea is to compute the optimal weights of a linear Autoencoder given the training set of sequences. The weights obtained for the Linear Autoencoder are then used as weights for the input to hidden, and hidden to hidden connections of a LDS. In the following we explain how to find a closed form solution for a Linear Autoencoder in a fast and convenient way.
In [18, 9] it is shown that principal directions of a set of vectors $\mathbf{x}_i \in \mathbb{R}^n$ are related to solutions obtained by training Linear Autoencoder networks

$$\mathbf{o}_i = \mathbf{C} \cdot \mathbf{A}\mathbf{x}_i, \ i = 1, \ldots, l, \tag{3.5}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{C} \in \mathbb{R}^{n \times m}$, $m \ll n$, and the network is trained so to get $\mathbf{o}_i = \mathbf{x}_i, \ \forall i$. When considering a temporal sequence $\mathbf{s} = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots$ of input vectors, where t is a discrete time index, a Linear Autoencoder can be defined by considering the coupled linear dynamical systems,

$$\mathbf{h}_t = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{h}_{t-1}, \tag{3.6}$$

$$\begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} = \mathbf{C}\mathbf{h}_t. \tag{3.7}$$

It should be noticed that eqs. (3.6) and (3.7) extend the linear transformation defined in eq. (3.5) by introducing a *memory term* involving matrix $\mathbf{B} \in \mathbb{R}^{m \times m}$. In fact, $\mathbf{h}_{t-1}$ is inserted in the right part of equation (3.6) to keep track of the input history through time: this is done exploiting a state space representation. Eq. (3.7) represents the decoding part of the Autoencoder: when a state $\mathbf{h}_t$ is multiplied by $\mathbf{C}$, the observed input $\mathbf{x}_t$ at time t and state at time $t-1$, i.e. $\mathbf{h}_{t-1}$, are generated. Decoding can then continue from $\mathbf{h}_{t-1}$. This formulation has been proposed, for example, in [90] where an iterative procedure to learn weight matrices $\mathbf{A}$ and $\mathbf{B}$, based on Oja's rule, is presented. No proof of convergence for the proposed procedure is

however given. More recently, an exact closed-form solution for the weights has been given in the case of a number of hidden units equal to the rank of the full data matrix (full rank case) [80, 82]. In this section, we revise this result. In addition, we give an exact solution also for the case in which the number of hidden units is strictly less than the rank of the full data matrix.

The basic idea of [80, 82] is to look for directions of high variance into the *state space* of the dynamical linear system (3.6). Let start by considering a single sequence $\mathbf{s} \equiv (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_l)$ and the state vectors of the corresponding induced state sequence collected as rows of a matrix $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3, \cdots, \mathbf{h}_l]^\mathsf{T}$. By using the initial condition $\mathbf{h}_0 = \mathbf{0}$ (the null vector), and the dynamical linear system (3.6), we can rewrite the $\mathbf{H}$ matrix as

$$\mathbf{H} = \underbrace{\begin{bmatrix} \mathbf{x}_1^\mathsf{T} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{x}_2^\mathsf{T} & \mathbf{x}_1^\mathsf{T} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{x}_3^\mathsf{T} & \mathbf{x}_2^\mathsf{T} & \mathbf{x}_1^\mathsf{T} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{x}_l^\mathsf{T} & \mathbf{x}_{l-1}^\mathsf{T} & \mathbf{x}_{l-2}^\mathsf{T} & \cdots & \mathbf{x}_2^\mathsf{T} & \mathbf{x}_1^\mathsf{T} \end{bmatrix}}_{\boldsymbol{\Xi}} \underbrace{\begin{bmatrix} \mathbf{A}^\mathsf{T} \\ \mathbf{A}^\mathsf{T}\mathbf{B}^\mathsf{T} \\ \mathbf{A}^\mathsf{T}\mathbf{B}^{2\mathsf{T}} \\ \vdots \\ \mathbf{A}^\mathsf{T}\mathbf{B}^{l-1\mathsf{T}} \end{bmatrix}}_{\boldsymbol{\Omega}}$$

where, given $k = nl$, $\boldsymbol{\Xi} \in \mathbb{R}^{l \times k}$ is a data matrix collecting all the (inverted) input subsequences (including the whole sequence) as rows, and $\boldsymbol{\Omega}$ is the parameter matrix of the dynamical system. Now, we are interested in using a state space of dimension $p \ll l$, i.e. $\mathbf{h}_t \in \mathbb{R}^p$, such that as much information as contained in $\boldsymbol{\Xi}$ is preserved. We start by factorizing $\boldsymbol{\Xi}$ using SVD, obtaining $\boldsymbol{\Xi} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{U}^\mathsf{T}$ where $\mathbf{V} \in \mathbb{R}^{l \times l}$ is an unitary matrix, $\boldsymbol{\Lambda} \in \mathbb{R}^{l \times k}$ is a rectangular diagonal matrix with nonnegative real numbers on the diagonal with $\lambda_{1,1} \geq \lambda_{2,2} \geq \cdots \geq \lambda_{l,l}$ (the singular values), and $\mathbf{U}^\mathsf{T} \in \mathbb{R}^{k \times k}$ is a unitary matrix. It is important to notice that columns of $\mathbf{U}^\mathsf{T}$ which correspond to nonzero singular values, apart some mathematical technicalities, basically correspond to the principal directions of data, i.e. PCA. If the rank of $\boldsymbol{\Xi}$ is p, then only the first p elements of the diagonal of $\boldsymbol{\Lambda}$ are not null, and the above decomposition can be reduced to $\boldsymbol{\Xi} = \mathbf{V}^{(p)}\boldsymbol{\Lambda}^{(p)}{\mathbf{U}^{(p)}}^\mathsf{T}$ where $\mathbf{V}^{(p)} \in \mathbb{R}^{l \times p}$, $\boldsymbol{\Lambda}^{(p)} \in \mathbb{R}^{p \times p}$, and ${\mathbf{U}^{(p)}}^\mathsf{T} \in \mathbb{R}^{p \times k}$. Now we can observe that ${\mathbf{U}^{(p)}}^\mathsf{T}\mathbf{U}^{(p)} = \mathbf{I}$ (where $\mathbf{I}$ is the identity matrix of dimension p), since by definition the columns of $\mathbf{U}^{(p)}$ are orthogonal, and by imposing $\boldsymbol{\Omega} = \mathbf{U}^{(p)}$, we can derive "optimal" matrices $\mathbf{A} \in \mathbb{R}^{p \times h}$ and $\mathbf{B} \in \mathbb{R}^{p \times p}$ for our dynamical system, which will have corresponding state space matrix $\mathbf{Y}^{(p)} = \boldsymbol{\Xi}\boldsymbol{\Omega} = \boldsymbol{\Xi}\mathbf{U}^{(p)} = \mathbf{V}^{(p)}\boldsymbol{\Lambda}^{(p)}{\mathbf{U}^{(p)}}^\mathsf{T}\mathbf{U}^{(p)} = \mathbf{V}^{(p)}\boldsymbol{\Lambda}^{(p)}$. Thus, if we represent $\mathbf{U}^{(p)}$ as composed of n submatrices $\mathbf{U}_i^{(p)}$, each of size $h \times p$, the problem

reduces to find matrices $\mathbf{A}$ and $\mathbf{B}$ such that

$$\mathbf{\Omega} = \begin{bmatrix} \mathbf{A}^\mathsf{T} \\ \mathbf{A}^\mathsf{T}\mathbf{B}^\mathsf{T} \\ \mathbf{A}^\mathsf{T}\mathbf{B}^{2\mathsf{T}} \\ \vdots \\ \mathbf{A}^\mathsf{T}\mathbf{B}^{l-1\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathbf{U}_1^{(p)} \\ \mathbf{U}_2^{(p)} \\ \mathbf{U}_3^{(p)} \\ \vdots \\ \mathbf{U}_l^{(p)} \end{bmatrix} = \mathbf{U}^{(p)}. \tag{3.8}$$

The reason to impose $\mathbf{\Omega} = \mathbf{U}^{(p)}$ is to get a state space where the coordinates are uncorrelated so to diagonalise the empirical sample covariance matrix of the states. Please, note that in this way each state (i.e., row of the $\mathbf{H}$ matrix) corresponds to a row of the data matrix $\mathbf{\Xi}$, i.e. the unrolled (sub)sequence read up to a given time t. If the rows of $\mathbf{\Xi}$ were vectors, this would correspond to compute PCA, keeping only the fist p principal directions.
In the following, we demonstrate that there exists a solution to the above equation. We start by observing that $\mathbf{\Xi}$ owns a special structure, i.e. given $\mathbf{\Xi} = [\mathbf{\Xi}_1 \ \mathbf{\Xi}_2 \ \cdots \ \mathbf{\Xi}_l]$, where $\mathbf{\Xi}_i \in \mathbb{R}^{l \times n}$, then for $i = 1, \ldots, l-1$, $\mathbf{\Xi}_{i+1} = \mathbf{R}_l \mathbf{\Xi}_i = \begin{bmatrix} \mathbf{0}_{1\times(l-1)} & \mathbf{0}_{1\times 1} \\ \mathbf{I}_{(l-1)\times(l-1)} & \mathbf{0}_{(l-1)\times 1} \end{bmatrix} \mathbf{\Xi}_i$, and $\mathbf{R}_l \mathbf{\Xi}_l = \mathbf{0}$, i.e. the null matrix of size $l \times n$. Moreover, by singular value decomposition, we have $\mathbf{\Xi}_i = \mathbf{V}^{(p)} \mathbf{\Lambda}^{(p)} {\mathbf{U}_i^{(p)}}^\mathsf{T}$, for $i = 1, \ldots, l$. Using the fact that ${\mathbf{V}^{(p)}}^\mathsf{T} \mathbf{V}^{(p)} = \mathbf{I}$, and combining the above equations, we get $\mathbf{U}_{i+t}^{(p)} = \mathbf{U}_i^{(p)} \mathbf{Q}^t$, for $i = 1, \ldots, l-1$, and $t = 1, \ldots, l-i$, where $\mathbf{Q} = \mathbf{\Lambda}^{(p)} {\mathbf{V}^{(p)}}^\mathsf{T} \mathbf{R}_l^\mathsf{T} \mathbf{V}^{(p)} {\mathbf{\Lambda}^{(p)}}^{-1}$. Moreover, we have that $\mathbf{U}_l^{(p)} \mathbf{Q} = \mathbf{0}$ since $\mathbf{U}_l^{(p)} \mathbf{Q} = \mathbf{U}_l^{(p)} \mathbf{\Lambda}^{(p)} {\mathbf{V}^{(p)}}^\mathsf{T} \mathbf{R}_l^\mathsf{T} \mathbf{V}^{(p)} {\mathbf{\Lambda}^{(p)}}^{-1} = (\underbrace{\mathbf{R}_l \mathbf{\Xi}_l}_{=\mathbf{0}})^\mathsf{T} \mathbf{V}^{(p)} {\mathbf{\Lambda}^{(p)}}^{-1}$. Thus, eq. (3.8) is satisfied by $\mathbf{A} = {\mathbf{U}_1^{(p)}}^\mathsf{T}$ and $\mathbf{B} = \mathbf{Q}^\mathsf{T}$. It is interesting to note that the original data $\mathbf{\Xi}$ can be recovered by computing $\mathbf{H}^{(p)} {\mathbf{U}^{(p)}}^\mathsf{T} = \mathbf{V}^{(p)} \mathbf{\Lambda}^{(p)} {\mathbf{U}^{(p)}}^\mathsf{T} = \mathbf{\Xi}$, which can be achieved by running the system

$$\begin{bmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\mathsf{T} \\ \mathbf{B}^\mathsf{T} \end{bmatrix} \mathbf{h}_t$$

starting from $t = l$, i.e. $\begin{bmatrix} \mathbf{A}^\mathsf{T} \\ \mathbf{B}^\mathsf{T} \end{bmatrix}$ is the matrix $\mathbf{C}$ defined in eq. (3.7).

Finally, it is important to remark that the above construction works not only for a single sequence, but also for a set of sequences of different length. For example, let consider the two sequences $S_1 = (\mathbf{x}_1^1, \mathbf{x}_2^1, \mathbf{x}_3^1)$ and $S_2 = (\mathbf{x}_1^2, \mathbf{x}_2^2)$. Then, we have

$$\mathbf{\Xi}_{\mathbf{S_1}} = \begin{bmatrix} {\mathbf{x}_1^1}^\mathsf{T} & \mathbf{0} & \mathbf{0} \\ {\mathbf{x}_2^1}^\mathsf{T} & {\mathbf{x}_1^1}^\mathsf{T} & \mathbf{0} \\ {\mathbf{x}_3^1}^\mathsf{T} & {\mathbf{x}_2^1}^\mathsf{T} & {\mathbf{x}_1^1}^\mathsf{T} \end{bmatrix} \quad \text{and} \quad \mathbf{\Xi}_{\mathbf{S_2}} = \begin{bmatrix} {\mathbf{x}_1^2}^\mathsf{T} & \mathbf{0} \\ {\mathbf{x}_2^2}^\mathsf{T} & {\mathbf{x}_1^2}^\mathsf{T} \end{bmatrix}$$

which can be collected together to obtain $\boldsymbol{\Xi} = \left[\begin{array}{c|cc} \boldsymbol{\Xi}_{\mathbf{S_1}} & & \\ \hline \boldsymbol{\Xi}_{\mathbf{S_2}} & \mathbf{0}_{2\times 1} & \end{array}\right]$, and $\mathbf{R} = \left[\begin{array}{c|cc} \mathbf{R}_4 & & \\ \hline \mathbf{R}_2 & \mathbf{0}_{2\times 1} & \end{array}\right]$.

The following lemma shows the link between matrix $\mathbf{Q}$ and the principal directions corresponding to matrix $\mathbf{U}^{(p)}$

Lemma *(Relationship with Principal Directions)*

$$\mathbf{Q} = \boldsymbol{\Lambda}^{(p)}{\mathbf{V}^{(p)}}^{\mathsf{T}}\mathbf{R}_l^{\mathsf{T}}\mathbf{V}^{(p)}{\boldsymbol{\Lambda}^{(p)}}^{-1} = {\mathbf{U}^{(p)}}^{\mathsf{T}}\mathbf{R}_{k,n}^{\mathsf{T}}\mathbf{U}^{(p)} = \sum_{i=1}^{l-1}{\mathbf{U}_i^{(p)}}^{\mathsf{T}}\mathbf{U}_{i+1}^{(p)}$$

where $\mathbf{R}_{k,n}^{\mathsf{T}} = \begin{bmatrix} \mathbf{0}_{(k-n)\times k} & \mathbf{I}_{(k-n)\times(k-n)} \\ \mathbf{0}_{n\times n} & \mathbf{0}_{n\times(k-n)} \end{bmatrix}$.

Proof:

By definition $\sum_{i=1}^{l}{\mathbf{U}_i^{(p)}}^{\mathsf{T}}\mathbf{U}_i^{(p)} = \mathbf{I}$ and

$$\mathbf{Q} = \left(\sum_{i=1}^{l}{\mathbf{U}_i^{(p)}}^{\mathsf{T}}\mathbf{U}_i^{(p)}\right)\mathbf{Q} = \sum_{i=1}^{l}{\mathbf{U}_i^{(p)}}^{\mathsf{T}}\left(\mathbf{U}_i^{(p)}\mathbf{Q}\right) = \sum_{i=1}^{n-1}{\mathbf{U}_i^{(p)}}^{\mathsf{T}}\mathbf{U}_{i+1}^{(p)}, \tag{3.9}$$

where we used $\mathbf{U}_{i+1}^{(p)} = \mathbf{U}_i^{(p)}\mathbf{Q}$ and $\mathbf{U}_l^{(p)}\mathbf{Q} = \mathbf{0}$. ■

As a final remark, it should be stressed that the above construction *only* works if p is equal to the rank of $\boldsymbol{\Xi}$. In the next section, we treat the case in which $p < rank(\boldsymbol{\Xi})$.

Optimal solution for low dimensional autoencoders

When $p < rank(\boldsymbol{\Xi})$ the solution given above breaks down because $\tilde{\boldsymbol{\Xi}}_i = \mathbf{V}^{(p)}\mathbf{L}^{(p)}{\mathbf{U}_i^{(p)}}^{\mathsf{T}} \neq \boldsymbol{\Xi}_i$, and consequently $\tilde{\boldsymbol{\Xi}}_{i+1} \neq \mathbf{R}_n\tilde{\boldsymbol{\Xi}}_i$. So the question is whether the proposed solutions for $\mathbf{A}$ and $\mathbf{B}$ still hold the best reconstruction error when $p < rank(\boldsymbol{\Xi})$. In this thesis, we answer in negative terms to this question by resorting to a new formulation of our problem where we introduce *slack-like* matrices $\mathbf{E}_i^{(p)} \in \mathbb{R}^{n\times p},\ i = 1,\ldots,l+1$ collecting the reconstruction errors, which need to be minimized:

$$\begin{aligned} &\min_{\mathbf{Q}\in\mathbb{R}^{p\times p},\mathbf{E}_i^{(p)}} \quad \sum_{i=1}^{l+1}\|\mathbf{E}_i^{(p)}\|_F^2 \\ &\textit{subject to}: \quad \begin{bmatrix} \mathbf{U}_1^{(p)} + \mathbf{E}_1^{(p)} \\ \mathbf{U}_2^{(p)} + \mathbf{E}_2^{(p)} \\ \mathbf{U}_3^{(p)} + \mathbf{E}_3^{(p)} \\ \vdots \\ \mathbf{U}_l^{(p)} + \mathbf{E}_l^{(p)} \end{bmatrix}\mathbf{Q} = \begin{bmatrix} \mathbf{U}_2^{(p)} + \mathbf{E}_2^{(p)} \\ \mathbf{U}_3^{(p)} + \mathbf{E}_3^{(p)} \\ \vdots \\ \mathbf{U}_l^{(p)} + \mathbf{E}_l^{(p)} \\ \mathbf{E}_{l+1}^{(p)} \end{bmatrix} \end{aligned} \tag{3.10}$$

Notice that the problem above is convex both in the objective function and in the constraints; thus it only has global optimal solutions $\mathbf{E}_i^*$ and $\mathbf{Q}^*$, from which we can derive $\mathbf{A}^\mathsf{T} = \mathbf{U}_1^{(p)} + \mathbf{E}_1^*$ and $\mathbf{B}^\mathsf{T} = \mathbf{Q}^*$. Specifically, when $p = rank(\mathbf{\Xi})$, $\mathbf{R}_{k,n}^\mathsf{T}\mathbf{U}^{(p)}$ is in the span of $\mathbf{U}^{(p)}$ and the optimal solution is given by $\mathbf{E}_i^* = \mathbf{0}_{n\times p}\ \forall i$, and $\mathbf{Q}^* = {\mathbf{U}^{(p)}}^\mathsf{T}\mathbf{R}_{k,n}^\mathsf{T}\mathbf{U}^{(p)}$, i.e. the solution we have already described. If $p < rank(\mathbf{\Xi})$, the optimal solution cannot have $\forall i,\ \mathbf{E}_i^* = \mathbf{0}_{n\times p}$. However, it is not difficult to devise an iterative procedure to reach the minimum. Since in the experimental section we do not exploit the solution to this problem for reasons that we will explain later, here we just sketch such procedure. It helps to observe that, given a fixed $\mathbf{Q}$, the optimal solution for $\mathbf{E}_i^{(p)}$ is given by

$$[\tilde{\mathbf{E}}_1^{(p)}, \tilde{\mathbf{E}}_2^{(p)}, \ldots, \tilde{\mathbf{E}}_{l+1}^{(p)}] = [\mathbf{U}_1^{(p)}\mathbf{Q} - \mathbf{U}_2^{(p)}, \mathbf{U}_1^{(p)}\mathbf{Q}^2 - \mathbf{U}_3^{(p)}, \mathbf{U}_1^{(p)}\mathbf{Q}^3 - \mathbf{U}_4^{(p)}, \ldots]\mathbf{M}_\mathbf{Q}^+$$

where $\mathbf{M}_\mathbf{Q}^+$ is the pseudo inverse of $\mathbf{M}_\mathbf{Q} = \begin{bmatrix} -\mathbf{Q} & -\mathbf{Q}^2 & -\mathbf{Q}^3 & \cdots \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \cdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$.

In general, $\tilde{\mathbf{E}}^{(p)} = \left[\tilde{\mathbf{E}}_1^{(p)^\mathsf{T}}, \tilde{\mathbf{E}}_2^{(p)^\mathsf{T}}, \tilde{\mathbf{E}}_3^{(p)^\mathsf{T}}, \cdots, \tilde{\mathbf{E}}_n^{(p)^\mathsf{T}}\right]^\mathsf{T}$ can be decomposed into a component in the span of $\mathbf{U}^{(p)}$ and a component $\mathbf{E}^{(p)^\perp}$ orthogonal to it. Notice that $\mathbf{E}^{(p)^\perp}$ cannot be reduced, while (part of) the other component can be absorbed into $\mathbf{Q}$ by defining $\tilde{\mathbf{U}}^{(p)} = \mathbf{U}^{(p)} + \mathbf{E}^{(p)^\perp}$ and taking

$$\tilde{\mathbf{Q}} = (\tilde{\mathbf{U}}^{(p)})^+ \left[\tilde{\mathbf{U}}_2^{(p)^\mathsf{T}}, \tilde{\mathbf{U}}_3^{(p)^\mathsf{T}}, \cdots, \tilde{\mathbf{U}}_l^{(p)^\mathsf{T}}, \mathbf{E}_{l+1}^{(p)^\mathsf{T}}\right]^\mathsf{T}.$$

Given $\tilde{\mathbf{Q}}$, the new optimal values for $\mathbf{E}_i^{(p)}$ are obtained and the process iterated till convergence.

3.2.3 Method $\mathcal{L}_3$

This method uses the **Backpropagation Through Time (BPTT)** [93]. BPTT exploits the **Stochastic Gradient Descent (SGD)** [16] and propagates the gradient signal backward through all steps of sequences. As we have said at the beginning of this chapter, the most common methods used to "solve" Linear Dynamic Systems is related to the Identification System methods [59]. The two main problems in applying these techniques to machine learning tasks are: *i)* the methods do not allow to compute a solution for a dataset, indeed they are used to compute the parameters that allow to model the behavior of a physical system; *ii)* they do not allow to "generalize" during training. For this reason, we decided to use Backpropagation Through Time [93]. An advantage of applying this

method on a dynamical linear system is that the BPTT exploits the Stochastic Gradient descent, and the gradient of a linear function (as the one used by LDS) is a constant. Therefore, computing the SGD step is easier and faster than in nonlinear models. An important issue is that the linear units, used in LDS, do not have any "bounds" that limit the increasing (or the decreasing) of their values. Therefore, it is crucial to set the learning rate in a wise way. Indeed, the risk by using high learning rate values is that the model tends to diverge. Moreover, it is important to initialize weights matrices in an appropriate way. For this reason, we used the same technique used in method $\mathcal{L}_1$, even for matrix C.
The SGD is applied in order to minimize the error function $\tilde{E}_{\mathcal{T}}$ (eq. 3.2) with respect to the parameters of the models.

Chapter 4

LDS-based Models

In this chapter, various models that exploit the power of LDS (Section 2.1.1) are presented. The idea is to explore the potential of this model and to study the limit of LDS in learning sequential data. Moreover, the computational complexity and the advantage/disadvantage in terms of time in comparison with common nonlinear models (like RNN) is studied. The first proposed model is called Linear System Network. The idea is to arrange several linear systems in order to obtain a nonlinear representation of the input without using nonlinear units.

4.1 Linear System Network

The LDS is not enough expressive to model the complexity of complex sequential data [68]. Therefore the idea is to obtain a nonlinear model by arranging several LDSs. A Linear System Network (Figure 4.1) is composed of three layers. The first layer is composed of several LDSs that receive the same input and project it into the state space. The second layer of the LSN model selects and merges in a single vector the states of the LDSs in the first layer. In this way, the system creates a nonlinear representation of the input. For each LDS two limits are set based on the lengths of the sequences contained in the dataset. These two values are used by the second layer in order to select which LDS's states to insert in the second layer vector. The third layer is the output layer.
In the following section, we formally define the Linear System Network.

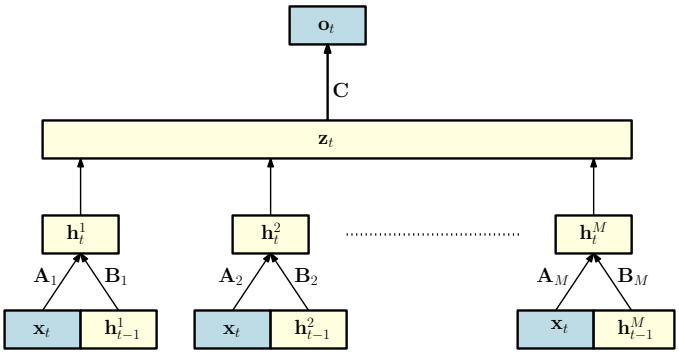

Figure 4.1: Schematic representation of the Linear System Network.

4.1.1 LSN Definition

The first layer of LSN is composed of M LDSs. Each LDS S_i is defined by the tuple $S_i = (\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ where $\mathbf{A}_i, \mathbf{B}_i$ and $\mathbf{C}_i$ are the input matrix, the state matrix, and the output matrix, respectively. The state of the i-th LDS at time step t given an input sequence $\mathbf{s} \equiv (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l)$ is computed as follows:

$$\mathbf{h}_t^i = \mathbf{A}_i \, \mathbf{x}_t + \mathbf{B}_i \, \mathbf{h}_{t-1}^i, \forall i \in \{1, \ldots, M\}, \tag{4.1}$$

$$\mathbf{o}_t^i = \mathbf{C}_i \, \mathbf{h}_t^i, \tag{4.2}$$

where $\mathbf{h}_t^i \in \mathbb{R}^{m_i}$ is the state of the i-th system at time t. In addition, we assume $\mathbf{h}_0^i = \mathbf{0} \in \mathbb{R}^{m_i}$ for each system i, i.e. the null vector. For each system S_i we can compute the error that it makes at each time step. In order to compute it, firstly, we have to calculate the matrix $\mathbf{C}_i$, that is computed by considering the entire dataset $\mathcal{T}$:

$$\mathbf{C}_i = \mathbf{D} \cdot \mathbf{H}_i^+, \tag{4.3}$$

where $\mathbf{D} = [\mathbf{d}_1^1, \mathbf{d}_2^1, \ldots, \mathbf{d}_{l_N}^N] \in \mathbb{R}^{s \times L}$, and $\mathbf{H}_i \in \mathbb{R}^{m_i \times L}$ (we recall that $L = \sum_{q=1}^{N} l_q$) that contains all the states computed by the i-th system for all inputs $\mathbf{x}_t$ of all sequences contained in $\mathcal{T}$. Given specific matrices for the LSD and a desired output sequence $\mathbf{d} \equiv (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_l)$ for a sequence $\mathbf{s}$, the error made by the i-th system at time step t is defined as

$$e_t^i = ||\mathbf{o}_t^i - \mathbf{d}_t||^2, \tag{4.4}$$

where $|| \cdot ||$ is the Euclidean norm. Each Linear Dynamical System in the first layer is associated to a tuple of values: (S_i, min_i, max_i), where min_i and max_i are two values used by the second layer to decide if the state $\mathbf{h}_t^i$ should be put forward in the computation or not. Specifically, the second

layer computes the $\mathbf{z}_t$ vector defined as:

$$\mathbf{z}_t = \begin{bmatrix} [\![min_1 < t \leq max_1]\!]\mathbf{h}_t^1, \\ [\![min_2 < t \leq max_2]\!]\mathbf{h}_t^2, \\ \vdots \\ [\![min_M < t \leq max_M]\!]\mathbf{h}_t^M \end{bmatrix} \in \mathbb{R}^{\sum_{i=1}^{M} m_i}, \tag{4.5}$$

where $[\![\cdot]\!]$ is equal to 1 if and only if the condition defined into the brackets is satisfied, otherwise it is 0. Finally, the output of the LSN model is given by:

$$\mathbf{out}_t = \mathbf{W}\mathbf{z}_t, \tag{4.6}$$

where $\mathbf{out}_t \in \mathbb{R}^s$ is the output of the LSN model, and $\mathbf{W} \in \mathbb{R}^{s \times \sum_{i=1}^{M} m_i}$ is the output matrix of the whole model. Values for min_i and max_i are defined on the basis of two limits which depend on the lengths of the sequences belonging to the training set $\mathcal{T}$:

$$\Delta s = \frac{\max(l_1, \ldots, l_N) - \min(l_1, \ldots, l_N)}{M}, \tag{4.7}$$

$$l_{min}^1 = \min(l_1, \ldots, l_N), \tag{4.8}$$

$$l_{max}^i = l_{min}^i + \Delta s, \quad \forall i \in \{1, \ldots, M-1\}, \tag{4.9}$$

$$l_{min}^i = l_{max}^{i-1}, \quad \forall i \in \{2, \ldots, M\}, \tag{4.10}$$

$$l_{max}^M = +\infty. \tag{4.11}$$

Training of the LSN model is inspired by the Reservoir Computing framework [61]. Indeed, it consists of two separated phases. The first phase consists in training the LDSs, while the second one computes $\mathbf{W}$. The LDSs composing the first layer of the LSN can be trained by using one of the techniques defined in Chapter 3. In particular, we tested the Linear Autoencoder method ($\mathcal{L}_2$). The selection of this technique is due to the fact that it ensures a faster training phase compared to SGD. Since our aim is to develop a fast and simple model capable to deal with sequential data, ensuring a fast training phase is crucial.
For what concerns the matrix $\mathbf{W}$ that connects the layer $\mathbf{z}$ to the output layer, it is computed as follows:

$$\mathbf{W} = \mathbf{D}\mathbf{Z}^+, \tag{4.12}$$

where $\mathbf{D} = [\mathbf{d}_1^1, \mathbf{d}_2^1, \ldots, \mathbf{d}_{l_N}^N]$ and $\mathbf{Z} = [\mathbf{z}_1^1, \mathbf{z}_2^1, \ldots, \mathbf{z}_{l_N}^N]$ (the upper index of each $\mathbf{z}$ refers to the input sequence index).
In the following we introduce the basic configuration for a LSN, while variants will be introduced in the subsequent section.

4.1.2 Basic Configuration

In the base version of the LSN, each system S_i is trained by using a subset of examples of $\mathcal{T}$ that contains all sequences with a length $\geq l^i_{min}$. Therefore, for each system S_i an ad-hoc training set $\mathcal{T}^i$ is defined. Each set $\mathcal{T}^i$ is defined as:

$$\mathcal{T}^i = \{(\mathbf{s}^q, \mathbf{d}^q) | (\mathbf{s}^q, \mathbf{d}^q) \in \mathcal{T}, \forall q \in \{1, \ldots, N\},\ s.t.\ l_q \geq l^i_{min}\}. \tag{4.13}$$

In this version, the parameters min_i and max_i are defined as follows:

$$min_i = 0, \quad \forall i, \tag{4.14}$$

$$max_i = l^i_{max}, \quad \forall i. \tag{4.15}$$

In this way, the contribution of the state of system S_i to $\mathbf{z}_t$ is $\neq 0$ if and only if $t \leq max_i$. Another parameter of the model is $p \in \mathbb{Z}^+$ that defines the number of dimensions of the state space of the LDSs. In the simplest version of the model all LDSs have the same state dimensions:

$$m_i = p \quad \forall i. \tag{4.16}$$

4.1.3 Configuration variants

We developed different versions of the LDS model, in order to explore the behaviors of the model under different variants. In the following these variants, that we dubbed **Var**, **Over**, **Select**, and **Error**, are presented and discussed. Please note that each variant reported below may be used or not regardless of whether the others are used or not. Indeed, each variant modifies a specific feature of the basic version of the model.

Var: The size of the state space of each system is tuned based on the rank of data matrix $\Xi_{\mathcal{T}^i}$. To make the notation easier to understand, as a subscript of Ξ we use $\mathcal{T}^i$, that means the data matrix Ξ is constructed by considering all inputs of the dataset $\mathcal{T}^i$.
The size of state space m_i of the i-th system is set as:

$$m_i = p\sqrt{\frac{rank(\Xi_{\mathcal{T}^i})}{rank(\Xi_{\mathcal{T}^1})}}. \tag{4.17}$$

Over: The training dataset associated with the system S_i contains all sequences that have length $\leq l^i_{max}$:

$$\mathcal{T}^i = \{(\mathbf{s^q}, \mathbf{d^q}) | (\mathbf{s^q}, \mathbf{d^q}) \in \mathcal{T}, \forall q \in \{1, \ldots, N\},\ s.t.\ l_q \leq l^i_{max}\}. \tag{4.18}$$

Select: The parameters min_i and max_i for each system S_i are set as follows:

$$min_i = l^i_{min} \quad \forall i, \tag{4.19}$$

$$max_i = l^i_{max} \quad \forall i. \tag{4.20}$$

In this way, the state computed by system S_i is inserted in $\mathbf{z}_t$ if and only if $l^i_{min} \leq t \leq l^i_{max}$. Therefore each system is used to represent the state of a specific part of the input sequence.

Error: The idea is to multiply the state of each system by a weight based on the normalized error made by it. At the beginning of this chapter, we defined e^i_t as the error made by the i-th LDS at time step t, given in input a sequence $\mathbf{s} \equiv (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l)$. Let's now define $e^i(t)$ as the error made by the system S_i till time step t, given a sequence $\mathbf{s}$ as input. The idea is to compute a weight $w^i(t)$ that gives more relevance to LDSs that make a lower overall error:

$$e^i(t) = \sum_{k=1}^{t} e^i_k. \tag{4.21}$$

Actually, we use the normalized error $n^i(t)$, that is normalized over the errors made by all LDSs till the current time step t:

$$n^i(t) = \frac{e^i(t)}{\sum_{j=1}^{M} e^j(t)}. \tag{4.22}$$

At each time step the state of each LDS is multiplied by the weight w^i_t that is defined in such a way that the higher is the global error made by a system, the lower will be the weight assigned to it:

$$w^i(t) = 1 - n^i(t). \tag{4.23}$$

Therefore, if the **Error** variant is turned on the $\mathbf{z}_t$ vector of the LSN model is computed as follows:

$$\mathbf{z}_t = \begin{bmatrix} [\![min_1 < t \leq max_1]\!] w^1(t) \mathbf{h}^1_t \\ [\![min_2 < t \leq max_2]\!] w^2(t) \mathbf{h}^2_t \\ \vdots \\ [\![min_M < t \leq max_M]\!] w^M(t) \mathbf{h}^M_t \end{bmatrix}. \tag{4.24}$$

An interesting advantage of defining w_t^i in this way is that it may be computed based on the result obtained during the previous time step. Indeed,

$$n^i(t) = \frac{e^i(t)}{\sum_{j=1}^{M} e^j(t)} = \frac{\sum_{k=1}^{t} e_k^i}{\sum_{j=1}^{M}(\sum_{k=1}^{t} e_k^j)} = \frac{\sum_{k=1}^{t-1} e_k^i + e_t^i}{\sum_{j=1}^{M}(\sum_{k=1}^{t-1} e_k^j) + \sum_{j=1}^{M} e_t^j}. \tag{4.25}$$

Let's define $N_{t-1} = \sum_{j=1}^{M}(\sum_{k=1}^{t-1} e_k^j)$, hence $n^i(t)$ can be compute based only on the error computed till time step $t-1$ and the "local" error made by the LDSs:

$$n^i(t) = \frac{e^i(t-1) + e_t^i}{N_{t-1} + \sum_{j=1}^{M} e_t^j}. \tag{4.26}$$

This allows to compute the weight w_t^i in a convenient way. In fact, at each time step t we can compute the overall error made by system S_i just by using information that come from the previous time step. Notice that

$$N_t = N_{t-1} + \sum_{j=1}^{M} e_t^j, \tag{4.27}$$

$$e^i(t) = e^i(t-1) + e_t^i. \tag{4.28}$$

Therefore all information that are necessary to compute the $n^i(t)$ are already computed during the previous time step. This solution allows to optimize the computation in terms of time and memory demands.

4.2 Sequential LSN

An evolution of LSN is the Sequential LSN (SLSN) (Figure 4.2). The idea is to connect several LDSs and specialize each one in computing the output for a small interval of time steps. We say that the LDSs are "connected" because, as shown in Figure 4.2, the first state vector $\mathbf{h}_0^i$ of each system S_i is defined as the last state $\mathbf{h}_{max_i}^{(i-1)}$ computed by the previous system S_{i-1}. Another important difference is that each system S_i has its own output $\mathbf{o}_t$. Indeed, in LSN the model uses the vector $\mathbf{z}_t$ in order to "collect" the state of all LDSs at time t, but in the case of SLSN each system S_i computes its state $\mathbf{h}_t$ (eq. 4.1), and uses it in order to compute its own output $\mathbf{o}_t$ (eq. 4.2).
The values min_i, max_i associated which each system S_i are used to identify the time interval of each sequence that system S_i manages. The size of this

interval is the same for each system. Therefore the higher is the number of systems, the smaller will be the interval where a specific system is used to contribute to the state of the overall system.
In order to compute the output, each system S_i uses a different output matrix. These matrices are trained by considering the projection computed via eq. (4.1) by using as input a specific interval of time steps instead of using the entire sequence.
In order to compute the state, each system needs to "receive" the value of $\mathbf{h}_0^i$ from the previous system S_{i-1}. The main issue that arises when the system S_{i-1} has to pass the state to the next LDS is that it's possible that the sizes of the states of the two systems are not equal. We have developed two different techniques to solve this problem. The first one consists in cutting, or padding with 0, the previous state in order to create a new hidden state that has the same sizes of the hidden state of the current system. The second method exploits the random projection technique using a random matrix $\mathbf{R}$ to map the previous state in the vectorial space where the hidden state of the current LDS belongs to. Formally:

- using random projection:

  $$\mathbf{h}^i_{min_{i-1}} = \mathbf{h}^{i-1}_{max_{i-1}}\mathbf{R},$$

  where $\mathbf{R}$ is a random matrix normalized with respect to norm 2. As we prove for LDS, this property is sufficient to ensure that all the eigenvalues of $\mathbf{R}$ are ≤ 1; thus, it ensures that the hidden state will not be asymptotically influenced by any initial condition caused by the values in $\mathbf{R}$.

- By cutting/padding with zeros $\mathbf{h}^{i-1}_{max_{i-1}}$:

  if $m_i < m_{i-1}$, $\mathbf{h}^i_{min_{i-1}} = \mathbf{h}^{i-1}_{max_{i-1}[1...m_i]}$ where $\mathbf{h}^{i-1}_{max_{i-1}[1...m_i]}$ is the vector composed of the first m_i elements of $\mathbf{h}^{i-1}_{max_{i-1}}$ (Cutting);

  if $m_i > m_{i-1}$, $\mathbf{h}^i_{min_{i-1}} = [\mathbf{h}^{i-1}_{max_{i-1}}, \mathbf{0}]$, where $\mathbf{0} \in \mathbb{R}^{m_i - m_{i-1}}$ (Padding).

Since each system S_i works only on a portion of sequence in input, the output of the model is defined as follows:

$$\mathbf{o}_t = \sum_{i=1}^{M} \mathbf{o}_t^i [\![min_i < t \leq max_i]\!], \tag{4.29}$$

where $\mathbf{o}_t^i$ is the output of an LDS computed as in eq. (4.2).

Until now we have discussed about the definition of the model, another important aspect is how the model is trained. Similar to LSN, each LDS

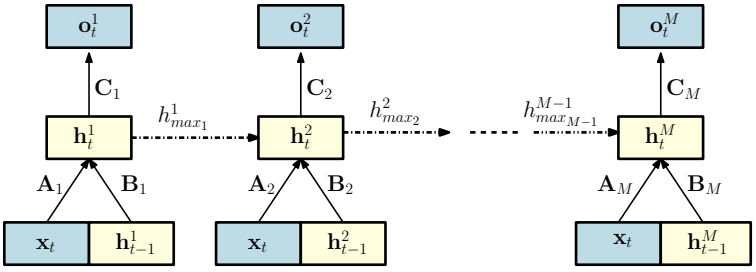

Figure 4.2: Schematic representation of the Sequential LSN.

is trained independently. We tested two methods to train the LDS, that compose this model. The first is the random method $\mathcal{L}_1$, the second is the the one based on Linear Autoencoder ($\mathcal{L}_2$). For what concerns the application of the second method, our aim is to train the model in order to specialize it on dealing with a specific part of the sequences. For this reason, the training set $\mathcal{T}^i$ used to train the S_i system is defined as follows:

$$\mathcal{T}^i = \{(\mathbf{s}^q_{S_i}, \mathbf{d}^q_{S_i}) | (\mathbf{s}^q, \mathbf{d}^q) \in \mathcal{T}, \mathbf{s}^q_{S_i} = \mathbf{s}^q_{[min_i \dots max_i]}, \\ \mathbf{d}^q_{S_i} = \mathbf{d}^q_{[min_i \dots max_i]}, q = 1, \dots, N\}. \tag{4.30}$$

The limits min_i and max_i are define as follows:

$$min_i = l^i_{min} \forall i, \tag{4.31}$$

$$max_i = l^i_{max} \forall i. \tag{4.32}$$

4.3 Co-learning with LDS[1]

In this section, we are going to introduce several models that exploit co-learning techniques in order to improve the results obtained with the LSN and SLSN. All the models that are presented in this section maintain a structure that is similar to the LDS model, but they try to use the co-learning techniques in order to bypass some of the limits of this simple model. Therefore the idea is to use along with the linear model a more complex model that is trained on the hidden states of the LDS. This idea is inspired by model for classification tasks proposed by Chen et al. in [21].
The main issue of LDS is that the **B** matrix tends to have a *spectral radius* lower or equal to 1, and this makes the system *contractive*, that means that the system tends to achieve a fixpoint solution in a few steps. Therefore dealing with long term temporal dependencies becomes unfeasible. Having

[1] In collaboration with Professor Peter Tino (department of Computer Science, University of Birmingham).

the spectral radius lower or equal to 1 is necessary when we randomly initialize the weights, otherwise, the output of the system tends to become higher and higher at every time step. This behavior makes it likely that the system diverges. Therefore our aim is to develop a model that, by setting the random output weights ($\mathbf{C}$), tries to learn a function that computes the best input and recursive weights ($\mathbf{A}$ and $\mathbf{B}$, respectively), for each input. Hence the first step is to randomly initialize the matrix $\mathbf{C} \in \mathbb{R}^{s \times m}$. Then, by maintaining the $\mathbf{C}$ matrix fixed, we want to compute the $\mathbf{h}_t$ values by minimizing the *MSE* error, for each target $\mathbf{d}_i^q$ in the training set $\mathcal{T}$. Once $\mathbf{C}$ is defined as a random matrix, the hidden states can be computed as:

$$\mathbf{h}_i^q = \mathbf{C}^+ \mathbf{d}_i^q, \quad \forall i \in \{1, \dots, l_q\} \wedge \forall q \in \{1, \dots, N\}. \tag{4.33}$$

Let's consider a sequence $\mathbf{s}^q$ and then define the vector $\mathbf{i}_t^q \in \mathbb{R}^{n+m}$ that contains the input at time step t, and the state $\mathbf{h}_{t-1}^q$ of the previous step:

$$\mathbf{i}_t^q = \begin{bmatrix} \mathbf{x}_t^q \\ \mathbf{h}_{t-1}^q \end{bmatrix}, \tag{4.34}$$

given $\mathbf{i}_t^q$ for each time step t, it is possible to compute the matrix $\mathbf{V}_t^q$ s.t.

$$\mathbf{h}_t^q = \mathbf{V}_t^b \mathbf{i}_t^q, \tag{4.35}$$

since we have already computed the values $\mathbf{h}_t^q$ and $\mathbf{i}_t^q$ we can compute $\mathbf{V}_t^q$ as:

$$\mathbf{V}_t^b = \mathbf{h}_t^q (\mathbf{i}_t^q)^+. \tag{4.36}$$

The goal is to collect $\mathbf{V}_t^b$ for each input $\mathbf{x}_t^q$ in the training set, and create the set $\mathcal{S} = \{(\mathbf{i}_t^q, \mathbf{V}_t^b) | \forall q \in \{1, \dots, M\}, t \in \{1, \dots, l_q\}\}$. This set will be used as a training dataset for an external model $\mathcal{F}$, that is trained in order to learn the function that maps $\mathbf{i}_t^q$ in $\mathbf{V}_t^b$. The result is a model that, given an input $\mathbf{x}_t$ computes the output $\mathbf{o}_t$ as follows:

$$\mathbf{h}_t = \mathcal{F}(\mathbf{i}_t)\mathbf{i}_t, \tag{4.37}$$

$$\mathbf{o}_t = \mathbf{C}\mathbf{h}_t. \tag{4.38}$$

For what concerns the model $\mathcal{F}$, it is possible to use many different models (e.g. Deep Neural Networks, Support Vector Machines [29], etc.).

A simplified version of this model uses the model $\mathcal{F}$ to learn the function that maps $\mathbf{i}_t$ to $\mathbf{h}_t$. In this case, instead of computing (and collecting) $\mathbf{V}_t^b$, we just collect $\mathbf{h}_t^q$ (computed by eq. (4.33)). The dataset used to train the model $\mathcal{F}$ is the following:

$$\mathcal{S} = \{(\mathbf{i}_t^q, \mathbf{h}_t^b) | \forall q \in \{1, \dots, M\}, t \in \{1, \dots, l_q\}\}. \tag{4.39}$$

Finally, the hidden state of the model during the execution of a task is computed as:

$$\mathbf{h}_t = \mathcal{F}(\mathbf{i}_t). \tag{4.40}$$

The output o_t is still computed by eq. (4.38).

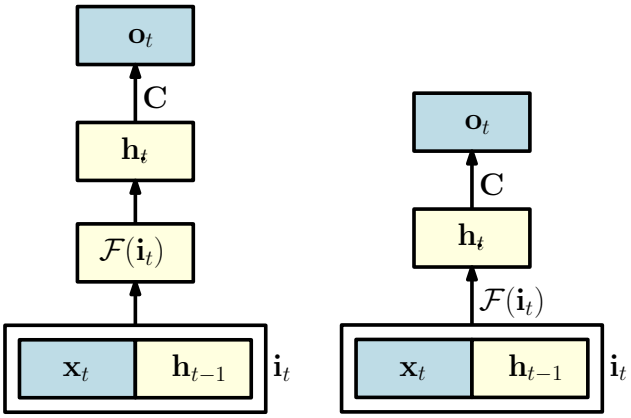

Figure 4.3: Schematic representation of the two Co-learning models.

4.3.1 Discussion on Linear Co-learning models

The two models presented above exploit co-learning techniques; moreover, both are trained by firstly initializing the matrix $\mathbf{C}$ with a random matrix that is used to compute the optimal state representations given the target $\mathbf{d}_t^q$. These hidden representations are used to train an external (no-recurrent) model in order to learn the function that maps the input (and the state computed during the previous time step) to the current computed state. Moreover, the models use the state space in order to store information that comes from previous time steps, and the model uses them to compute the output.
In particular, we consider the prediction task where $\mathbf{d}_t^q = \mathbf{x}_{t+1}^q$. Hence, given the current input $\mathbf{x}_t^q$ and the previous state $\mathbf{h}_{t-1}^q$, the system tries to compute the next time step $\mathbf{x}_{t+1}^q$ of the input sequence. Both items of the sequence, $\mathbf{x}_t^q$ and $\mathbf{x}_{t+1}^q$ belong to $\mathbb{R}^n$. Let's define $\mathbf{D}$ the matrix that contains all targets: $\mathbf{D} = [\mathbf{d}_1^1, \ldots, \mathbf{d}_{l_N}^N]$, and $r = rank(\mathbf{D})$ i.e. the number of basis vectors needed to generate the column space of $\mathbf{D}$. Recall that $\mathbf{h}_t^q \in \mathbb{R}^m$, $\mathbf{d}_t^q \in \mathbb{R}^n$. In the models discussed above we compute the state representation $\mathbf{h}_t^q$ as:

$$\mathbf{h}_t^q = \mathbf{C}^+ \mathbf{d}_t^q, \tag{4.41}$$

where matrix $\mathbf{C} \in \mathbb{R}^{m \times r}$. Notice that we consider $\mathbf{C}$ as a full rank matrix since it is a random matrix[2]. Now we prove that in case we fix $\mathbf{C}$ and use it to compute $\mathbf{h}_t^q$, the obtained LDS is stateless.

[2]In practice $\mathbf{C}$ is computed by using a pseudo-random generator of some programming language, that can not ensure that the obtained matrix is really random. Anyway, it's very likely that it is a full rank matrix. Therefore after creating it, it is important to check if $\mathbf{C}$ is really full-rank, otherwise, we can just re-compute $\mathbf{C}$ until we obtain a full rank matrix.

Theorem 2. *Given an LDS with a fixed full rank matrix* $\mathbf{C}$*, where* $\mathbf{C}$ *is used to compute* $\mathbf{h}_t^q$*, and where* $\mathbf{d}_t^q = \mathbf{x}_{t+1}^q$*, then the considered LDS is stateless. A stateless LDS does not exploit the* $\mathbf{h}_{t-1}^q$ *to compute* $\mathbf{o}_t^q$*, therefore* $\mathbf{o}_t^q = \mathcal{F}(\mathbf{x}_t^q, \mathbf{C})$*.*

Proof. Let's start by considering three possible cases:

1: $r = m$, hence $\mathbf{C}$ is a squared, invertible matrix;

2: $r < m$;

3: $r > m$.

In case of $r = m$, $\mathbf{C}^{-1}$ exists. From equations (4.1) and (4.2) it is possible to derive that:

$$\begin{aligned} \mathbf{o}_t^q &= \mathbf{C}(\mathbf{A}\mathbf{x}_t^q + \mathbf{B}\mathbf{h}_{t-1}^q) = \\ &= \mathbf{C}\mathbf{A}\mathbf{x}_t^q + \mathbf{C}\mathbf{B}\mathbf{h}_{t-1}^q. \end{aligned} \tag{4.42}$$

We consider a prediction task. If $\mathbf{A}$ and $\mathbf{B}$ exist such that $\mathbf{o}_t^q = \mathbf{x}_{t+1}^q$, we can state $\mathbf{h}_{t-1}^q = \mathbf{C}^{-1}\mathbf{x}_t^q$; then $\mathbf{x}_{t+1}^q$ can be defined as:

$$\mathbf{x}_{t+1}^q = \mathbf{C}\mathbf{A}\mathbf{x}_t^q + \mathbf{C}\mathbf{B}(\mathbf{C}^{-1}\mathbf{x}_t^q). \tag{4.43}$$

Notice that, the state $\mathbf{h}_t^q$ does not appear in equation (4.43) therefore, we can conclude that the LDS does not need the state $\mathbf{h}_t^q$ in order to compute $\mathbf{x}_{t+1}^q$.
In addition, if we consider a model similar to the one presented in Section 4.3, in general, we have that the state of the system is computed by a function $\mathcal{F}$ that receives in input $\mathbf{x}_t^q$ and the state computed during the previous step, so $\mathbf{h}_t^q$ is:

$$\mathbf{h}_t^q = \mathcal{F}(\mathbf{x}_t^q, \mathbf{h}_{t-1}^q). \tag{4.44}$$

Even in this case, if $\mathbf{C}^{-1}$ exists, we can compute the output of the model as:

$$\mathbf{x}_{t+1}^q = \mathbf{C}\mathcal{F}(\mathbf{x}_t^q, \mathbf{C}^{-1}\mathbf{x}_t^q), \tag{4.45}$$

therefore the model does not use the state values $\mathbf{h}_{t-1}^q$ to compute the output.

In case of $r < m$:
Let's define $\mathcal{H}$ as the set that contains all vectors in $\mathbb{R}^m$ spanned by the base $\mathcal{B}^{\mathcal{H}} = \{\mathbf{b}_1^H, \mathbf{b}_2^H, \ldots, \mathbf{b}_m^H\}$ of the matrix $\mathbf{H}$, and analogously let's define $\mathcal{D}$ as the set that contains all vectors in $\mathbb{R}^n$ spanned by the base $\mathcal{B}^D = \{\mathbf{b}_1^D, \mathbf{b}_2^D, \ldots, \mathbf{b}_r^D\}$ of the matrix $\mathbf{D}$.
Let's define $\mathcal{H}_1$ and $\mathcal{H}_2$ as follows:

$$\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2, \mathcal{H}_1 \perp \mathcal{H}_2, |\mathcal{B}^{\mathcal{H}_1}| = r, \mathcal{H}_2 = Ker(\mathbf{C}), \tag{4.46}$$

where $Ker(\mathbf{C})$ is the kernel of the matrix $\mathbf{C}$, $\mathcal{B}^{\mathcal{H}_1} = \{\mathbf{b}_1^H, \mathbf{b}_2^H, \ldots, \mathbf{b}_r^H\}$ is the set that contains the vectors that compose a base of $\mathcal{H}_1$ and $\mathcal{B}^{\mathcal{H}_2} = \{\mathbf{b}_{r+1}^H, \ldots, \mathbf{b}_m^H\}$ the set of vectors that compose a base for $\mathcal{H}_2$. Given a non-null vector $\mathbf{h} \in \mathcal{H}_1$, we can define a vector $\mathbf{d} \in \mathcal{D}$ as:

$$\mathbf{d} = \mathbf{C}\mathbf{h} \neq \mathbf{0}, \tag{4.47}$$

where $\mathbf{0} \in \mathbb{R}^r$ is the vector with all values set to 0. But we can obtain the same vector $\mathbf{d}$ as follows:

$$\mathbf{d} = \mathbf{C}(\mathbf{h} + \mathbf{v}) = \mathbf{C}\mathbf{h} + \mathbf{C}\mathbf{v} \quad \forall \mathbf{v} \in \mathcal{H}_2. \tag{4.48}$$

This equation holds $\forall \mathbf{h} \in \mathcal{H}_1, \mathbf{h} \neq \mathbf{0}$. Therefore, we can conclude that there exist at least two elements belonging to $\mathcal{H}$ that have the same projection in $\mathcal{D}$. $\mathbf{h}$ and $\mathbf{v}$ belong to $\mathcal{H}$, so they can be written as a linear combination of some vectors contained in $\mathcal{B}^{\mathcal{H}}$; therefore, even $\mathbf{h} + \mathbf{v}$ can be written as a linear combination of the vectors contained in $\mathcal{B}^{\mathcal{H}}$. Moreover, $\|\mathbf{h} + \mathbf{v}\| \geq \|\mathbf{h}\|$.
All elements $\mathbf{h}_t^q \in \mathcal{H}$ are computed using equation (4.41), which exploits the pseudo-inverse. The pseudo-inverse selects the vector that minimizes the *MSE* and that minimizes its norm. Hence, we can consider only the elements that respect these two conditions, i.e. the vectors in $\mathcal{H}_1$. Therefore, this case can be reduced to the case where $r = m$. For these reasons, also in this case the linear system is stateless.

In case of $r > m$:
similarly to the previous case we have some vectors in $\mathcal{D}$ that have the same projection in $\mathcal{H}$. Hence the intuition is that at least two targets will be represented in the same way in the state space, and this makes them indistinguishable. Therefore, we can consider only the targets that can be uniquely represented in $\mathcal{H}$. This brings back this case to the case where $r = m$.
More formally, we can divide $\mathcal{D}$ in two subsets $\mathcal{D}_1$ and $\mathcal{D}_2$ where:

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{D}_1 \perp \mathcal{D}_2, |\mathcal{B}^{D_1}| = m, \mathcal{D}_2 = Ker(\mathbf{C}^+). \tag{4.49}$$

Where $\mathcal{B}^{\mathcal{D}_1} = \{\mathbf{b}_1^D, \mathbf{b}_2^D, \ldots, \mathbf{b}_m^D\}$ is the set of vectors that define a base for $\mathbf{D}_1$, and $\mathcal{B}^{\mathcal{D}_2} = \{\mathbf{b}_{m+1}^D, \ldots, \mathbf{b}_r^D\}$ is the set of vectors that define a base for $\mathbf{D}_2$. Given a vector $\mathbf{d} \in \mathcal{D}$, its projection vector $\mathbf{h} \in \mathcal{H}$ is computed as follows:

$$\mathbf{h} = \mathbf{C}^+\mathbf{d}. \tag{4.50}$$

Moreover $\mathbf{h}$ can be also computed as:

$$\mathbf{h} = \mathbf{C}^+(\mathbf{d} + \mathbf{v}) = \mathbf{C}^+\mathbf{d} + \mathbf{C}^+\mathbf{v} \quad \forall \mathbf{d} \in \mathcal{D}_2 \tag{4.51}$$

As a consequence, there are two vectors in $\mathcal{D}$ that have the same projection in $\mathcal{H}$. Since $\mathbf{d}, \mathbf{v} \in \mathcal{D}$ then they can be written as a liner combination of some vectors contained in $\mathcal{B}^{\mathcal{D}}$, and therefore even $\mathbf{b} + \mathbf{v}$ belongs to $\mathcal{D}$. Additionally, the number of distinct vectors that can be represented in $\mathcal{H}$, that are projections of a vector belonging to $\mathcal{D}$, is equal to m. Hence, even in this case, we have the same situation studied in the case where $m = r$. □

4.3.2 Uni-Network

One of the most interesting results obtained by testing the LDS is the capability of this system to learn one single sequence. Indeed, by training a LDS by a single sequence and then using it to perform prediction on the same sequence, the system has achieved an accuracy higher than 90%, as reported in Section 6.3.3, Table 6.5. These results are all obtained by using the random training method ($\mathcal{L}_1$), presented in Section 3. In order to exploit this feature of the LDS model, we have developed a brand new co-learning method. The method consists in creating one LDS for each sequence $\mathbf{s}^q$ contained in the training set $\mathcal{T}$. Each system shares the same matrix $\mathbf{A}$ and $\mathbf{B}$, that are randomly computed. Differently, from previous cases each LDS S_i has its own $\mathbf{C}_i$ computed as follows:

$$\mathbf{C}_i = \mathbf{H}^{i^+}\mathbf{D}^i, \tag{4.52}$$

where $\mathbf{H}^i = [\mathbf{A}\mathbf{x}^i_1, \mathbf{A}\mathbf{x}^i_2 + \mathbf{B}\mathbf{h}^i_1, \ldots, \mathbf{A}\mathbf{x}_{l_i} + \mathbf{B}\mathbf{h}_{l_{i-1}}]$ and $\mathbf{D}^i = [\mathbf{d}^i_1, \mathbf{d}^i_2, \ldots, \mathbf{d}^i_{l_q}]$, $\forall i \in \{1, \ldots, N\}$. Then all $\mathbf{C}_i$ and $\mathbf{s}^i$ are collected together in a set $\mathcal{T}_{\mathcal{M}} = \{(\mathbf{s}^1, \mathbf{C}_1), (\mathbf{s}^2, \mathbf{C}_2), \ldots, (\mathbf{s}^N, \mathbf{C}_N)\}$ that is used as a training set to train a nonlinear model $\mathcal{M}$. The idea is to use $\mathcal{M}$ in order to compute the best matrix $\mathbf{C}$ for the sequence $\mathbf{s}^i$ given in input to the system. The prediction, given $\mathbf{x}^i_j$ as input, is computed as follows:

$$\mathbf{h}^i_j = \mathbf{A}\mathbf{x}^i_j + \mathbf{B}\mathbf{h}^i_{j-1}, \tag{4.53}$$

$$\mathbf{o}^i_j = \mathcal{M}(\mathbf{x}^i_j, \mathbf{h}^i_{j-1})\mathbf{h}^i_j. \tag{4.54}$$

This method has the advantage that just the output weights have to be considered when training $\mathcal{M}$, but the main disadvantage is the huge size of the data contained in $\mathcal{T}_M$ that makes it difficult to train $\mathcal{M}$.

4.4 Encode-Decode LDS

This model (Figure 4.4) is based on the Linear Autoencoder and in particular on the training method presented in Section 3.2.2. The idea is to use two different Autoencoders: one to encode the input and one to encode the target. The idea to use an Autoencoder for the targets is to get new "hidden targets" that encode contextual information, thus disambiguating

the same target value occurring in different positions of the sequence. Since the training method presented in Section 3.2.2 is unsupervised, it is possible to train the two Autoencoders independently. The idea is to compute the encoding of the input and map it to the corresponding encoding of its target. We refer to the LDS that encodes the input as "Encoder", and to the LDS that encodes the output as "Decoder".

Encoder and Decoder are trained respectively with the following datasets:

$$\mathcal{T}_{Encoder} = \{\mathbf{x}_t^q | \mathbf{x}_t^q \in \mathbf{s}_q \, \forall t \in \{1, \ldots, l_q\}, \, \forall (\mathbf{s}_q, \mathbf{d}_q) \in \mathcal{T}\}, \tag{4.55}$$

$$\mathcal{T}_{Decoder} = \{\mathbf{d}_t^q | \mathbf{d}_t^q \in \mathbf{d}_q \, \forall t \in \{1, \ldots, l_q\}, \, \forall (\mathbf{s}_q, \mathbf{d}_q) \in \mathcal{T}\}. \tag{4.56}$$

$\mathcal{T}_{Encoder}$ and $\mathcal{T}_{Decoder}$ allow to train the two LDSs in order to respectively learn how to encode the input $\mathbf{x}_t$ and encode the output (target) $\mathbf{d}_t$. Both systems compute the encoding function by using eq. (4.1), but in order to clarify the various elements that compose the models we rewrite the equation that computes the projection of the input (calculated by the Encoder) as:

$$\mathbf{h}_t^{(Encoder)} = \mathbf{A}^{(Encoder)} \, \mathbf{x}_t + \mathbf{B}^{(Encoder)} \, \mathbf{h}_{t-1}^{(Encoder)}, \tag{4.57}$$

and the equation that computes the projection of the target (calculated by the Decoder) as:

$$\mathbf{h}_t^{(Decoder)} = \mathbf{A}^{(Decoder)} \, \mathbf{o}_t + \mathbf{B}^{(Decoder)} \, \mathbf{h}_{t-1}^{(Decoder)}. \tag{4.58}$$

The idea is to connect the two projections $\mathbf{h}_t^{(Encoder)}$ and $\mathbf{h}_t^{(Decoder)}$, by using a matrix $\mathbf{E}$, that has to be trained in order to have:

$$\mathbf{h}_t^{(Decoder)} = \mathbf{E}\mathbf{h}_t^{(Encoder)}. \tag{4.59}$$

Then, exploiting the matrix $\mathbf{A}^{(Decoder)}$ the model has to map $\mathbf{h}_t^{(Encoder)}$ to the output:

$$\mathbf{o}_t = \mathbf{A}^{(Decoder)^\mathsf{T}} \, \mathbf{h}_t^{(Decoder)}. \tag{4.60}$$

It's important to notice that $\mathbf{B}^{(Decoder)}$ is not used to compute $\mathbf{o}_t$.
Given an input $\mathbf{x}_t$ the model computes $\mathbf{h}_t^{(Encoder)}$ by applying eq. (4.57). Then, the model projects $\mathbf{h}_t^{(Encoder)}$ in to Decoder's state space by eq. (4.59). Finally, the output $\mathbf{o}_t$ is computed by using (eq. 4.60). For what concerns the training phase, we have already explained how to compute matrices $\mathbf{A}^{(Encoder)}$, $\mathbf{B}^{(Encoder)}$ and $\mathbf{A}^{(Decoder)}$, while matrix $\mathbf{E}$ is computed via SGD or pseudo-inverse. In order to do that, firstly we have to compute $\mathbf{h}_t^{(Encoder)}$ and $\mathbf{h}_t^{(Decoder)}$ for each input $\mathbf{x}_t$ and for each target $\mathbf{d}_t$ by using eqs. (4.57) and (4.58). Then, the computed values are collected in a dataset $\mathcal{T}_E$:

$$\mathcal{T}_E = \{(\mathbf{h}_t^{(Encoder)}, \mathbf{h}_t^{(Decoder)})\}, \tag{4.61}$$

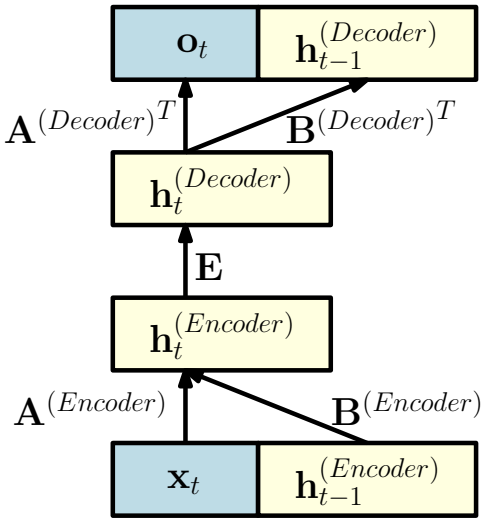

Figure 4.4: Schematic representation of the Encode-Decode LDS.

and used to train matrix $\mathbf{E}$. In the case that matrix $\mathbf{E}$ is computed by using the pseudoinverse, the values $\mathbf{h}_t^{(Encoder)}$ and $\mathbf{h}_t^{(Decoder)}$ are arranged in two matrices $\mathbf{H}^{(Encoder)}$ and $\mathbf{H}^{(Decoder)}$, and matrix $\mathbf{E}$ is computed as:

$$\mathbf{E} = \mathbf{H}^{(Decoder)} \, \mathbf{H}^{(Encoder)^+}. \tag{4.62}$$

Chapter 5

Pre-Training Via Linear Models

The task of learning in sequential domains has been widely studied in the last few years. In particular, many deep learning models have been proposed to perform classification and prediction of sequential data. Typically Deep Learning models for vectorial data use a pre-training phase in order to obtain better results in less time. Unfortunately, the use of a pre-training phase is not so common in a sequential domain. In this chapter, we briefly introduce the concept of pre-training, and then we show two powerful methods to pre-training sequential Deep Models. The most interesting aspect is that these two methods are based on linear models. In both cases the idea is to use a simple model to extract knowledge from the training set in a short time, and to exploit such knowledge to initialize the weights of a more complex nonlinear model, such as RNN.

5.1 Pre-Training

The pre-training phase was firstly introduced in 2006 [45], and has been the real breakthrough in Deep Learning. Indeed, the introduction of Pre-Training made possible to train deep models in an effective and efficient way. It has been introduced as an algorithm to pre-train deep belief networks (DBN) [44] and stacked Autoencoders [89, 14]. The idea that these pre-training methods exploit is to perform an unsupervised training phase on each layer of DNN separately (layer-wise). After the pre-training phase the model has to be fine tuned, via supervised training by gradient-based optimization. The aim of the pre-training phase is to drive the parameters of the model in a region from where reaching a better local (or global) optimal solution is somehow easier [14]. in [33] it is argued that the pre-training phase acts as an unusual form of regularization. Indeed, the pre-training

renders the parameters space more "complex" to be traveled away from the basin where a local (or global) optimum is located. For instance, in models that exploit sigmoid nonlinearity, after pre-training, the cost function becomes more "complex" (the function will present more topological features e.g. plateau, peak, etc.) with the increasing of the distance from the basin. Therefore the pre-training restricts the area where the solution will be searched. This behavior corresponds to "learn" the structure of the probability distribution of the input. This idea is the one we followed when developing the *HMM-based pre-training* method published in [70, 69]. Indeed, this method uses a simple dynamical model (e.g HMM) in order to learn an approximation of the probability distribution of the inputs. After that, the pre-training method has to pass these information to the network that has to be pre-trained.
The most common application of pre-training is the DBN layer-wise pre-training via RBM. This pre-training technique drives each layer to represent the dominant factors of variation occurring in the input data [33]. The second method that will be presented in this chapter is based on a similar idea. the method is called *pre-training via linear autoencoders* [67]. It exploits *Principal Components Analysis* (PCA) [53]. In the following following these two methods are presented in detail.

5.2 HMM-based Pre-training

The first pre-training method we propose relies on an approximation of the actual data distribution in order to drive the network weights in a better region of the parameter space. The general idea is to exploit a linear simple model in order to "learn" an approximation of the distribution of the input data, and then transfer this knowledge from the linear model to the more complex neural network for sequences that has to be pre-trained. We do not want to limit the application of this method only to specific models (as in the case of RBM-based pre-training for DBN). On the contrary, our aim is to develop a technique that allows to use any simple linear model in order to pre-train any neural network for sequential data. For this reason, the method used to transfer the knowledge from the linear model to sequential Neural Networks has to be completely detached from the architecture. For this reason, the knowledge transfer between the two models will be performed by using a linear model to create a pre-training dataset. An instance of this method is the one that uses HMM (Section 2.1.2) as a linear simple model. As explained above, a linear HMM is firstly trained on the real sequences ("original data"). We have chosen this type of probabilistic model because it is very efficient in training, and it has shown to be effective on many sequence learning problems [74], including music modeling [17]. After training, the model is used to generate a fixed number of sequences that will

populate a new dataset for the pre-training phase ("smooth data"). The intuition is that the sequences generated by the linear model will constitute a smoothed, approximated version of the original sequences contained in the initial dataset, but will nevertheless retain the main structure of the data. In order to generate a set of sequences from the HMM, we first select the starting hidden state according to the learned initial state probability distribution. The first element of the sequence is then sampled according to the emission distribution associated with that hidden state, and the same process is iteratively repeated by selecting the next target state according to the learned state transition probabilities. A straightforward, nave implementation of the random sampling process can be obtained by first computing the cumulative probability distribution from the target distribution and then selecting the element corresponding to a random number drawn from the interval [0,1]. Notably, our pre-training procedure does not need any form of bootstrapping from the original sequences, because the smooth dataset is generated by sampling the HMM in a completely unconstrained fashion. The simplified, HMM-generated dataset is then used to pre-train the more powerful nonlinear model, with the aim of transferring the knowledge acquired by the HMM to the recurrent neural network. The recurrent network pre-training phase uses the same algorithm that is used for the normal training phase. Therefore, this pre-training method can be applied to any recurrent non-linear network. Moreover, this technique does not require to develop any ad-hoc algorithm to perform a pre-training of the network. This advantage also ensures complete independence from the architecture of the pre-trained model.
After completing the pre-training phase on the smooth dataset, the neural network is then fine-tuned using the original sequences, in order to allow the nonlinear model to extract a more complex structure from the data distribution. The pseudocode for the proposed HMM-based pre-training method is given in Algorithm 1, and a flow chart of the procedure is illustrated in Figure 5.1.

Notably, the introduction of the pre-training phase before fine-tuning does not significantly affect the computational cost of the whole learning procedure, because both learning and sampling in HMMs can be performed in an efficient way. In particular, our method performs three main steps during pre-training: train the HMM, generate the smooth dataset and pre-train the nonlinear network. The training phase for the HMM (step 3) is performed using the Baum-Welch algorithm, which has a complexity of order $O(N^2T)$ for each iteration and observation, where T is the length of the observation used to train the model, and N is the number of states in the HMM [77]. The smooth sequences generation (step 4) is performed using the Viterbi algorithm. For each generated sequence, this algorithm has a computational complexity of order $O((NF)^2T)$, where F is the size of the input at a single time step, T is the length of the generated sequence and N

Algorithm 1 Pseudocode for the proposed HMM-based pre-training. At the beginning, several parameters need to be initialized: n and l represent, respectively, the number and length of the sequences generated by the HMM, θ_{hmm} represents the training hyperparameters for the HMM (e.g., number of hidden states) and θ_{rnn} represents the training hyperparameters for the recurrent neural network (e.g., the number of hidden units and the learning rate).

1: $set \;\; n, l, \theta_{hmm}, \theta_{rnn}$;
2: $hmm \leftarrow train\_hmm(originalData, \theta_{hmm})$;
3: $smoothData \leftarrow sample(hmm, n, l)$;
4: $rnn \leftarrow random\_initialization$;
5: $rnn \leftarrow train\_rnn(smoothData, \theta_{rnn}, rnn)$;
6: $rnn \leftarrow train\_rnn(originalData, \theta_{rnn}, rnn)$;
7: $return(rnn)$;

is the number of states in the HMM. Finally, step 6 consists in pre-training the recurrent neural network. In order to perform the pre-training phase we exploit the standard training algorithm, therefore the complexity of this step depends on the type of network that we aim to use. Moreover, it should be noted that the improved initialization of the network weights could allow to speed up convergence during the fine-tuning phase. The number and length of the sequences generated by the HMM are important parameters for which it is difficult to make an operational choice. A rule of thumb is to choose them in accord with the training set statistics.

5.3 Pre-training via Linear Autoencoder

The second technique that we propose in order to pre-train an RNN model is called **Pre-Training via Linear Autoencoder**. Differently than the HMM-base approach, this method is developed specifically for pre-train RNN networks. The idea is to exploit the hidden state representation obtained by an autoencoder (defined by eqs. (2.3)) as initial hidden state representation for the RNN described by eqs. (2.8). This is implemented by initializing the weight matrices $\mathbf{A}$ and $\mathbf{B}$ of eqs. (2.8) by using the matrices that jointly solve eqs. (2.1) and eqs. (2.3), i.e. $\mathbf{A}$ and $\mathbf{B}$ (since $\mathbf{C}$ is function of $\mathbf{A}$ and $\mathbf{B}$). Specifically, we initialize the input weights of the RNN with $\mathbf{A}$, and its hidden weights with $\mathbf{B}$. Moreover, the use of symmetrical sigmoidal functions, which do give a very good approximation of the identity function around the origin, allows a good transferring of the linear dynamics inside RNN. For what concerns the output weights, we initialize it by using the best possible solution, i.e. the pseudoinverse of $\mathbf{H}$ (which is the matrix that is composed of all hidden representation of the items contained

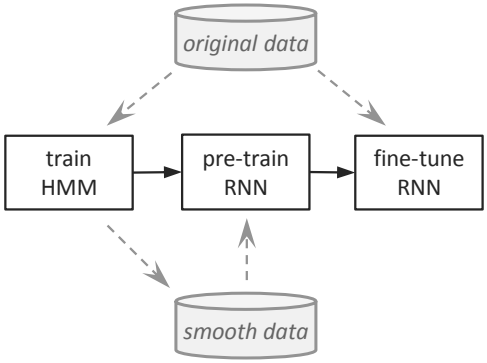

Figure 5.1: Flow chart of the proposed HMM-based pre-training method for RNN. The flow chart is the same if an RNN-RTRBM model is used in place of an RNN: it is sufficient to replace the label RNN with the label RNN-RBM in the picture (in fact, any model for sequences could in principle be used as an alternative to the RNN).

in the training set) times the target matrix $\mathbf{D}$ (the matrix composed of all targets $\mathbf{d}^q \in \mathcal{T}$), which does minimize the output squared error. Learning is then used to introduce nonlinear components that allow to improve the performance of the model.
More formally, let consider a prediction task where for each sequence $\mathbf{s}^q \equiv (\mathbf{x}_1^q, \mathbf{x}_2^q, \ldots, \mathbf{x}_{l_q}^q)$ of length l_q in the training set, a sequence $\mathbf{d}_q$ of target vectors is defined, i.e. a training sequence is given by $\langle \mathbf{s}^q, \mathbf{d}^q \rangle \equiv \langle (\mathbf{x}_1^q, \mathbf{d}_1^q), (\mathbf{x}_2^q, \mathbf{d}_2^q), \ldots, (\mathbf{x}_{l_q}^q, \mathbf{d}_{l_q}^q) \rangle$, where $\mathbf{d}_i^q \in \mathbb{R}^s$. Given a training set with N sequences, let define the target matrix $\mathbf{D} \in \mathbb{R}^{s \times L}$, where $L = \sum_{q=1}^{N} l_q$, as $\mathbf{D} = \left[\mathbf{d}_1^1, \mathbf{d}_2^1, \ldots, \mathbf{d}_{l_1}^1, \mathbf{d}_1^2, \ldots, \mathbf{d}_{l_N}^N\right]^\mathsf{T}$. The input matrix Ξ will have size $L \times nL$. Let p^* be the desired number of hidden units for the recurrent neural network (RNN). Then the pre-training procedure can be defined as follows: *i)* compute the linear autoencoder for Ξ using p^* principal directions, obtaining the optimal matrices $\mathbf{A}^* \in \mathbb{R}^{p^* \times n}$ and $\mathbf{B}^* \in \mathbb{R}^{p^* \times p^*}$; *i)* set input weights equal to $\mathbf{A}^*$ and the hidden weights equal to $\mathbf{B}^*$; *iii)* run the RNN over the training sequences, collecting the hidden activities vectors (computed using symmetrical sigmoidal functions) over time as rows of matrix $\mathbf{H} \in \mathbb{R}^{L \times p^*}$; *iv)* set the output weights as $\mathbf{H}^+\mathbf{D}$, where $\mathbf{H}^+$ is the (left) pseudoinverse of $\mathbf{H}$.

5.3.1 Computing an approximate solution for large datasets

In real world scenarios, the application of our approach may turn difficult because of the size of the data matrix. In fact, stable computation of principal

directions is usually obtained by SVD decomposition of the data matrix $\mathbf{\Xi}$, that in typical application domains involves a number of rows and columns which are easily of the order of hundreds of thousands. Unfortunately, the computational complexity of SVD decomposition is basically cubic in the smallest of the matrix dimensions. Memory consumption is also an important issue. Algorithms for approximate computation of SVD have been suggested (e.g., [63]), however, since for our purposes we just need matrices $\mathbf{V}$ and $\mathbf{\Lambda}$ with a predefined number of columns (i.e. p), here we present an ad-hoc algorithm for approximate computation of these matrices. Our solution is based on the following four main steps: *i)* divide $\mathbf{\Xi}$ in slices of k (i.e., size of input at time t) columns, so to exploit SVD decomposition at each slice separately; *ii)* compute approximate $\mathbf{V}$ and $\mathbf{\Lambda}$ matrices, with p columns, incrementally via truncated SVD of temporary matrices obtained by concatenating the current approximation of $\mathbf{V\Lambda}$ with a new slice; *iii)* compute the SVD decomposition of a temporary matrix via either its kernel or covariance matrix, depending on the smallest between the number of rows and the number of columns of the temporary matrix; *iv)* exploit QR decomposition to compute SVD decomposition. Algorithm 2 shows in pseudo-code the main steps of our procedure. It maintains a temporary matrix $\mathbf{D}$ which is used to collect incrementally an approximation of the principal subspace of dimension p of $\mathbf{\Xi}$. Initially (line 4) $\mathbf{D}$ is set equal to the last slices of $\mathbf{\Xi}$, in a number sufficient to get a number of columns larger than p (line 2). Matrices $\mathbf{V}$ and $\mathbf{\Lambda}$ from the p-truncated SVD decomposition of $\mathbf{D}$ are computed (line 5) via the KeCo procedure, described in Algorithm 3, and used to define a new $\mathbf{D}$ matrix by concatenation with the last unused slice of $\mathbf{\Xi}$. When all slices are processed, the current $\mathbf{V}$ and $\mathbf{\Lambda}$ matrices are returned. The KeCo procedure, described in Algorithm 3 , reduces the computational burden by computing the p-truncated SVD decomposition of the input matrix $\mathbf{M}$ via its kernel matrix (lines 3-4) if the number of rows of $\mathbf{M}$ is no larger than the number of columns, otherwise the covariance matrix is used (lines 6-8). In both cases, the p-truncated SVD decomposition is implemented via QR decomposition by the INDIRECTSVD procedure described in Algorithm 4. This allows to reduce computation time when large matrices must be processed [73]. Finally, matrices $\mathbf{V}$ and $\mathbf{\Lambda}^{\frac{1}{2}}$ (both kernel and covariance matrices have squared singular values of $\mathbf{M}$) are returned.

We use the strategy to process slices of $\mathbf{\Xi}$ in reverse order since, moving versus columns with larger indices, the rank, as well as the norm of slices, become smaller and smaller, thus giving less and less contribution to the principal subspace of dimension p. This should reduce the approximation error cumulated by dropping the components from $p+1$ to $p+n$ during computation [99]. As a final remark, we stress that since we compute an approximate solution for the principal directions of $\mathbf{\Xi}$, it makes no much sense to solve the problem given in eq. (3.10): learning will quickly compen-

Algorithm 2 Approximated $\mathbf{V}$ and $\mathbf{\Lambda}$ with p components

function SVForBigData($\mathbf{\Xi}, n, p$)
  $nStart = \lceil p/n \rceil$ ▷ Number of starting slices
  $nSlice = (\mathbf{\Xi}.columns/n) - nStart$ ▷ Number of remaining slices
  $\mathbf{D} = \mathbf{\Xi}[:, n * nSlice : \mathbf{\Xi}.columns]$
  $\mathbf{V}, \mathbf{\Lambda}$ =KeCo($\mathbf{D}, p$) ▷ Computation of $\mathbf{V}$ and $\mathbf{\Lambda}$ for starting slices
  for i in reversed(range($nSlice$)) **do** ▷ Computation of $\mathbf{V}$ and $\mathbf{\Lambda}$ for remaining slices
    $\mathbf{D} = [\mathbf{\Xi}[:, i * n:(i+1) * n], \mathbf{V\Lambda}]$
    $\mathbf{V}, \mathbf{\Lambda}$ =KeCo($\mathbf{D}, p$)
  end for
  return $\mathbf{V}, \mathbf{\Lambda}$
end function

Algorithm 3 Kernel vs covariance computation

function KeCo($\mathbf{M}, p$)
  if $\mathbf{M}.rows <= \mathbf{\Xi}.columns$ **then**
    $\mathbf{K} = \mathbf{M}\mathbf{M}^\intercal$
    $\mathbf{V}, \mathbf{\Lambda}_{sqr}, \mathbf{U}^\intercal$ =indirectSVD($\mathbf{K}, p$)
  else
    $\mathbf{C} = \mathbf{M}^\intercal\mathbf{M}$
    $\mathbf{V}, \mathbf{\Lambda}_{sqr}, \mathbf{U}^\intercal$ =indirectSVD($\mathbf{C}, p$)
    $\mathbf{V} = \mathbf{M}\mathbf{U}^\intercal\mathbf{\Lambda}_{sqr}^{-\frac{1}{2}}$
  end if
  return $\mathbf{V}, \mathbf{\Lambda}_{sqr}^{\frac{1}{2}}$
end function

Algorithm 4 Truncated SVD by QR

function indirectSVD($\mathbf{M}, p$)
  $\mathbf{Q}, \mathbf{R}$ =QR($\mathbf{M}$)
  $\mathbf{V}_r, \mathbf{\Lambda}, \mathbf{U}^\intercal$ =SVD($\mathbf{R}$)
  $\mathbf{V} = \mathbf{Q}\mathbf{V}_r$
  $\mathbf{S} = \mathbf{\Lambda}[1:p, 1:p]$
  $\mathbf{V} = \mathbf{V}[1:p, :]$
  $\mathbf{U}^\intercal = \mathbf{U}^\intercal[:, 1:p]$
  return $\mathbf{V}, \mathbf{\Lambda}, \mathbf{U}^\intercal$
end function

sate for the approximations and/or sub-optimality of $\mathbf{A}$ and $\mathbf{B}$ obtained by matrices $\mathbf{V}$ and $\mathbf{\Lambda}$ returned by Algorithm 2.

Chapter 6

Experimental Assessment

In this chapter, we report and discuss all the results obtained by testing the various methods and models presented in Chapter 3. The results are obtained by testing the models on the prediction task, over four benchmarking datasets containing polyphonic music sequences in midi format. The nature of these data makes the prediction task particularly interesting: the music sequences are complex, and follow a complex multi-modal distribution that makes it difficult to perform training on them. Moreover, prediction of these sequences requires a good capability by the network in managing long-term temporal dependencies, since the structure of a song could be very complex. This task has been already used for testing purposes in some pre-existing works ([17, 37]).

This chapter starts with a description of the experimental setting we have used for our tests in Section 6.1. We then present some experimental results obtained with simple LDS trained with the various techniques presented in Chapter 3. Moreover, we compare these results with the results of other similar nonlinear models, i.e., ESN and RNN (Section 6.2). Then, we present the results obtained by testing the LDS-based models, focusing in particular on LSN, which has shown to be the most interesting and promising model (Section 6.3). Finally, we present the results obtained by applying the proposed pre-training methods. We perform a comparison of them and discuss how to tune the various parameters of the models (Section 6.4).

6.1 Experimental Setting

In this section, we briefly present the prediction task we have used to carry out our evaluation (Section 6.1.1). We then describe the four benchmarking

datasets we have used (Section 6.1.2), and our reference performance metrics (Section 6.1.3).

All the experiments were run using the Theano software [3], on an Intel© Xeon© CPU E5-2670 @2.60GHz with 128 GB of RAM, equipped with an NVidia© K20 GPU.

6.1.1 Prediction Task

In this thesis, we focus on a prediction task over sequences that can be formalized as follows. We would like to learn a function $\mathcal{F}(\cdot)$ from multivariate bounded length input sequences to desired output values. Specifically, given a training set $\mathcal{T} = \{(\mathbf{s}^q, \mathbf{d}^q) \mid q = 1, \ldots, N,\ \mathbf{s}^q \equiv (\mathbf{x}_1^q, \mathbf{x}_2^q, \ldots, \mathbf{x}_{l_q}^q), \mathbf{d}^q \equiv (\mathbf{d}_1^q, \mathbf{d}_2^q, \ldots, \mathbf{d}_{l_q}^q),\ \mathbf{x}_t^q \in \mathbb{R}^n,\ \mathbf{d}_t^q \in \mathbb{R}^s\}$, we want to learn a function $\mathcal{F}(\cdot)$ such that $\forall q, t\ \mathcal{F}(\mathbf{s}^q[1,t]) = \mathbf{d}_t^q$, where $\mathbf{s}^q[1,t] \equiv (\mathbf{x}_1^q, \mathbf{x}_2^q, \ldots, \mathbf{x}_t^q)$. Our experimental assessment has been performed in the special case where $\mathbf{d}_k^q = \mathbf{x}_{k+1}^q$. This prediction task allows us to test different linear and non-linear systems. Different learning approaches have been considered for both linear and non-linear dynamical systems, as described in the following. for each learning technique has performed several tests, in order to evaluate strengths and weaknesses of different approaches.

6.1.2 Datasets

In order to assess the prediction abilities of the various models, we adopted the prediction task presented in Section 6.1.1, over polyphonic music sequences. Each music sequence consists of a sequence of binary arrays of 88 dimensions representing the 88 notes spanning the whole piano range (from A0 to C8). In particular, each binary value is set to 1 if the note is played at the current time step, and 0 otherwise. The number of notes played simultaneously varied from 0 to 15. The output prediction is represented using the same format of the input. The polyphonic music datasets considered in our study contain different musical genres, which involve varying degrees of complexity in the temporal dependencies that are relevant for the prediction task. The Nottingham dataset contains folk songs, characterized by a small number of different chords and a redundant structure; the Piano-midi.de dataset is an archive of classic piano music, containing more complex songs with many different chords; the Muse Data and JSB Chorales datasets contain, respectively, piano and orchestral classical music; moreover, the JSB chorales are redundant and all composed by a single author, so the style of the songs is largely shared by different patterns. In Table 6.1 we report the main datasets statistics, including the size of the training and test sets, and the maximum, minimum, and average length of the contained sequences. We decided to perform prediction task on these datasets because this allows us to obtain benchmarking results. Indeed, several other works, that

	Set	# Samples	Max length	Min length	Avg Length
Nottingham	**Training**	694	641	54	200.8
	Test	170	1495	54	219
	Validation	173	1229	81	220.3
Piano-midi.de	**Training**	87	4405	78	812.3
	Test	25	2305	134	694.1
	Validation	12	1740	312	882.4
MuseData	**Training**	524	2434	9	474.2
	Test	25	3402	70	554.5
	Validation	135	2523	94	583
JSB Chorales	**Training**	229	259	50	120.8
	Test	77	320	64	123
	Validation	76	289	64	121.4

Table 6.1: Datasets statistics, including the number of sequences contained in each dataset.

introduce the most powerful models that deal with sequential data, use the same datasets on this task to perform the evaluation of the model [17] [37]. Thanks to this we can compare the obtained results with several other models. In particular, we use as a baseline the results obtained by RNN-RBM model in [17], that achieved the state-of-the-art on the polyphonic music prediction task. Due to computational time constraints, for some experiments, we have just used the Nottingham and Piano datasets because these two datasets contain a very different type of data. Moreover, the results obtained on these two sets turn out to be a very good guidance in order to evaluate how good a model performs.

6.1.3 Performance Metric

To measure the performance of each model on the prediction task, we adopted the same accuracy metric used in [13], and described in [10]. The idea is to compute an accuracy metric based on three values:

- TP - True Positive: is the number of correctly predicted outputs.
- FP - False Positive: is the number of note-off examples predicted as note-on.
- FN - False Negative: is the number of note-on examples predicted as note-off

Given a sequence, these three values are computed by summing the obtained values at each time step. For each sequence the accuracy is defined as follows:

$$Accuracy = \frac{TP}{FP + FN + TP}. \tag{6.1}$$

The results obtained over a dataset is computed by calculating the average accuracy over all the sequences it contains.

6.2 Polyphonic Music Prediction Task with LDS

In this section, we are going to compare the capability of the LDS model against similar models trained with different training techniques. In addition to LDS, in order to make a meaningful comparison of the results, we tested also nonlinear models that have the same topology of LDS, and that have been trained by using the following techniques:

$\mathcal{N}_1$: Adopt the Echo State Network training procedure that randomly initializes matrices $\mathbf{A}$ and $\mathbf{B}$ according to the Echo State Property and only trains the output weights using pseudo-inverse of the hidden representations, i.e. compute $\mathbf{C} = \mathbf{D}\mathbf{H}^{+}$, where $\mathbf{D} = [\mathbf{d}_1^1, \mathbf{d}_2^1, \ldots, \mathbf{d}_{l_N}^N]$ and, $\mathbf{H} = [\mathbf{h}_1^1, \mathbf{h}_2^1, \ldots, \mathbf{h}_{l_N}^N]$ is the matrix that collects all the hidden representations obtained by running the system, with random matrices $\mathbf{A}$ and $\mathbf{B}$, over all sequences in the training set.

$\mathcal{N}_2$: Perform SGD with respect to the regularized error function $\tilde{E}_{\mathcal{T}}$, that is defined as:

$$\tilde{E}_{\mathcal{T}} = \frac{1}{NL} \sum_{q=1}^{N} \sum_{j=1}^{l_q} (\mathbf{d}_j^q - \mathbf{o}_j^q)^2 + R_1 + R_2,$$

where $L = \sum_{q=1}^{N} l_q$, and

$$R_1 = \sum_{i}^{m} \sum_{j}^{n} |\mathbf{A}_{ij}| + \sum_{i}^{m} \sum_{j}^{m} |\mathbf{B}_{ij}| + \sum_{i}^{s} \sum_{j}^{m} |\mathbf{C}_{ij}|,$$
$$R_2 = \sum_{i}^{m} \sum_{j}^{n} \mathbf{A}_{ij}^2 + \sum_{i}^{m} \sum_{j}^{m} \mathbf{B}_{ij}^2 + \sum_{i}^{s} \sum_{j}^{m} \mathbf{C}_{ij}^2,$$

with standard random initialization for matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. This corresponds to a standard RNN training procedure.

$\mathcal{N}_3$: Perform SGD with respect to the regularized error function $\tilde{E}_{\mathcal{T}}$. The matrices $\mathbf{A}$ and $\mathbf{B}$ are initialized according to the procedure proposed in Section 3.2.2 for the linear auto-encoder, and $\mathbf{C}$ with $\mathbf{D}\mathbf{H}^{+}$, where $\mathbf{H}$ is obtained by first running eq. (4.1) with initialized matrices $\mathbf{A}$ and $\mathbf{B}$. This approach corresponds to the pre-training one proposed in Section 5.3. More details about the results obtained by this pre-training approach are reported in Section 6.4.2

$\mathcal{N}_4$: Perform SGD with respect to the regularized error function $\tilde{E}_{\mathcal{T}}$, starting from matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ obtained by the following pre-training approach: *i)* train a linear HMM over the training set $\mathcal{T}$; *ii)* using the obtained HMM, generate N_{pt} sequences that will constitute the

pre-training dataset $\mathcal{T}_{pr}$; *iii)* perform SGD with respect to the regularized error function $\tilde{E}_{\mathcal{T}_{pr}}$ using $\mathcal{T}_{pr}$ and standard random initialization for matrices **A**, **B**, and **C**. This approach corresponds to the pre-training one proposed in Section 5.2. More information about the results obtained with this method, and the techniques used to tune the parameters is reported in Section 6.4.1.

These linear and nonlinear methods could be divided into two main sets based on the fact that there is or not the supervised learning of the hidden state mapping. Indeed, some of this methods use the supervision only for the hidden to output mapping via pseudo-inverse. It is important to make this distinction because we have to take into account that for these last methods the unsupervised linear or non-linear projections are used to define the current (hidden) state. The approaches that use the supervision only for the hidden to output mapping are: LDS-$\mathcal{L}_1$, LDS-$\mathcal{L}_2$, and RNN-$\mathcal{N}_1$. Because of the unsupervised projections, a larger state space is supposedly needed for these approaches to get good performances. Subsequently, we present experimental results obtained for SGD-based approaches, i.e. LDS-$\mathcal{L}_3$, RNN-$\mathcal{N}_2$, RNN-$\mathcal{L}_3$, and RNN-$\mathcal{N}_4$, where a smaller state space is sufficient.

6.2.1 Results of approaches using unsupervised projections

As discussed before, approaches that use random or unsupervised projections, in principle, require larger state spaces in order to get good performances. A profitable size for the state space also depends on the complexity of the dataset. We have explored this issue by performing experimental tests using 500, 1000, 1500 hidden units for the Nottingham dataset, and 500, 1000, 2000 hidden units for the Piano-midi.de dataset. The use of 2000 units takes into account the higher complexity of the Piano-midi.de dataset. The considered approaches are LDS-$\mathcal{L}_1$, LDS-$\mathcal{L}_2$, and RNN-$\mathcal{N}_1$.
Experimental results for the Nottingham dataset are shown in Figure 6.1, while the results for the Piano-midi.de dataset are shown in Figure 6.2.
In general, it seems that random projection-based approaches (i.e. LDS-$\mathcal{L}_1$ and RNN-$\mathcal{N}_1$) are insensitive to the size of the state space, while this seems not to be the case for LDS-$\mathcal{L}_2$. In fact, the autoencoder-based training approach seems to be sensitive to the state space size. This fact can be explained by considering the nature of the training technique. Indeed, the autoencoder-based method exploits the SVD decomposition in order to extract the most relevant information from input and previous states. What happens by increasing the state space size is that those features that have lower variance will be added to the state representation. This allows to collect more relevant information in the state. In other words, by using this approach with increasing size of state space, is very likely that some new relevant features enter the state representation. Therefore, since the size of the state space is $\ll rank(\mathbf{\Xi})$, the approach will improve its performance

(at least on the training set).
For both datasets it seems that a further increase of the state space size would keep improving performances. Finally, it can be noticed that on the Piano-midi.de dataset the RNN-$\mathcal{N}_1$ approach seems to show some overfitting with the increase of state space size.

6.2.2 Results of approaches using supervision and pre-training

Here we present the results obtained for approaches exploiting supervision and pre-training, i.e. LDS-$\mathcal{L}_3$, RNN-$\mathcal{L}_2$, RNN-$\mathcal{N}_3$, and RNN-$\mathcal{N}_4$. Actually, all approaches use a form of pre-training except for RNN-$\mathcal{L}_2$, which uses random initialization for the weights. Since full supervision is used for all approaches, much smaller state spaces are used. Figure 6.3 and Figure 6.4 reports results (on training and test set, respectively), for both datasets, obtained for approaches LDS-$\mathcal{L}_3$, RNN-$\mathcal{L}_2$, and RNN-$\mathcal{N}_3$ using the same settings and model selection procedure described in [67].
From Figure 6.4 it is clear that the use of pre-training exploiting the linear autoencoder allows to achieve better results in less epochs. Indeed, in both datasets the pre-trained versions of the RNN obtain a higher accuracy regardless the number of hidden units. Moreover, a very good level of accuracy can be reached in a lower number of epochs. The LDS-$\mathcal{L}_3$ approach (i.e., LDS pre-trained via linear autoencoder initialization and then fine-tuned via SGD) on Nottingham dataset achieved results similar to a non-linear RNN model. However, the same approach has a totally different behavior on Piano-midi.de. In that case, after a few training epochs, the accuracy quickly decreases under the accuracy achieved by randomly-initialized systems. This behavior is due to the high complexity of sequences in Piano-midi.de.

6.2.3 Discussion

In this section, we try to summarize the experimental results obtained for the different approaches. In Figure 6.5 we report the performances, after model selection via the validation sets, of all the studied approaches on the two considered datasets. The reported performances for RNN-$\mathcal{N}_2$ is taken from [17], since those values are better than the ones we were able to achieve. The results obtained on the Nottingham dataset seem to show that better performances can be obtained thanks to pre-training. In fact, both linear and non-linear approaches with pre-training return good results, while all the remaining approaches get more or less the same lower performance, regardless of the linearity or non-linearity of the considered model. Among approaches that exploit pre-training, non-linear models get a slighter better performance. It must, however, be noticed that the computationally less

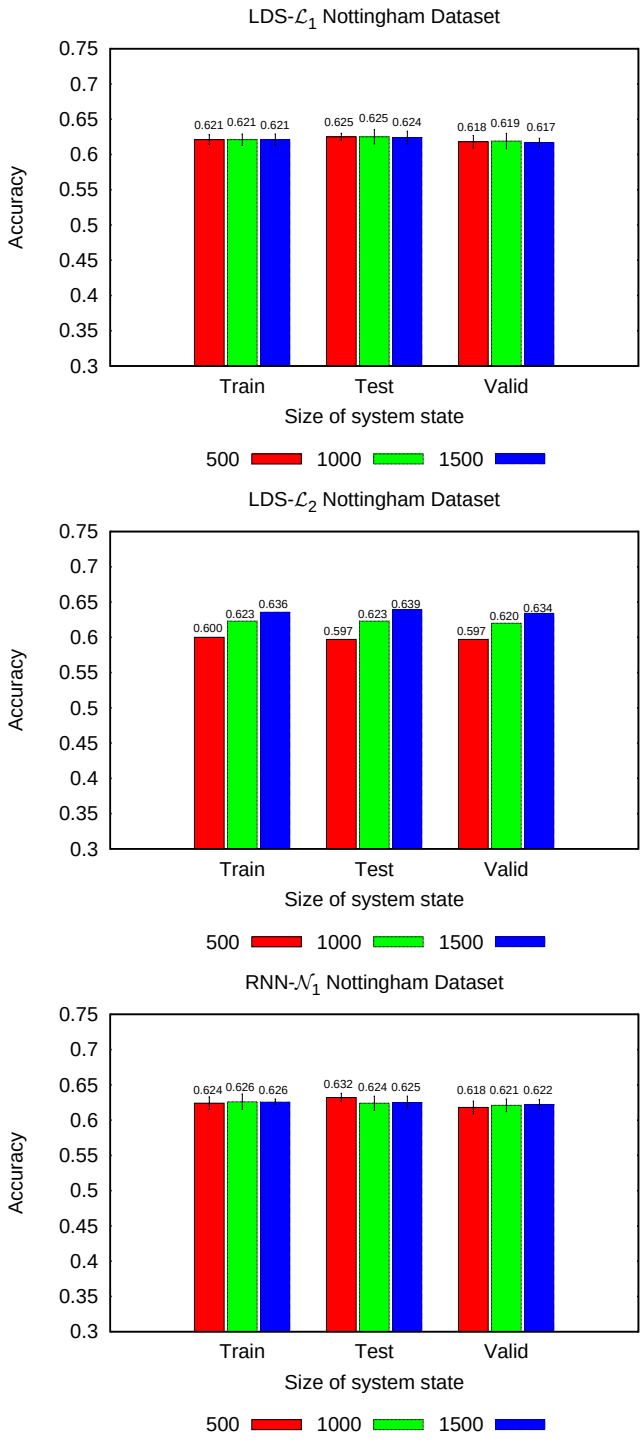

Figure 6.1: Results achieved for the Nottingham dataset. Each chart shows the accuracy achieved by training LDS-$\mathcal{L}_1$, LDS-$\mathcal{L}_2$, and RNN-$\mathcal{N}_1$ with different state space sizes. For methods LDS-$\mathcal{L}_1$ and LDS-$\mathcal{L}_2$, that use random matrices, the standard deviation is reported by using the black lines on the top of the bars.

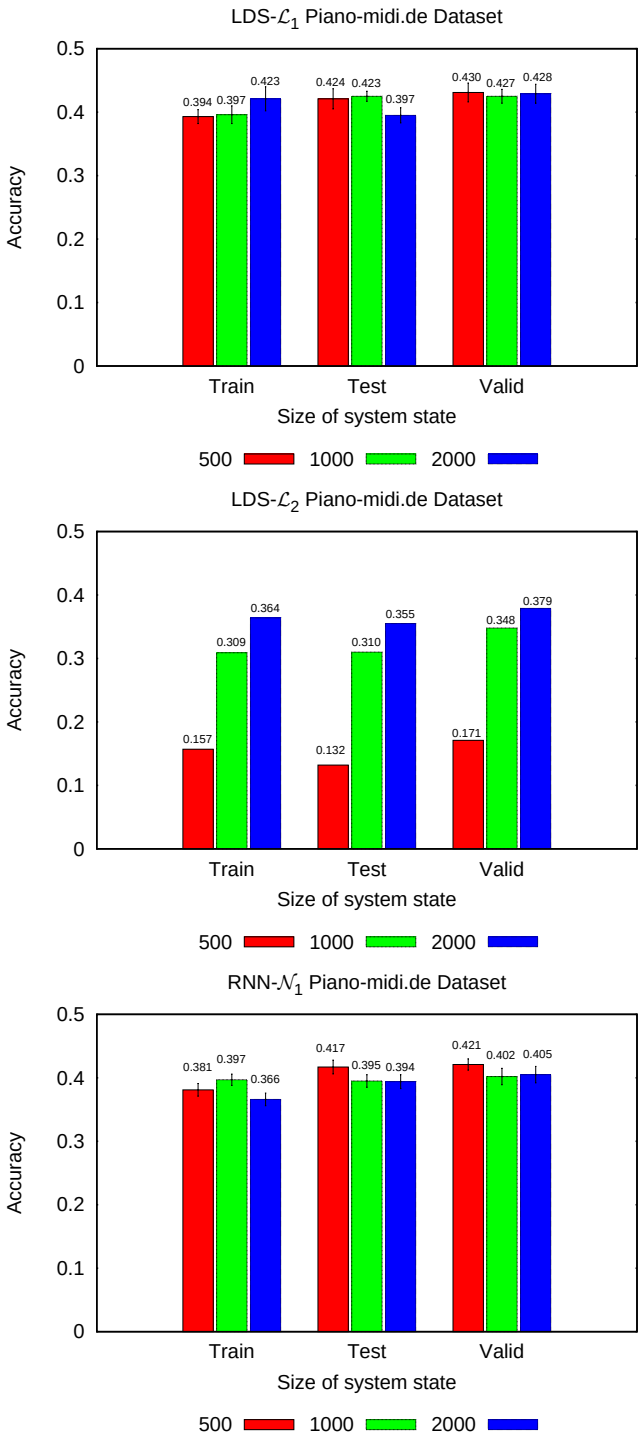

Figure 6.2: Results achieved by LDS Piano-midi.de dataset. Each chart shows the accuracy achieved by training LDS-$\mathcal{L}_1$, LDS-$\mathcal{L}_2$, and RNN-$\mathcal{N}_1$ with different state space sizes. For methods LDS-$\mathcal{L}_1$ and LDS-$\mathcal{L}_2$, that use random matrices, the standard deviation is reported by using the black lines on the top of the bars.

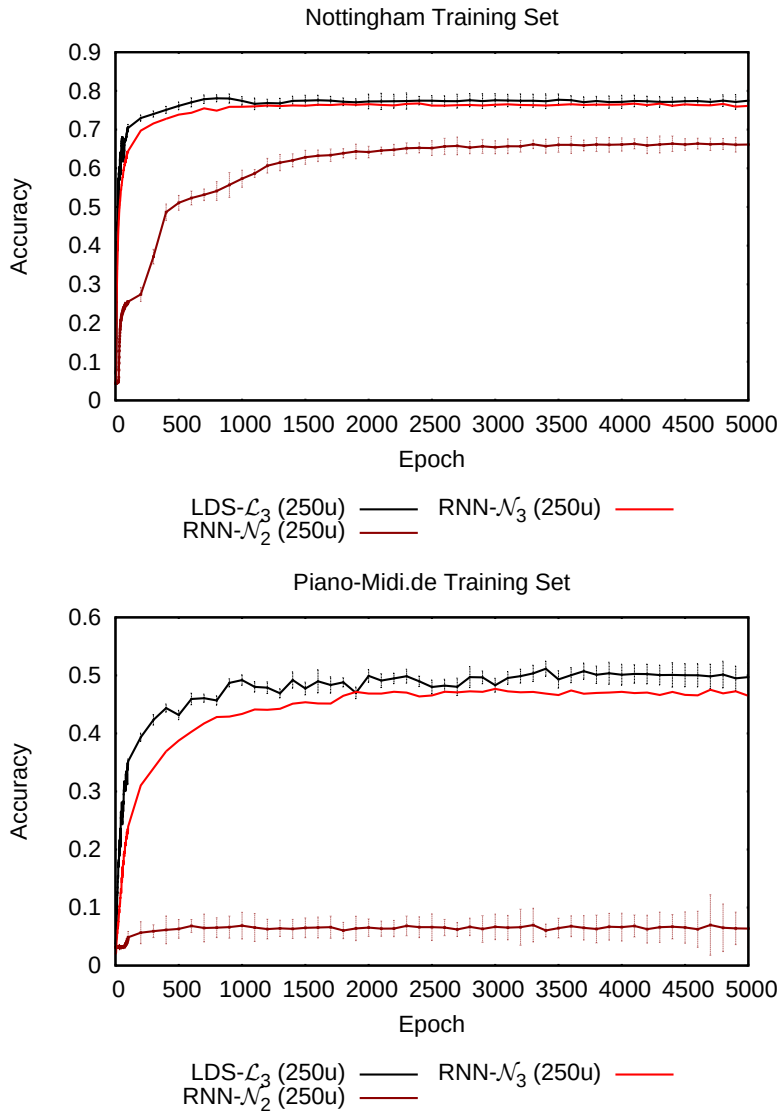

Figure 6.3: Train curves on the two datasets by models LDS-$\mathcal{L}_3$, RNN-$\mathcal{N}_2$ and RNN-$\mathcal{N}_3$. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards. For methods LDS-$\mathcal{L}_3$ and RNN-$\mathcal{N}_2$, that use random matrices, the standard deviation is reported as well.

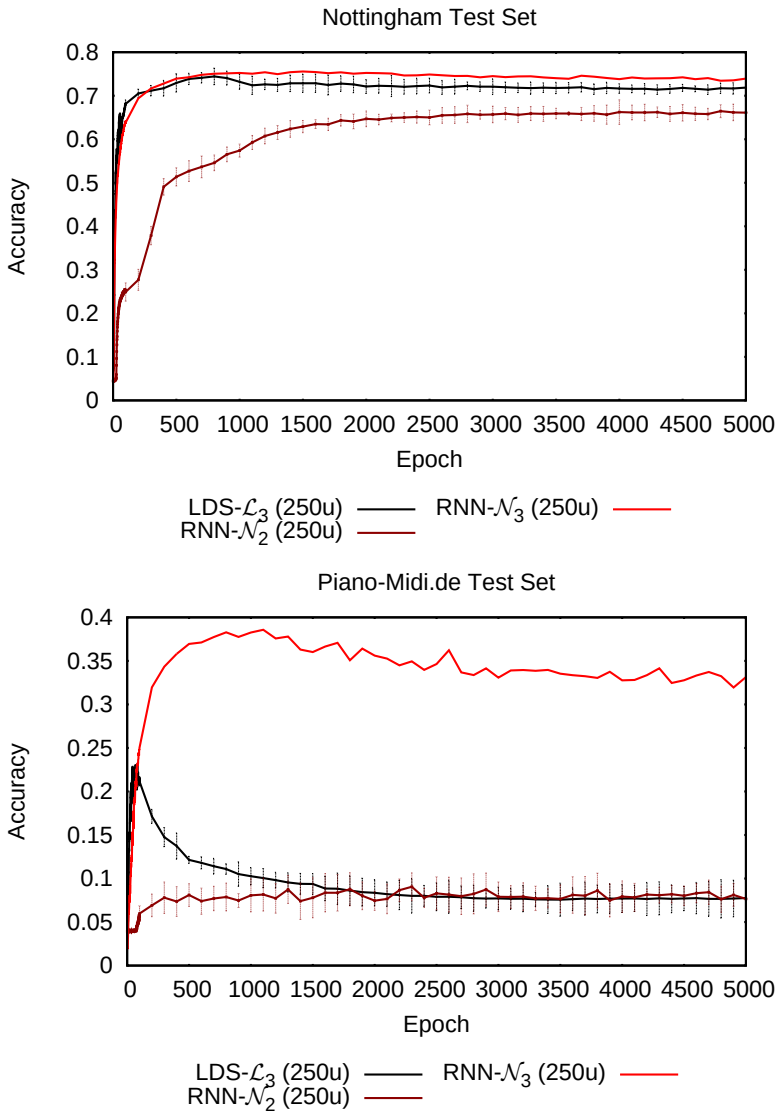

Figure 6.4: Test curves on the two datasets by models LDS-$\mathcal{L}_3$, RNN-$\mathcal{N}_2$ and RNN-$\mathcal{N}_3$. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards. For methods LDS-$\mathcal{L}_3$ and RNN-$\mathcal{N}_2$, that use random matrices, the standard deviation is reported as well.

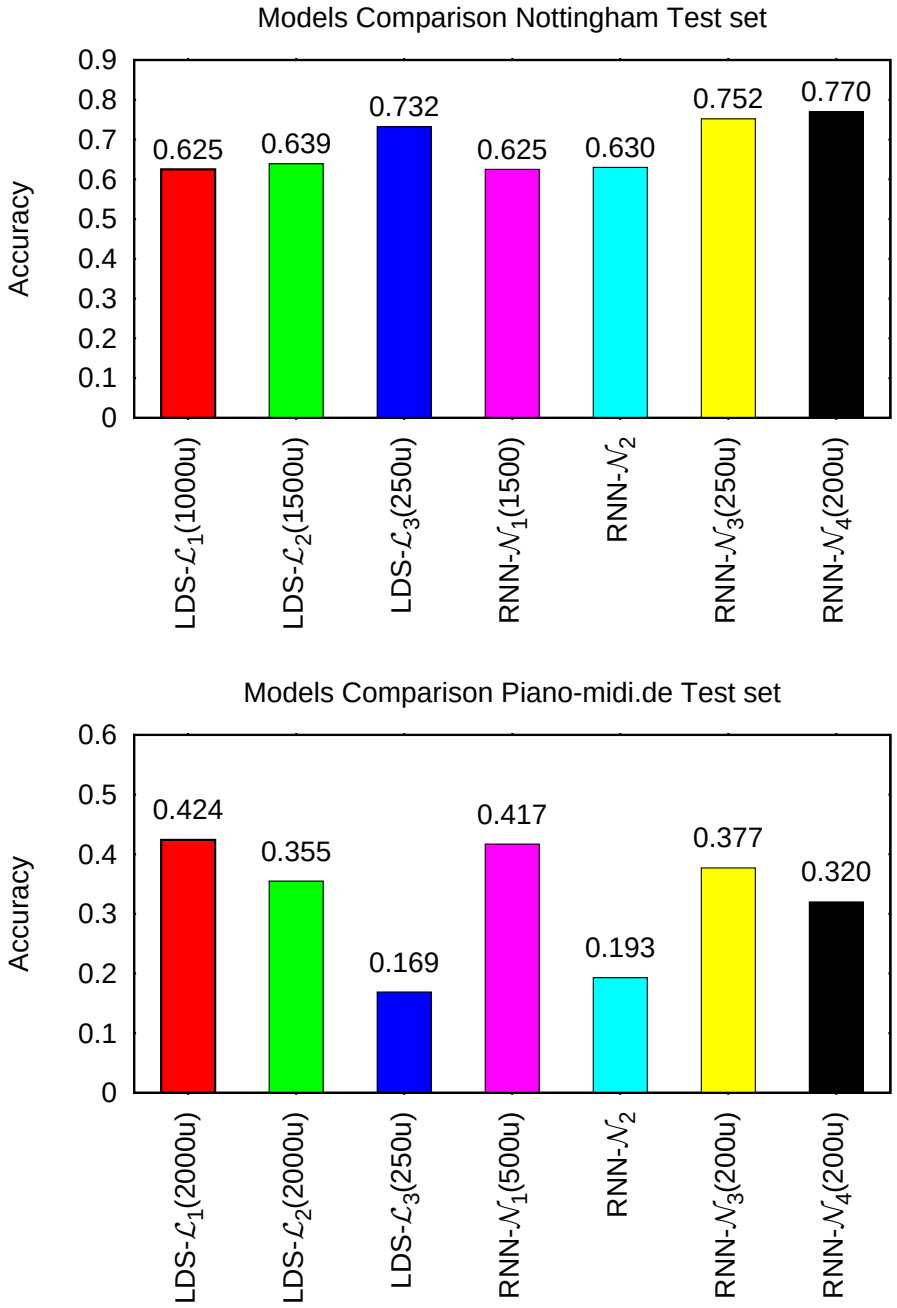

Figure 6.5: Comparison of the performances obtained by the studied approaches. For each model the charts report the accuracy achieved on the test set. The number reported in brackets is the number of hidden units selected by model selection using the validation set. The results for RNN-$\mathcal{N}_2$ are taken from [17].

demanding linear model with pre-training already returns a quite good performance.
For the Piano-midi.de, the situation seems to be quite different, since the best performers are based on random projections, independently from the linearity or non-linearity of the model. Pre-training based approaches seem, for this dataset, to be less effective. This may be due to the fact that the Piano-midi.de owns a higher complexity, and it is likely that far larger state spaces are needed to reach better performances. In fact, it can be noted that approaches based on random projections, either linear or non-linear, exploit a state space that is tenfold larger than the ones used by SGD-based approaches.
Overall, it is clear that the adoption of a pre-training approach allows to systematically improve over the standard RNN approach, i.e. RNN-$\mathcal{N}_2$.
From a computational point of view, linear approaches are far more efficient than non-linear ones, with the only exception of RNN-$\mathcal{N}_1$, that does not require training of the hidden state mapping.

6.3 Polyphonic Music Prediction Task With LDS-based Models

In this section, we present and discuss the results obtained by testing the models presented in Chapter 4 on the same polyphonic music prediction task presented above. We firstly discuss about the preliminary application of the LSN model by using single configurations. Then, the configurations that have obtained the best results are tested more in deep. Moreover, we evaluate and discuss the time required to perform the training of the model by using different sets of parameters.
After that, we present the results obtained by testing the linear co-learning architectures (proposed in Section 4.3). The goal is to empirically verify the theoretical results of our study on this type of model (Section 4.3.1). Finally, we present the results obtained by UniNet.

6.3.1 Experimental results obtained by LSN

In order to test the LSN, we used two datasets: Nottingham and Piano-midi.de. The software implementing the LSN model is realized in Python. The training of each LDS is performed independently and in parallel. The number of LDSs trained in parallel depends on the amount of memory available on the machine used for performing the tests.

Comparison among different configurations

In this section, we compare the various proposed configurations for LSN. We proposed four configurations: **Var**, **Over**, **Select**, and **Error**. In order

to perform a fair comparison among them, we decided to use the same parameters by varying just the configuration. We firstly decide to apply to the model a single configuration at a time, in order to evaluate the impact of each configuration on the performance of the model. Moreover, testing all possible combinations of variants turns out to be unfeasible due to the high number of possible combinations (we recall that each variant can be used regardless that the others are used or not). We have tested both datasets by using M LDSs with state space of 200 dimensions. For Nottingham dataset we have set $M = 30$, whilst for Piano-midi.de we have set $M = 10$. We have decided to use these two parameters settings because in our preliminary tests they showed the best results in both datasets (Table 6.2). In the next section, we will give a more formal and complete discussion about the tuning of model parameters. For what concerns the **Var** configuration, the state size is set as described in equation (4.17). The comparison among the various configurations is performed by taking care of the obtained accuracy, and the time required to perform the training phase. Indeed, some configurations like **Error** and **Var**, may slow down the training process. In particular, it is interesting to notice that the amount of memory used by the **Error** configuration is very similar to the memory occupation used by the others configurations, mainly thanks to the reformulation of the error function given in equation (4.26).
Figures 6.6, 6.7 and 6.8 show the results obtained by the LSN by using the basic configuration (Section 4.1) in conjunction with one of the four different configuration variants. The results show that the **Var** configuration turns out to be the most effective. Indeed, the obtained results suggest that the model that uses this option significantly improves the prediction accuracy. For what concerns the other three options, the benefits that the model gains by using one of them are not meaningful, and sometimes the results show a lower accuracy compared with the basic configuration. Therefore, we decided not to apply them to the polyphonic music prediction task.

For what concerns the computational time demand of the various configurations, Figure 6.9 shows that the **Var** configuration slows down significantly the training process. The reason why the training phase becomes slower is the fact that each system has a different size of the state, and accordingly to equation (4.17), the state size increases as the number of LDSs. Training of an LDS that has larger state space requires more time. Moreover, also the size of vector $\mathbf{z}_t$ will increase, and this leads to a more complex training phase, even for the second layer of the model. This is due to the fact that training the weights matrix $\mathbf{D}$ involves the computation of the pseudoinverse of a matrix that has a number of rows that increases as the size of the LDS state space.
The time required to perform training by using one of the other three variants is still higher than the basic configuration. In particular, the increasing

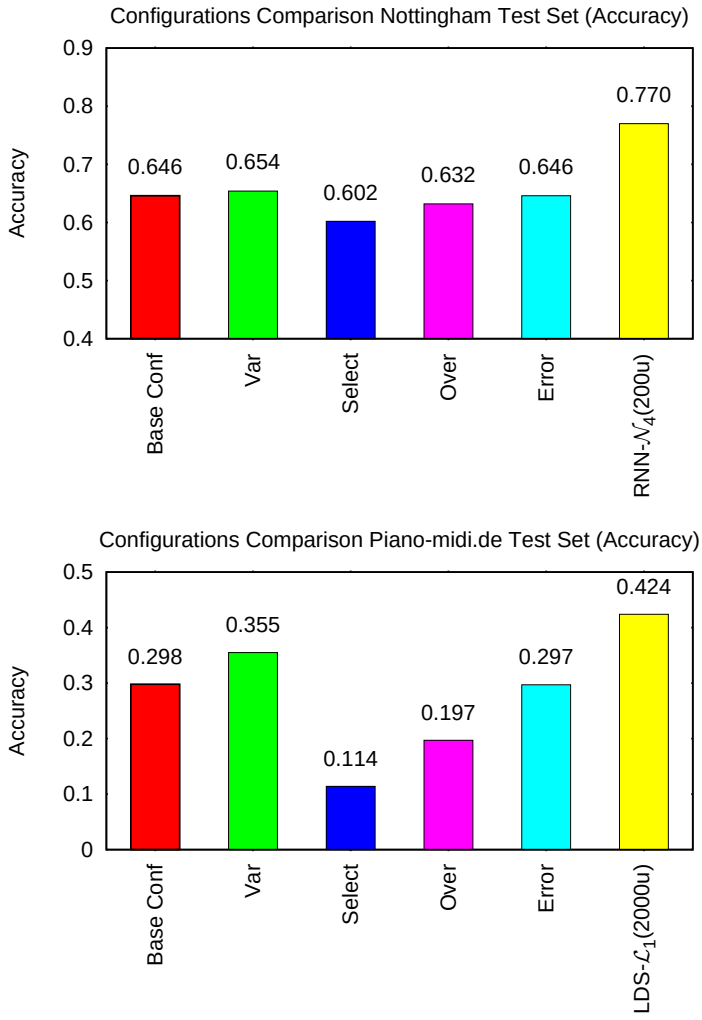

Figure 6.6: Accuracy obtained by LSN on Nottingham test set (top) and on Piano-midi.de test set (bottom) by using different configurations. Each configuration has been tested by using the base configuration in conjunction with a single configuration (**Var**, **Select**, **Over** and **Error**). For the Nottingham dataset the model is trained by using paramaters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

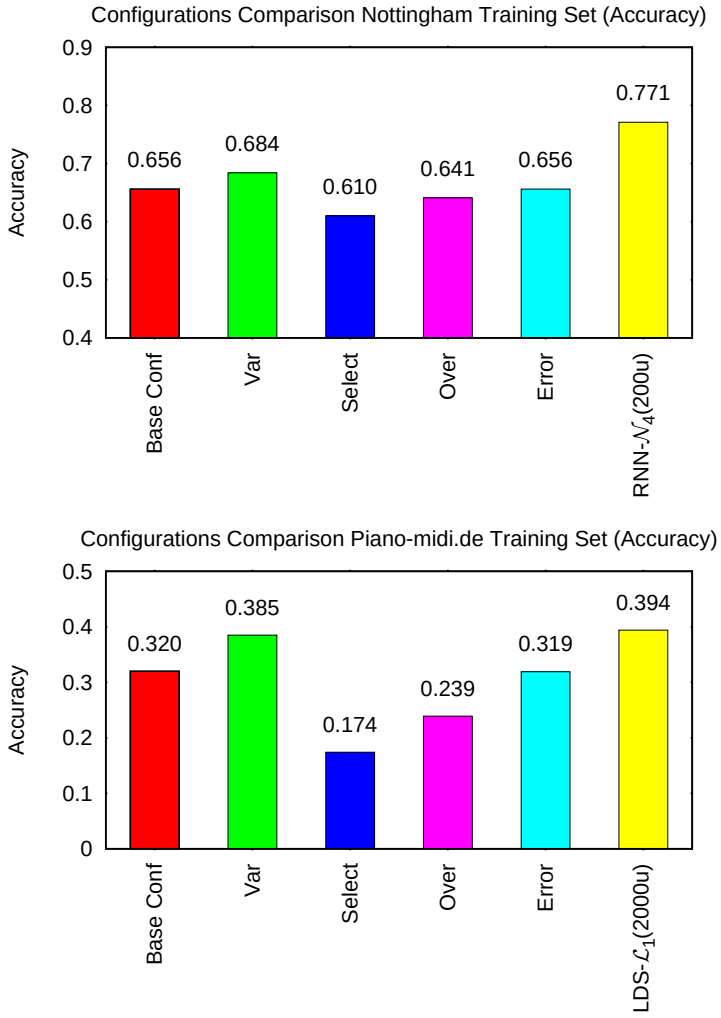

Figure 6.7: Accuracy obtained by LSN on Nottingham training set (top) and on Piano-midi.de training set (bottom) by using different configurations. Each configuration has been tested by using the base configuration in conjunction with a single configuration (**Var**, **Select**, **Over** and **Error**). For the Nottingham dataset the model is trained by using paramaters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

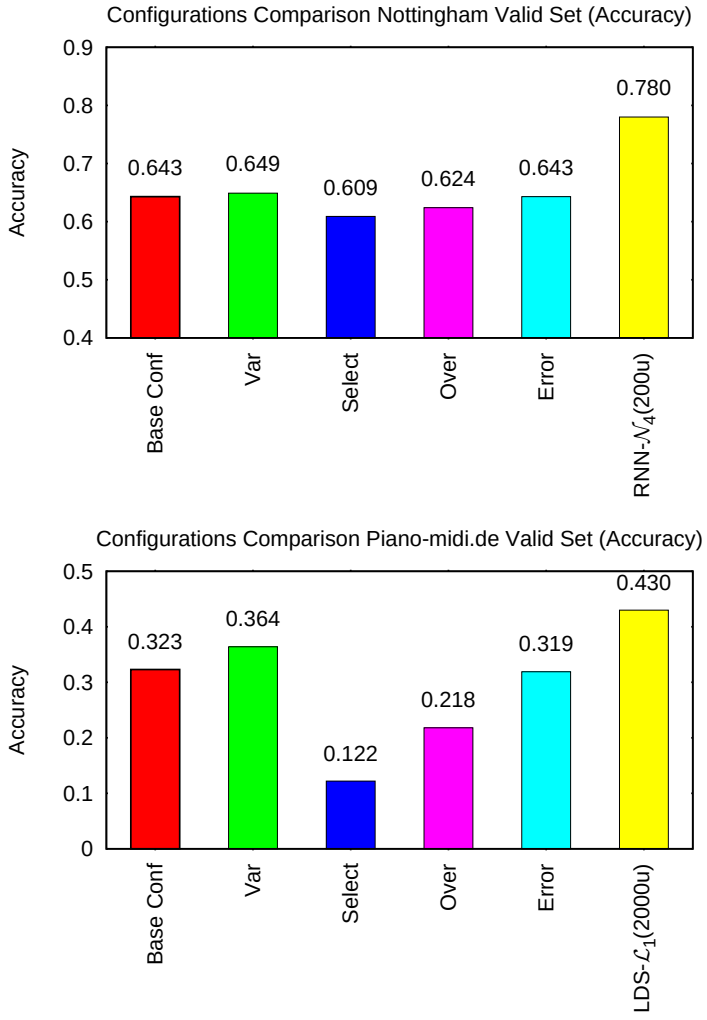

Figure 6.8: Accuracy obtained by LSN on Nottingham validation set (top) and on Piano-midi.de validation set (bottom) by using different configurations. Each configuration has been tested by using the base configuration in conjunction with a single configuration (**Var**, **Select**, **Over** and **Error**). For the Nottingham dataset the model is trained by using paramaters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

in time required to perform the operation used in the **Error** configuration turns out to be correlated with the lengths of the sequences contained in the dataset. Indeed, the difference between the time required to train the LSN with the basic configuration and the time needed to perform training of an LNS exploiting the **Error** configuration, dramatically increases in Piano-midi.de. This behavior is related to the fact that the error is computed incrementally during each time step, and Piano-midi.de contains longer sequences. The other two variants show a similar behavior on both datasets, and the time required for training phase is similar to the time required by the basic configuration.

We have also assessed the behavior of the LSN model when more than one variants are activated at the same time. Considering the large number of possible combinations and the high computational burden of each single execution, we decided to test only the combinations of two variants where one is **Var**. This choice is justified by the fact that **Var** is the most promising one. Figures 6.10, 6.11 and 6.12 compares the results obtained by the following configurations: **Var**+**Over**, **Var**+**Select**, **Var**+**Error**. The obtained results show that mixing together more than one option does not turn out to be very effective. Indeed, the obtained accuracies are very close (and sometimes lower than) to the accuracy obtained by just using the **Var** option. Anyway, by using the **Var**+**Error** configuration allows to achieve bit higher results on both datasets.

Tuning Parameters

The LSN model has two parameters that have to be tuned in order to optimize the performance of the model. The first parameter is the number of systems (M) that compose the first layer of the architecture. As explained in Section 4.1, given the number of desired LDSs, each system will be trained with an ad-hoc sub-dataset, that is a subset of the original training set. The input sequences that populate each sub-dataset are chosen based on the configuration that is in use at given time step. Both in the basic configuration, and in the **Over** configuration, the sequences are chosen by considering their lengths and the bounds l^i_{min} and l^i_{max} of the considered LDS. In some particular cases, some of these sub-datasets may be empty. When this happens the corresponding LDS will not be inserted in the first layer. Therefore, the number of LDSs in the first layer of the LSN will be always $\leq M$. Finding the best value for parameter M require also to take care about two important aspects: *i)*The number of LDSs determines also the size of the second layer $\mathbf{z}_t$; indeed, it collects the state of all LDSs, so the number of LDSs (and the size of their state space) will determine the size of $\mathbf{z}_t$. The size of the second layer significantly affects the time and the memory required to train the whole model. Choosing a large value for M may turn the training phase to be very slow and highly memory demand-

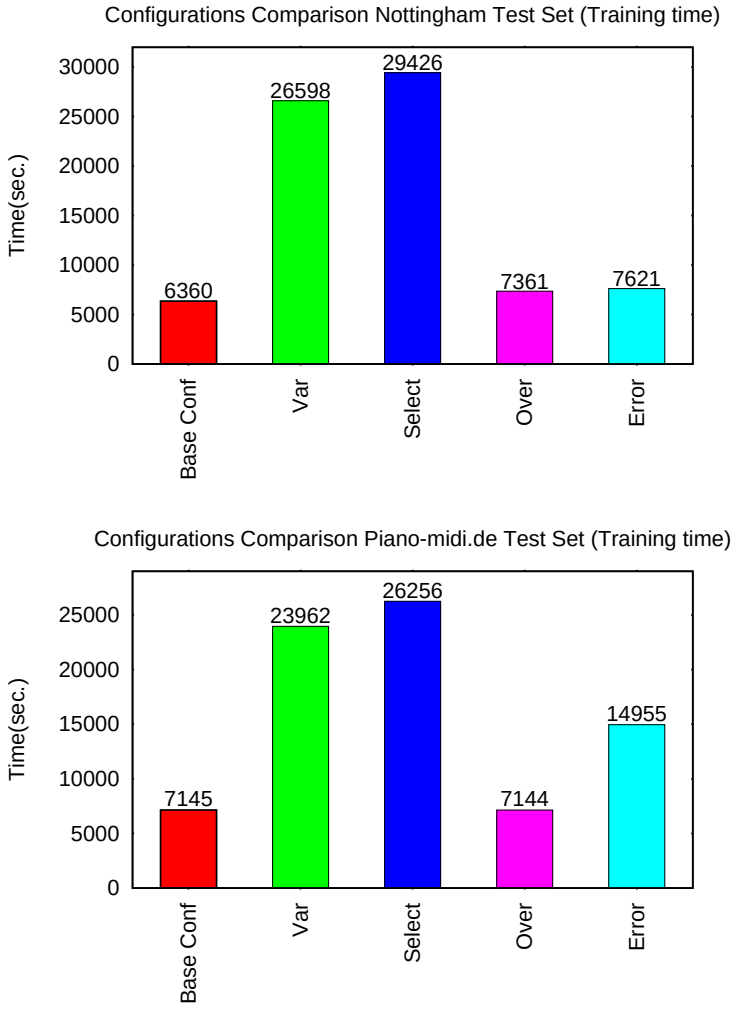

Figure 6.9: Time required to train the LSN model on Nottingham dataset (top) and on Piano-midi.de dataset (bottom) by using different configurations. Each configuration has been tested by using the base configuration in conjunction with a single configuration (**Var**, **Select**, **Over** and **Error**). For the Nottingham dataset the model is trained by using paramaters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

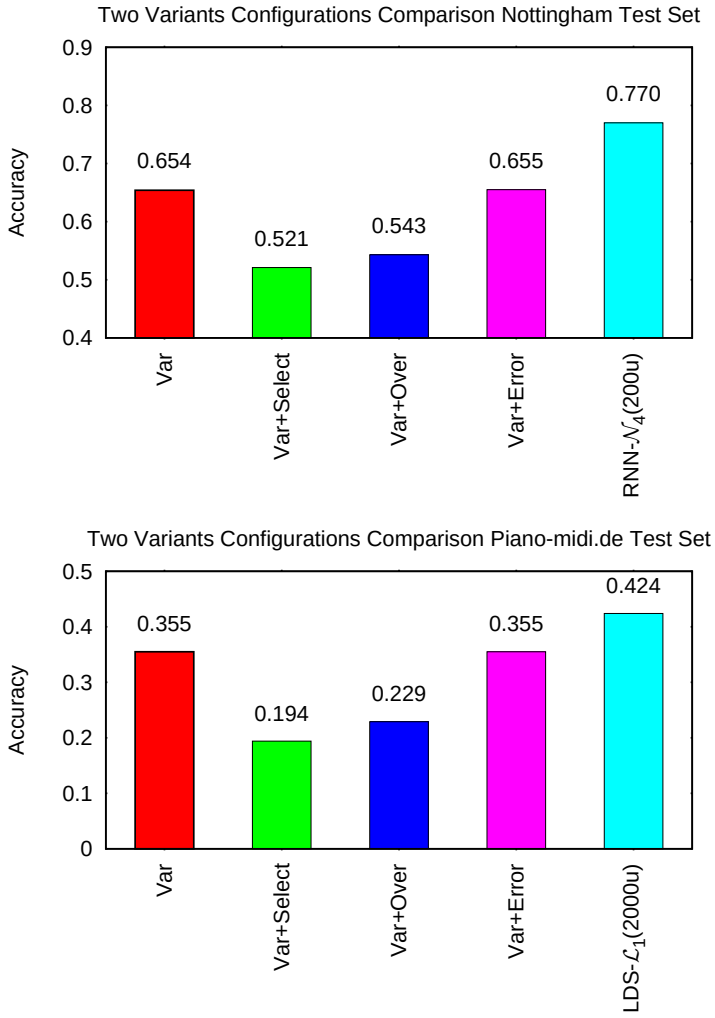

Figure 6.10: Accuracy obtained by LSN model on Nottingham test set (top) and on Piano-midi.de test set (bottom) by using different configurations that combine two variants. Each configuration has been tested by using the **Var** configuration in conjunction with a single configuration variants (**Select**, **Over** or **Error**). For the Nottingham dataset the model is trained by using parameters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

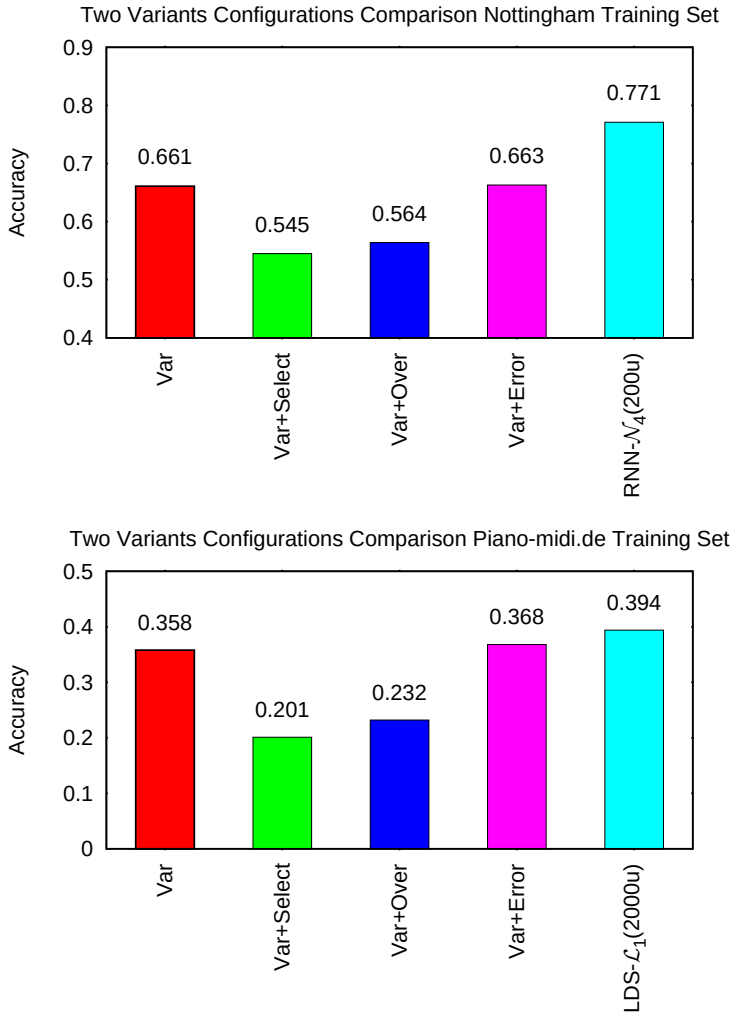

Figure 6.11: Accuracy obtained by LSN model on Nottingham training set (top) and on Piano-midi.de training set (bottom) by using different configurations that combine two variants. Each configuration has been tested by using the **Var** configuration in conjunction with a single configuration variants (**Select**, **Over** or **Error**). For the Nottingham dataset the model is trained by using parameters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

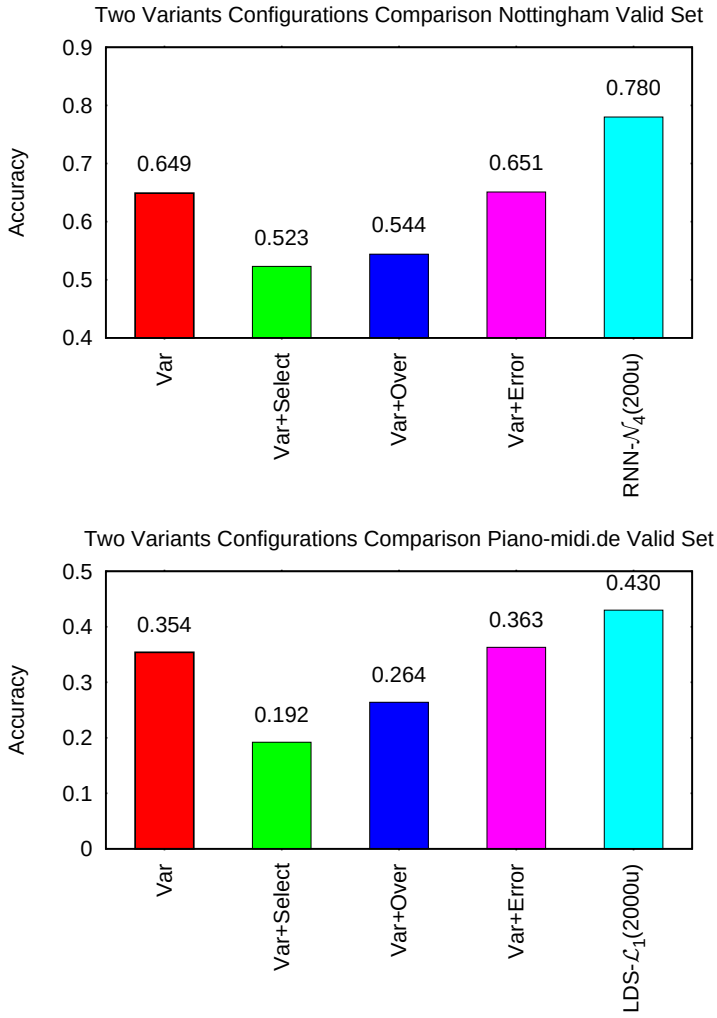

Figure 6.12: Accuracy obtained by LSN model on Nottingham validation set (top) and on Piano-midi.de validation set (bottom) by using different configurations that combine two variants. Each configuration has been tested by using the **Var** configuration in conjunction with a single configuration variants (**Select**, **Over** or **Error**). For the Nottingham dataset the model is trained by using parameters $p = 200$ and $M = 30$. While for the Piano-midi.de dataset the used values are $p = 200$ and $M = 10$.

ing. *ii)* For what concerns the training of the first layer, we can perform the training of each LDS in parallel. Therefore, increasing the number of LDSs does not slow down the computation. The only constraint is the amount of available memory. Indeed, training LDS by method $\mathcal{L}_2$ requires a considerable amount of memory, and training many LDSs at the same time may turn out unfeasible, in particular for large datasets. In our specific case, we had to limit the number of parallel LDSs training processes to 5 on the Piano-midi.de dataset. On the contrary, no bound was needed for the Nottingham dataset.
The other parameter that may create some problems during the training phase, and that has to be tuned, is the size of the state space. As we already explained this parameter determines the size of vector $\mathbf{z}_t$. Moreover, it may affect the quantity of memory used, and the time required to train a single LDS. The higher is the size of the LDS state, the higher will be the amount of memory used during training. It is important to consider that, in case the **Var** variant is used, the state size of each LDS varies on the basis of the rank of the matrix Ξ used in method $\mathcal{L}_2$.
For all the reasons explained above, we decided to test different parameters configurations inside reasonable bounds. In particular, we tested the model with M set to $10, 20$ and 30, and the size of LDS state space p equal to $50, 100, 200$. The model is tested by using two different configurations: the configuration where only the **Var** variant is turned on, and the configuration where both **Var** and **Error** variant are used. In all the performed experiments, the parameters validation is performed by evaluating the various models on the validation set. Figure 6.13 reports the accuracy obtained by the LDS model using the **Var** configuration on the Nottingham and the Piano-midi.de datasets. The best result, as selected by the validation set (Table 6.2), is obtained by using $p = 200$ and $M = 30$ for the Nottingham dataset. For the Piano-midi.de dataset we obtained the best results by using $p = 200$ and $M = 10$. For the Nottingham dataset, it is interesting to notice that for each value of M there is a gain in terms of accuracy by increasing the value of p. That means that the vector $\mathbf{z}_t$ in each case takes advantage in having more information from the LDSs and moreover, each single LDS can "retain" more relevant information about the input. Differently, for the Piano-midi.de dataset, having a larger value for p does not always helps. Indeed, in case of $M = 30$, the model obtains better accuracy by using $p = 50$ instead using $p = 200$. It is possible to observe the same behavior for $M = 20$. This behavior suggests that for large values of p, the vector $\mathbf{z}_t$ has a higher variance, that makes it complex for the matrix $\mathbf{D}$ to map the LSN state on the desired target.
On the other hand, Figure 6.14 shows the time required to train the LSN model, and it is easy to notice that, given a specific value for M using $p = 200$ the training phase requires more or less 3 or 4 times the time required with $p = 50$ (on both datasets). We can notice the same behavior

also with $p = 100$, where the training time doubles with respect to $p = 50$. Similarly, behavior can be observed by fixing a value for p, and varying the value for M. This suggests that the critical part of the training phase is the training of weights matrix D. Figures 6.15 and 6.16 show the results obtained, this time, on the test set by using an LSN with the **Var+Error** configuration. Notice that the trends in terms of time complexity and accuracy are similar to the trends visible in Figures 6.13 and 6.14. The only relevant difference is that, for the Nottingham dataset, the values for accuracy are a bit higher than the ones obtained by the **Var** configuration. The best parameters for the **Var+Error** configuration are the same obtained in the **Var** configuration. For the Nottingham dataset, the time efforts of the two tested configurations are similar. While, for what concerns the Piano-midi.de dataset, the time effort significantly increases by using Var+Error configuration. This behavior and the similarity of the obtained results in terms of accuracy suggest that using two variants together does not lead to any advantage.
The obtained results showed it is important to balance the state space size of the LDSs and the time required for the training phase. Indeed, in both datasets, using $p = 200$ leads to a gain in terms of accuracy. Whilst, using a higher number of LDSs turns not out to be crucial for improving the best results, while it slows down significantly the training of the model. Therefore, we can conclude that a higher values for parameter p lead to larger benefits than using a higher number of LDSs.
We decided not to use method $\mathcal{L}_1$ to train the LDSs because it requires a large state space in order to obtain good state-space representations, making unfeasible in practice to perform the training on the second layer of the LSN model. Table 6.2 reports the results obtained on the training set, test set, and validation set, using the **Var** configuration on the Nottingham and the Piano-midi.de dataset. It is interesting to notice that the LSN model exhibits symptoms which are typical of over-fitting. Indeed, the accuracy obtained on that training set is higher than the results obtained on validation and test set, in particular for higher values of p and M.

6.3.2 Experimental results obtained by SLSN

The SLSN model (Section 4.2) directly derives from the LSN. In practical terms, the main difference is that SLSN does not use the z_t layer. Therefore, instead of having one huge hidden layer, in this case, we have several small hidden states that depend directly on a single LDS state. Moreover, since each LDS depends on the last state computed by the precedent LDS for the SLSN model it is not possible to parallelize the training phase of the various LDSs. This entails that the computational effort for this model, in general, is higher than in a normal LDS. On the other hand, the computational burden is reduced since each LDS is trained only on a specific part of each sequence.

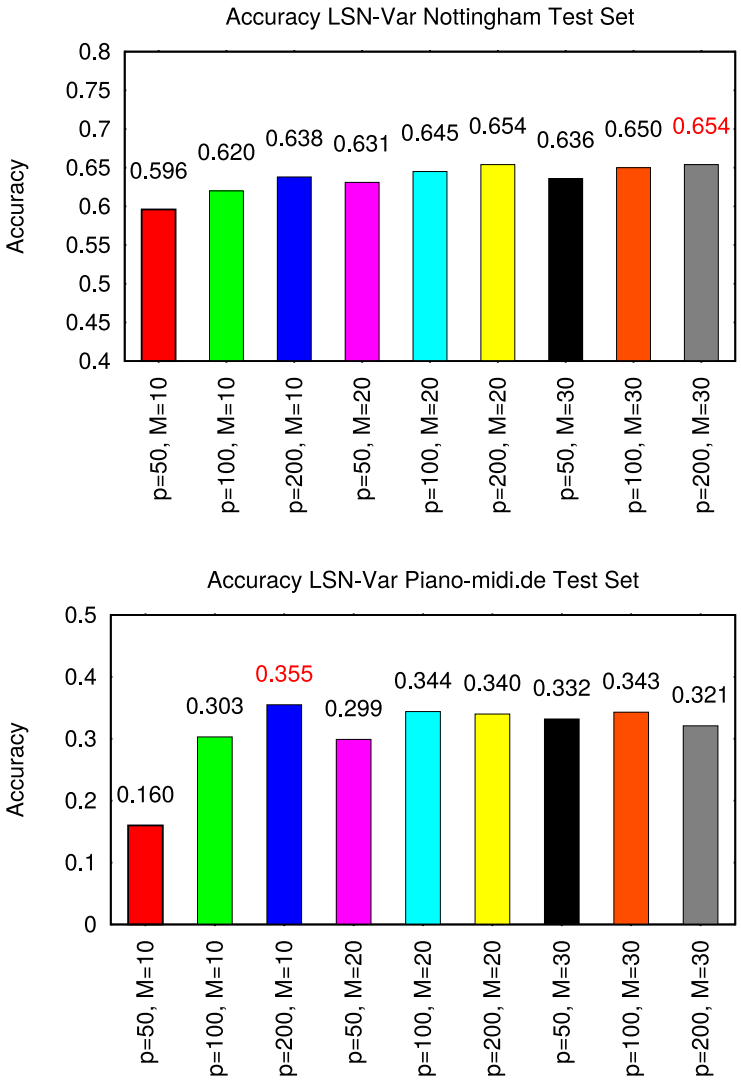

Figure 6.13: The accuracy obtained by the LSN model on the Nottingham dataset (top) and on the Piano-midi.de dataset (bottom) by using different parameters and the **Var** configuration. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20, 30\}$. The value in red identifies the model selected by the validation set.

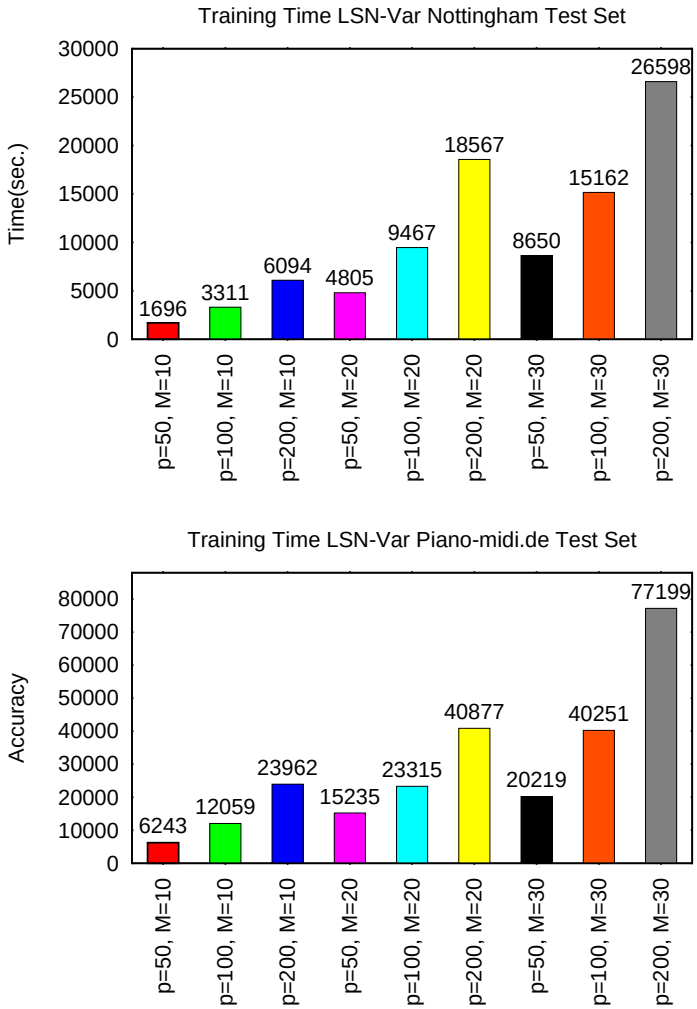

Figure 6.14: The required time to train the LSN model on the Nottingham dataset (top) and the on Piano-midi.de dataset (bottom) by using different parameters and the **Var** configuration. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20, 30\}$.

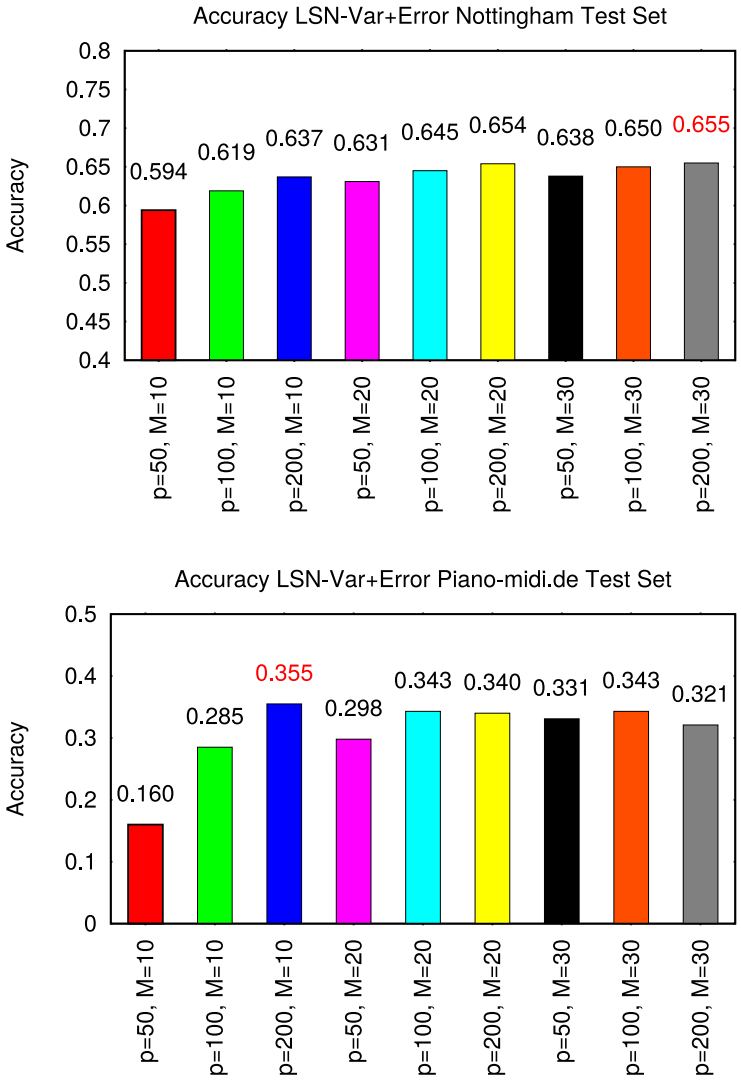

Figure 6.15: The accuracy obtained by the LSN model on the Nottingham dataset (top) and on the Piano-midi.de dataset (bottom) by using different parameters and the **Var+Error** configuration. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20, 30\}$. The value in red identifies the model selected by the validation set.

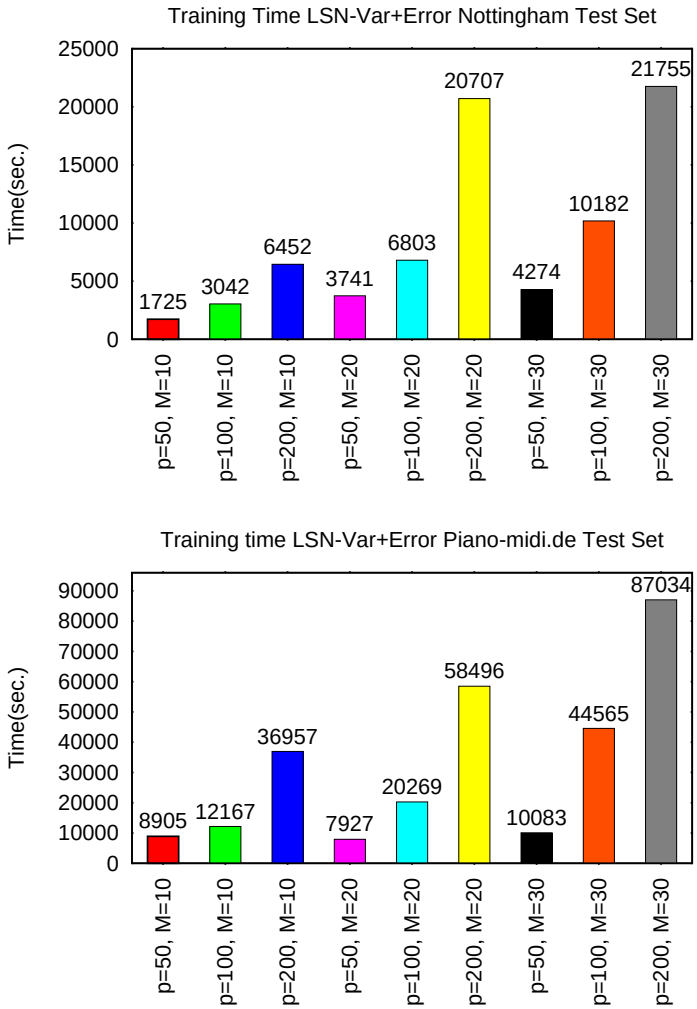

Figure 6.16: The required time to train the LSN model on the Nottingham dataset (top) and on the Piano-midi.de dataset (bottom) by using different parameters and the **Var+Error** configuration. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20, 30\}$.

Parameters	Set	Accuracy on Nott	Accuracy on Piano
$p = 50,\ M = 10$	Training	0.593	0.182
	Test	0.596	0.160
	Validation	0.591	0.190
$p = 100,\ M = 10$	Training	0.619	0.320
	Test	0.620	0.303
	Validation	0.616	0.335
$p = 200,\ M = 10$	Training	0.640	0.385
	Test	0.638	0.355
	Validation	0.634	0.364
$p = 50,\ M = 20$	Training	0.632	0.320
	Test	0.631	0.299
	Validation	0.627	0.328
$p = 100,\ M = 20$	Training	0.653	0.384
	Test	0.645	0.344
	Validation	0.642	0.352
$p = 200,\ M = 20$	Training	0.674	0.421
	Test	0.654	0.340
	Validation	0.649	0.332
$p = 50,\ M = 30$	Training	0.640	0.363
	Test	0.636	0.332
	Validation	0.640	0.346
$p = 100,\ M = 30$	Training	0.662	0.407
	Test	0.650	0.343
	Validation	0.646	0.338
$p = 200,\ M = 30$	Training	0.684	0.452
	Test	0.654	0.321
	Validation	0.650	0.308

Table 6.2: The accuracy obtained by LSN with the **Var** configuration on Nottingham and Piano-midi.de datasets, by using the various set of parameters. The best parameters set for each dataset is highlighted in yellow.

Similarly to what was done for the LNS model, we tested SLSN by using several values for parameters M and p. Specifically, we run the model with $p \in \{10, 20\}$ and $M \in \{50, 100, 200\}$. Figure 6.17 shows the results obtained by the SLSN model on the test set of the Nottingham and Piano-midi.de datasets. Similarly to what we have observed for the LSN model, it can be noted that, larger values for parameter p allow to achieve better results in terms of accuracy. The value of M seems to be less important in order to obtain a gain in the accuracy results. Indeed, this can be explained by the fact that increasing the value of M entails that the sub-training set used to train a single LDS will have shorter sequences; moreover, the dataset of the last LDSs will contain fewer sequences. As a consequence, the contribution of the total error by the last LDSs will be very small.
By considering the best accuracy (selected by validation set) obtained by the SLSN model (that is 0.580 on Nottingham dataset and 0.076 on Piano-midi.de) we can observe that, in particular for the Piano-midi.de dataset, we have obtained very poor results, compared with the results obtained by other models. This behavior may be due to the fact that each LDS is trained (and then applied) on a specific interval of the sequences, and it seems unable to fully exploit the information about the previous time steps passed to it by the precedent LDS. As a matter of fact, the results obtained on Piano-midi.de, where dealing with long terms temporal dependencies is crucial, are very unsatisfactory. In terms of time effort to train the model, the results reported in Figure 6.18 show that the model is really efficient and fast. This good computational performance is obtained thanks to the fact that the size of the sub-dataset used to train each LDS is small, and it contains short sequences. Unfortunately, despite the good performance in terms of time and computational effort, the prediction accuracy achieved by SLSN is very unsatisfactory.

6.3.3 Experimental results obtained by linear co-learning models

The tests performed on linear co-learning models (described in Section 4.3) show results that confirm the theoretical results presented in Section 4.3.1. Indeed, the capability of these models to perform prediction in sequential domains is undermined by the fact that they do not use information coming from the previous state. The performed tests use, as external model, a (Deep) Neural Network (DNN). We decided to use DNN because its high expressiveness ensures the possibility to cope with the complexity of the task. Two different DNN architectures have been tested. The first architecture involves two hidden layers with 500 hidden units each. The adopted activation functions are logistic functions. Training is performed by using the Back-propagation algorithm (500 epochs, learning rate= 0.1, momentum= 0.8). The second used DNN architecture has just one hidden layer of 1000 units.

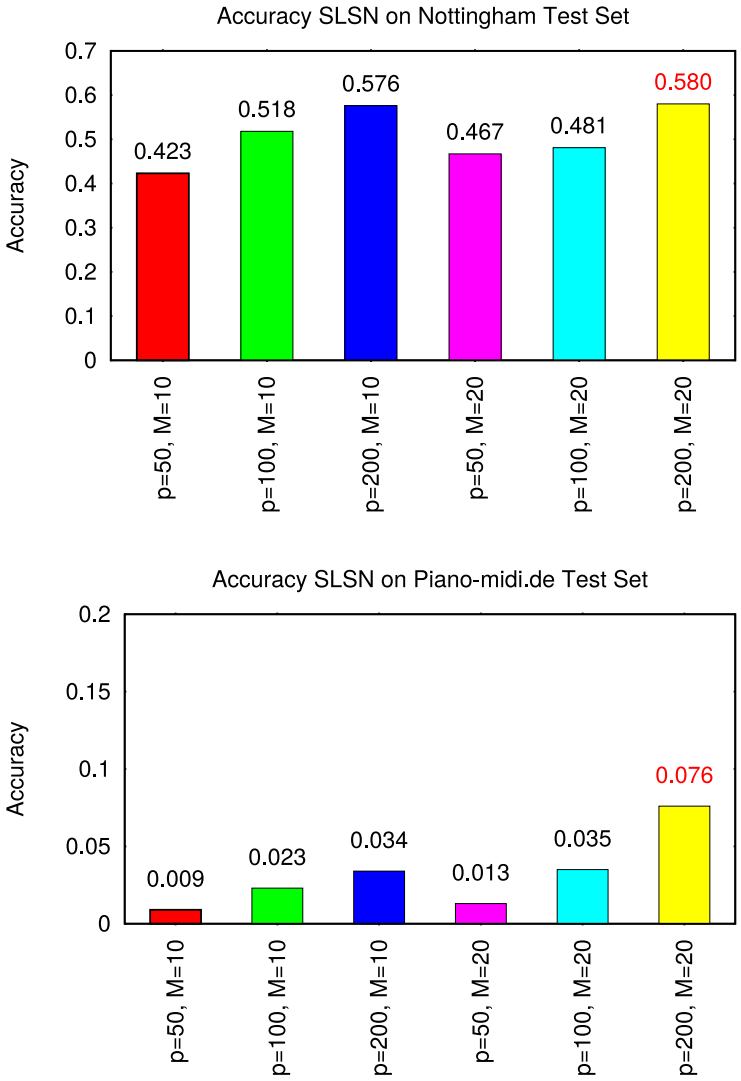

Figure 6.17: Accuracy obtained by the SLSN model on the Nottingham dataset (top) and on the Pinao-midi.de dataset (bottom) by using different parameters. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20\}$. The value in red identifies the model selected by the validation set.

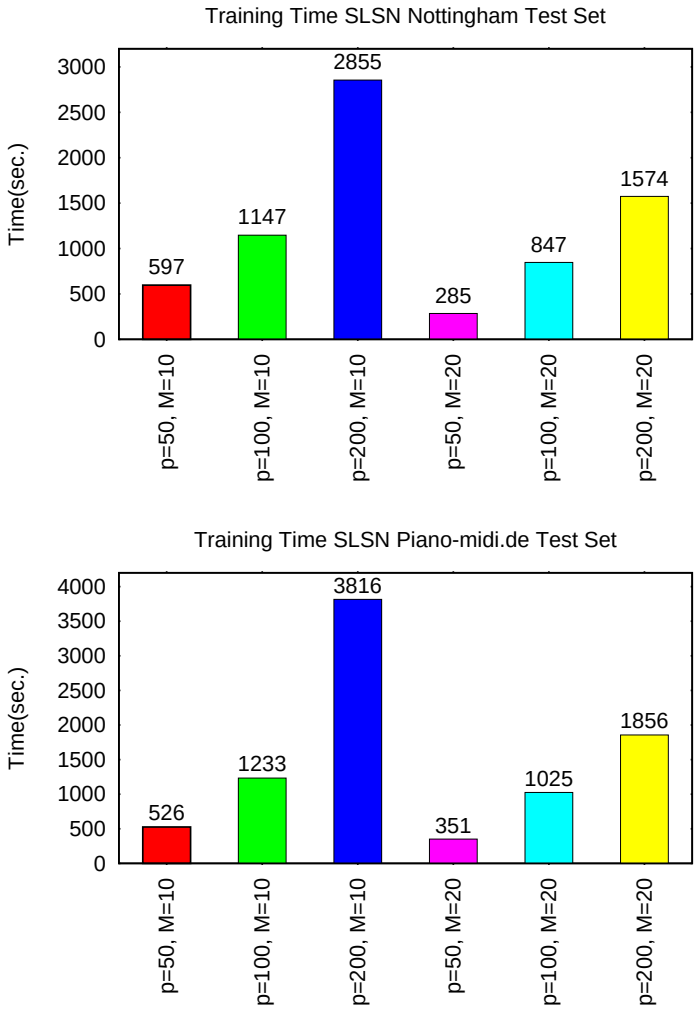

Figure 6.18: Required time to train the SLSN model on the Nottingham dataset (top) and on the Piano-midi.de dataset (bottom) by using different parameters. In particular, $p \in \{50, 100, 200\}$ and $M \in \{10, 20\}$.

Parameters	Set	Acc on Nott (Std Dev)	Acc on Piano (Std Dev)
$p = 200$ $TwoLayerDNN$	Training	0.0564 (±0.0014)	0.0021 (±0.0002)
	Test	0.0558 (±0.0009)	0.019 (±0.0002)
	Validation	0.0537 (±0.0010)	0.0019 (±0.0004)
$p = 200$ $SingleLayerDNN$	Training	0.0428 (±0.0006)	0.0018 (±0.0003)
	Test	0.0422 (±0.0010)	0.0014 (±0.0002)
	Validation	0.0414 (±0.0007)	0.0013 (±0.0004)

Table 6.3: Averege accuracy obtained by 10 runs of the linear co-learning method that exploits a DNN to compute the matrix $\mathbf{V}_t$ on the Nottingham and the Piano-midi.de datasets.

Parameters	Set	Acc on Nott (Std Dev)	Acc on Piano (Std Dev)
$p = 200$ $TwoLayerDNN$	Training	0.0636 (±0.0015)	0.0024 (±0.0002)
	Test	0.0607 (±0.0009)	0.0020 (±0.0004)
	Validation	0.0615 (±0.0010)	0.0025 (±0.0005)
$p = 200$ $SingleLayerDNN$	Training	0.0520 (±0.0014)	0.0021 (±0.0006)
	Test	0.0509 (±0.0009)	0.0019 (±0.0002)
	Validation	0.0514 (±0.0012)	0.0022 (±0.0005)

Table 6.4: Average accuracy obtained by 10 runs of the linear co-learning method that exploits a DNN to compute the hidden state $\mathbf{h}_t$ on the Nottingham and the Piano-midi.de datasets.

The Keras.io framework [23] has been used to implement the DNN architectures.
Table 6.3 shows the obtained results on Nottingham and Piano-midi.de datasets, by using the model that generates matrix $\mathbf{V}_t$. The results obtained by using the external model to compute the hidden state (eq. 4.38) is reported in Table 6.4. In both cases, we obtained, as expected, very poor results. The results in Table 6.4 show a small improvement, but the obtained accuracies are still very low.

The last tested linear co-learning model is the Uninet model (Section 4.3.2). Training this model requires the creation of N LDSs, where N is the number of sequences contained in the dataset. Each LDS is trained, by method $\mathcal{L}_1$, on a training set that contains a single sequence. Therefore, training these LDSs is very fast. Indeed, since each LDS performs learning independently of the others, it is possible to perform parallel training of them.
However, there is a bottleneck in the training of the external model that requires all the LDSs output matrices as targets in the training set. Even in this case, as external model we test two different DNN architectures. The first one uses only one hidden layer with 500 units, while the second one has two hidden layers of 500 units each. For what concerns the used LDSs, they have a state space of size 100.
The training of this model turns out to be very complex because it is trained by using as input the current input of the model and the previous state, and as output a complete matrix $\mathbf{C}$, computed in the first phase of the training by the LDS trained with the current sequence. The task performed by the

Parameters	Set	Acc on Nott (Std Dev)	Acc on Piano (Std Dev)
$p = 100$ $SingleLayerDNN$	Training	0.0181 (±0.0021)	0.0031 (±0.009)
	Test	0.1084 (± 0.0107)	0.0722 (±0.0024)
	Validation	0.1043 (±0.0152)	0.0520 (±0.0032)
$p = 100$ $TwoLayerDNN$	Training	0.016 (±0.0004)	0.0019 (±0.0009)
	Test	0.0512 (±0.0027)	0.0347 (± 0.0016)
	Validation	0.083 (±0.0041)	0.0421 (± 0.0015)

Table 6.5: Accuracy obtained by the Uninet model, that exploits a DNN as external model. The reported results are about the tests performed on the Nottingham and the Piano-midi.de datasets.

Parameters	Set	Avg Acc (Std Dev)
$p = 100$	Nottingham	0.97 (±0.011)
$p = 100$	Piano-midi.de	0.93 (±0.018)

Table 6.6: Average accuracy obtained by using an LDS for each sequence in the training set, and by using each LDS to perform prediction on the same sequence. The table reports the result obtained on the Nottingham and the Piano-midi.de datasets.

DNN is very complex. Table 6.5 reports the obtained results. The results show that Uninet model can not deal with such complex problems. Indeed, accuracy never rises above 8%. This inability in performing the proposed task is due to the high complexity of the problem that the external model has to manage. The accuracy obtained on the training set is always one order of magnitude lower than the accuracy obtained on validation and test sets. We argue that this behavior is due to the fact that the training set contains a significantly higher number of sequences. Since the model gives in output results that are very far from the required target, computing the output of a higher number of elements makes it more likely to have many outputs with an accuracy very close to 0. Finally in Table 6.6 we report the results obtained by the single LDS trained (method $\mathcal{L}_1$) by using as a training set a single sequence. The reported results are the average of the accuracies obtained by each LDS in performing the prediction task on the single sequence used to training it. We perform this test for each sequence in the training set. The results show an accuracy close to 95% in both datasets.

6.3.4 Encode-Decode LDS

The Encode-Decode LDS has been tested on the Nottingham and Piano-midi.de datasets by using different sizes for the state space of the of autoencoder. In each performed test the coding and decoding autoencoders have the same state size. Since the two linear autoencoders that compose the model are trained independently on two different datasets ($\mathcal{T}_{Encoder}$ and $\mathcal{T}_{Decoder}$), the training phases, performed with method $\mathcal{L}_2$, are performed in parallel. Also all states $\mathbf{h}_t^{(Encoder)}$ and $\mathbf{h}_t^{(Encoder)}$ are calculated in parallel.

Parameters	Set	Acc on Nott	Acc on Piano
$p = 250$,	Training	0.5631	0.0056
	Test	0.5592	0.0090
	Validation	0.5626	0.0035
$p = 500$,	Training	0.6038	0.0080
	Test	0.6025	0.0172
	Validation	0.6027	0.0085
$p = 1000$	Training	0.6303	0.0214
	Test	0.6312	0.0403
	Validation	0.6285	0.0235

Table 6.7: Accuracy obtained by the Encode-Decode LDS using different values for p on the Nottingham and the Piano-midi.de datasets.

Once the matrices $\mathbf{H}^{(Encoder)}$ and $H^{(Decoder)}$ are created, the training phase uses the pseudoinverse in order to train the weight matrix $\mathbf{E}$. Therefore, training phase turns out to be very efficient, and thanks to this, it is possible to use a wide state space. The only limitation about the size of state space comes from the amount of available memory. For this reason, and since the model is completely linear, we decided to test it by using different values for p (size of LDSs state space). We set the parameter p equal to 250, 500 and 1000. In Table 6.7 we reported the obtained results. The second column of the table shows the results obtained on the Nottingham dataset. The results are quite close to the results obtained by the LSN and LDS models. It is interesting to notice that LSN obtains an accuracy of 0.655 by using LDSs with $p = 200$, while Encode-Decode LDS requires $p = 1000$ to obtain a result that is even lower. The results show the benefits of using a wider state.
By comparing the obtained results with the results obtained by an LDS (that exploits the same training method $\mathcal{L}_2$, Figure 6.1) we can notice that Encode-Decode LDS is significantly less effective. Indeed, the LDS with $p = 500$ obtains an accuracy of 0.639, while the Encode-Decode LDS obtains an accuracy of 0.603. This fact is even more visible on the Piano-midi.de dataset where the model obtains an accuracy lower than 0.05 regardless of the size state space. This problem, we guess, is due to the difficulty of training the matrix $\mathbf{E}$ used to map (linearly) the projections of the inputs to the projection of the targets. The obtained results suggest that this task is too complex to be performed by using a simple linear operator. Indeed, the good results obtained on Nottingham are related to the low variance between two consecutive time steps of the sequences contained in this dataset. For this reason, the projection of the target (that is the next time step) and the input turn out to be very close and the matrix $\mathbf{E}$ can "map" the connection between them. Otherwise, in Piano-midi.de, the structure of a song is really complex, and in the same song a wide number of different notes and chords are played. This entails that the projections of two consecutive time steps will be very different, and the used liner function is not sufficient to perform

the mapping between the input and the target projections.
The obtained results suggest that the Encode-Decode LDS model is not effective in the polyphonic music prediction task. Indeed, the results show that using a simple LDS allows to achieve better results in terms of accuracy. Furthermore, the Encode-Decode LDS training phase is more complex and requires more memory than the training methods suggested for the LDS model.

6.4 Experiments on Pre-training Methods

In this section we report the results obtained by using the two pre-training techniques presented in Chapter 5 for all the four datasets presented in Section 6.1.2. Firstly, for each approach, we explain how the experiments have been performed and then we discuss the results. For what concerns the HMM-based pre-training approach we tested the method on two different models: RNN and RNN-RBM. The interesting aspect about RNN-RBM is that it is the only one nonlinear sequential model that has an ad-hoc pre-training method that has already been developed and tested. Hence for this model it is possible to make a comparison versus our pre-training method. The test for the linear Autoencoder based pre-training approach is performed only on the RNN model because this second method is developed specifically for RNN models. The code used for RNN and RNN-RBM could is available from [3, 1]. Moreover to develop the HMM model we have used the GHMM library [2]. For what concerns the pre-training method based on linear dynamic autoencoder, the used code is available here [4]. In the final part of this section a comparison of the two pre-training techniques is presented. All the code used to perform the test has been written by using the Theano framework [15] in order to exploit the computational power of GPU computing [66].

6.4.1 HMM-based Pre-Training

In this section, we report the results obtained by applying the HMM-based pre-training approach.

Experimental setup

We tested the generality of the HMM-based pre-training approach by validating it on two of the different types of networks for sequential data described in Chapter 2, i.e., RNNs and RNN-RBMs. Due to the large number of hyperparameters involved in the computation (e.g., the number of hidden variables for both HMMs and recurrent networks, the number, and length of the sequences generated by the HMM, etc.), a systematic exploration of the parameters space was unfeasible. We, therefore, adopted a "probe" approach, where the model-dependent parameters that were already known to give good results for the used datasets where kept fixed, while the remaining parameters were probed for few different values. Specifically, the fixed parameters for both the sequential networks are the number of pre-training and training epochs (which are set to 100 and 200, respectively, for the RNN-RBM and to 2500 and 5000 for the RNN). Moreover, for both the RNN and the RNN-RBM we used a learning rate of 0.001. Only for the RNN-RBM, the number of hidden units was fixed to 150 for the RBM-part, and to 100 for the RNN-part. For the pre-training of both networks, we used an HMM that was trained for $10,000$ iterations using the Baum-Welch algorithm. This setting might not be ideal since a large portion of the parameter space is left unexplored. Nevertheless, the few parameter configurations we have probed were enough to provide evidence that the proposed approach is robust with respect to the learning parameters, since for all the considered datasets we obtained significant improvements, either in terms of accuracy or in terms of computation time.
Following the procedure outlined above, we started our investigation with the RNN-RBM network. Specifically, we explored the effect of using different numbers of states for the HMM, while keeping the number of HMM generated sequences fixed at 500, all of length 200. As mentioned above, since training of RNN-RBMs is very time consuming, we restricted our experiments to the Nottingham and Piano-midi.de datasets. In order to evaluate the experimental results, we made a comparison of our method against the pre-existing pre-training approaches, i.e. SGD and HF, in terms of accuracy (calculated according to the method proposed in [10]) and computation time. For RNNs, we investigated the impact of our pre-training approach on the learning process by varying the number of hidden units and the number (and length) of the sequences generated by the HMM, while keeping the number of hidden states of the HMM fixed to 10 (i.e. the number of hidden states

that constituted to the best trade-off between speed of training and quality of the final result in the RNN-RBMs experiments).

Experimental results

Learning the structure of polyphonic music with HMMs was challenging due to the potentially exponential number of possible configurations of notes that can be produced at each time step, which would cause the alphabet of the model to have an intractable size. We fixed this issue by only considering the configurations that were actually present in each dataset, which reduced the complexity of the alphabet but at the same time maintained enough variability to produce realistic samples. We assessed the accuracy of the models on the prediction task defined in 6.1.1. We also collected the total training times, which are composed of both the pre-training time and the fine-tuning time. For the RNN-RBM, we compared our pre-training method with that used by the authors of the model. Pre-training was performed for 100 epochs, and fine-tuning for 200 epochs. Total training times and prediction accuracies for the HMM, SGD and HF pre-trainings are reported in Figure 6.19 for the Nottingham dataset, and in Figure 6.20 for the Piano-midi.de dataset. In both figures, information about pre-training time is reported in the figure legend, after each curve label. Specifically, each label is followed by a couple of values in parenthesis that represent, for our pre-training approach, the time required for training the HMM and the time required for the pre-training phase, while for the other pre-training approaches the values represent the time required for pre-training the RBM-part and the RNN-part of the network by using HF or SGD.

In general, different pre-training methods led to similar accuracies (both for training and test sets) at the end of the fine-tuning phase. However, in the more complex Piano-midi.de dataset our pre-training obtained slightly better results. Regarding convergence speed, the HMM method always significantly outperformed the others (e.g., it saved more than 8 hours of computing in the Nottingham dataset). We also assessed the change in performance as the number of HMM states varies. As expected, using a smaller number of hidden states (≤ 25) reduced pre-training times. Interestingly, this did not affect the quality of the models after the fine-tuning phase, which still converged to good solutions. Using an HMM with 50 states, instead, was detrimental due to the slow convergence speed of the HMM training. Thus, the HMM pre-training seems to perform a better initialization of the network, which allowed to improve convergence speed also during the fine-tuning phase. For example, the network pre-trained with the HMM reached the highest accuracy after only 110/120 epochs, compared to 200 epochs required by the other methods. It is worth noting that the accuracies measured directly on the HMMs were always fairly low, at best approaching 53% in the Nottingham dataset and 10.1% in the Piano-midi.de dataset.

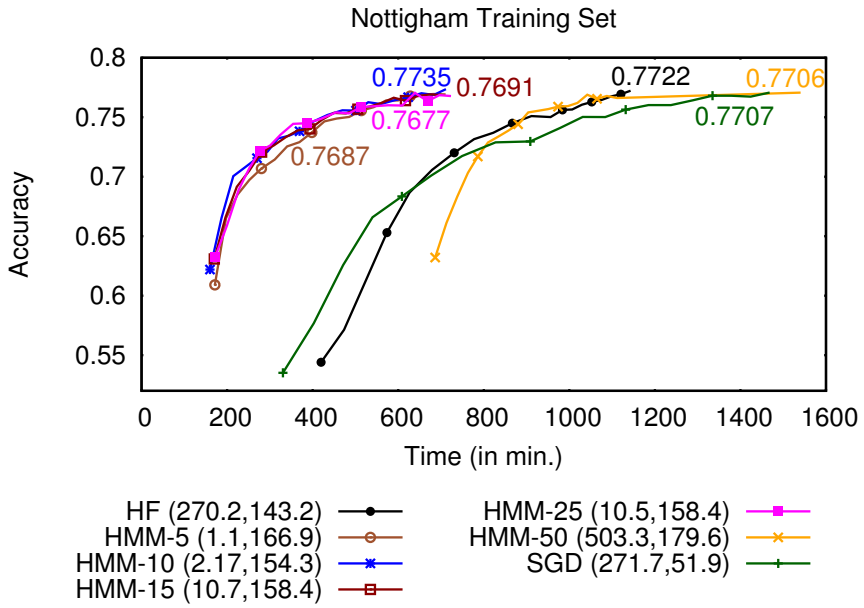

Figure 6.19: Accuracy and running times of the tested pre-training methods, measured on the Nottingham dataset. Each curve is identified by a label followed by a couple of execution times in parenthesis: the pattern `HMM-n (time1,time2)` refers to our approach, where `n` is the number of hidden states used for the HMM, $\texttt{time}_1$ is the training time for the HMM, $\texttt{time}_2$ is the pre-training time; with the label `HF` (or `SGD`) we represent the Hessian Free (or Stochastic Gradient Descent) pre-training performed in $\texttt{time}_1$ time for the RBM-part, and $\texttt{time}_2$ time for the RNN-part. The final test set performance for each method is reported at the end of each corresponding curve.

Concerning the experiments involving RNNs, we recall that a single HMM with 10 hidden states for each dataset was used to generate all the pre-training sequences of the corresponding dataset. Three different network architectures were used containing, respectively, 50, 100, and 200 hidden units. Pre-training for the network with 50 hidden units was performed by using both 25 generated sequences of length 25 and 500 generated sequences of length 200. For the architecture with 100 hidden units, three different pre-training configurations were used: 250 generated sequences of length 200, 500 generated sequences of length 50, 500 generated sequences of length 50. Finally, for the architecture with 200 hidden units, 500 generated sequences of length 200 were used.

In Figures 6.21-6.24 we compare, for all the four datasets, the learning curves obtained for the training and test sets by our pre-training method versus the learning curves obtained without pre-training. The starting point of the curves involving pre-training takes into consideration the pre-training

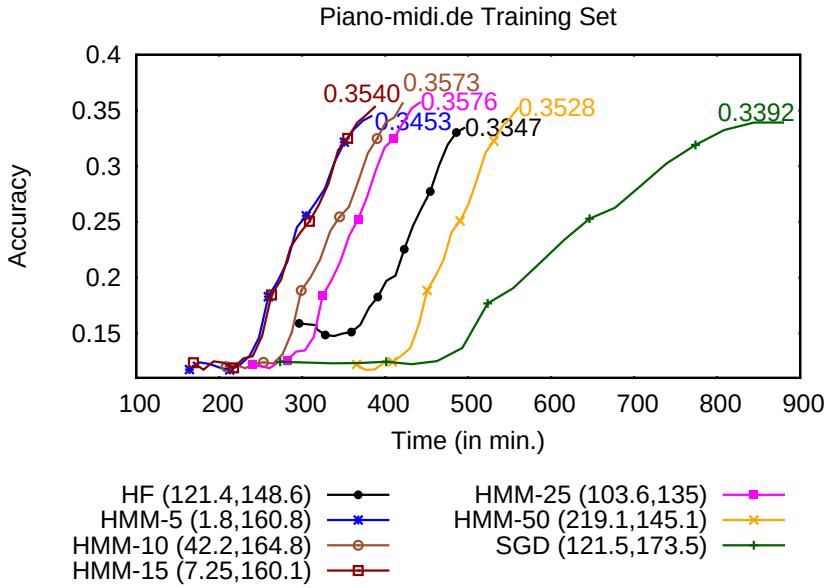

Figure 6.20: Accuracy and running times of the tested pre-training methods, measured on the Piano-midi.de dataset. Each curve is identified by a label followed by a couple of execution times in parenthesis: the pattern `HMM-n` `(time`$_1$`,time`$_2$`)` refers to our approach, where `n` is the number of hidden states used for the HMM, `time`$_1$ is the training time for the HMM, `time`$_2$ is the pre-training time; with the label `HF` (or `SGD`) we represent the Hessian Free (or Stochastic Gradient Descent) pre-training performed in `time`$_1$ time for the RBM-part, and `time`$_2$ time for the RNN-part. The final test set performance for each method is reported at the end of each corresponding curve.

time. Since the number of training epochs for the RNNs is fixed to 5000 and it does not depend on the presence of pre-training, in the plots we highlighted, via a vertical dotted line, the point in time where the slowest RNN without pre-training finished training. From the plots, it can be observed that some runs of RNNs using the same number of hidden units have a significant difference in execution time. We believe this is mainly due to the Theano dynamic C code generation feature, which can, under favorable conditions, speed up computation in a significant way.

From the learning curves, we can notice that the performance on the test sequences is very similar to the behavior on the training sequences (i.e., the models did not overfit). Concerning the effectiveness of pre-training, it is clear that using just 50 hidden units does not lead to any benefit, while pre-training turns to be quite effective for networks with 100 and 200 hidden units, allowing the RNN to reach very good generalization performances. Moreover, using many short generated sequences seems to be the best choice

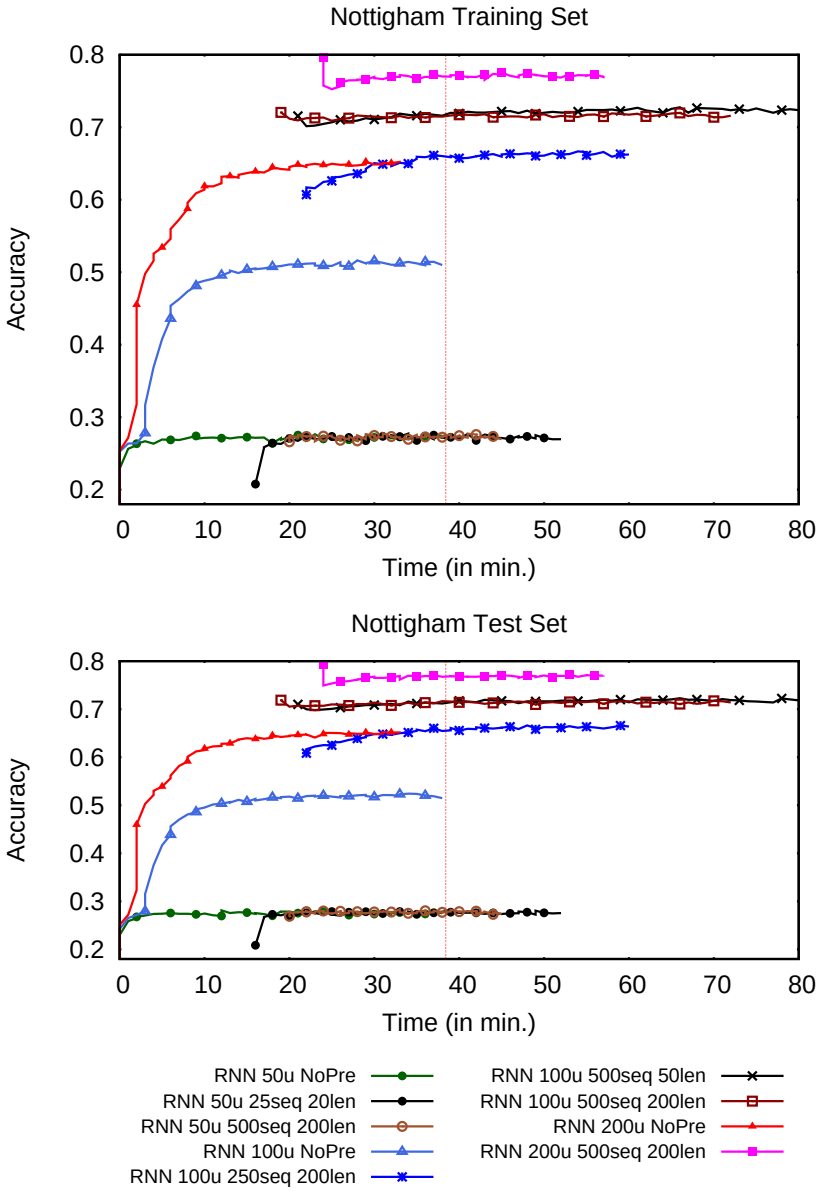

Figure 6.21: Training (top) and test (bottom) accuracy and running times for RNNs on the Nottingham dataset. Each curve is identified by the label `RNN` followed by three or two identifiers: the three identifiers pattern $\texttt{n}_1\texttt{U}\ \texttt{n}_2\ \texttt{n3}$ refers to our approach, where $\texttt{n}_1$ is the number of used hidden units for RNN, $\texttt{n}_2$ is the number of sequences generated by an HMM with 10 states, and $\texttt{n}_3$ is the length of such sequences; the two identifiers pattern `nU NoPre` refers to a RNN with standard random initialization and `n` hidden units. A dotted vertical line is used to mark the end of training of RNNs, with no pre-training, after 5000 epochs. The same number of epochs is used to train RNNs with pre-training.

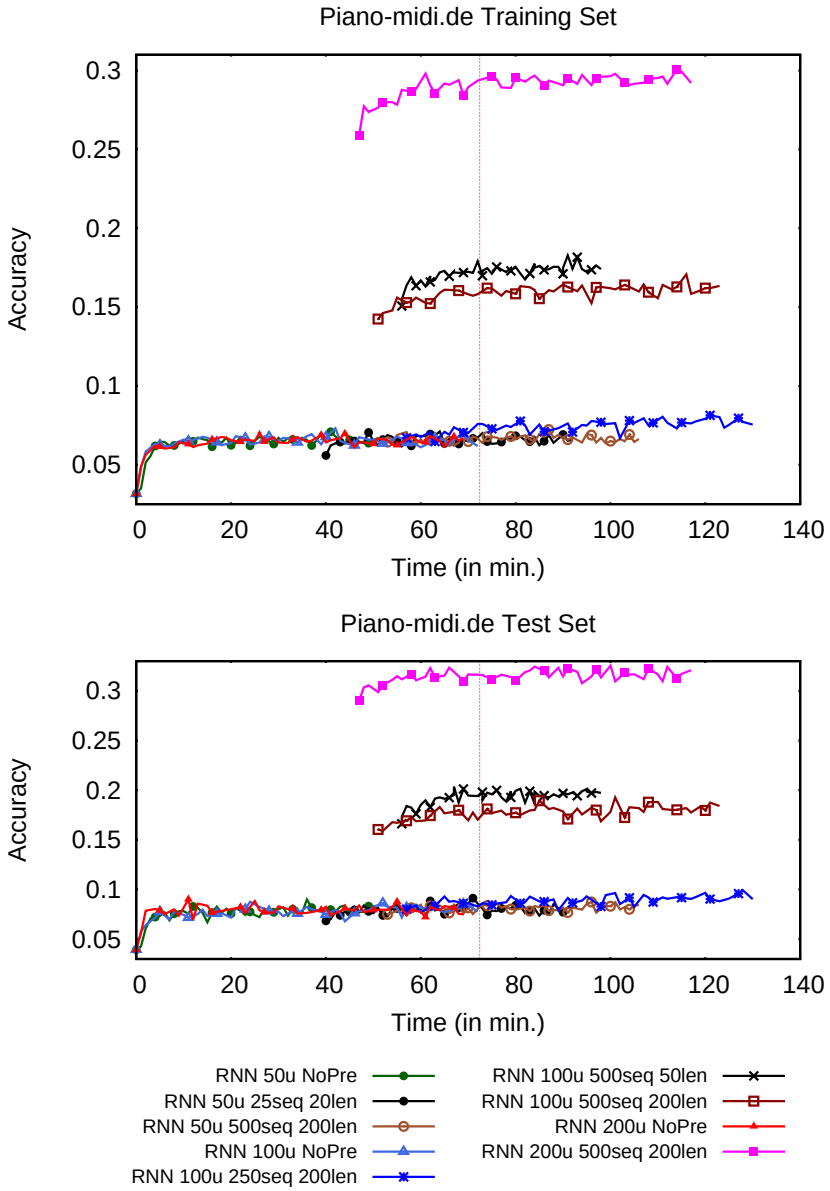

Figure 6.22: Training (top) and test (bottom) accuracy and running times for RNNs on the Piano-midi.de dataset. Each curve is identified by the label `RNN` followed by three or two identifiers: the three identifiers pattern $\mathtt{n}_1\mathtt{U}\ \mathtt{n}_2\ \mathtt{n3}$ refers to our approach, where $\mathtt{n}_1$ is the number of used hidden units for RNN, $\mathtt{n}_2$ is the number of sequences generated by an HMM with 10 states, and $\mathtt{n}_3$ is the length of such sequences; the two identifiers pattern `nU NoPre` refers to a RNN with standard random initialization and `n` hidden units. A dotted vertical line is used to mark the end of training of RNNs, with no pre-training, after 5000 epochs. The same number of epochs is used to train RNNs with pre-training.

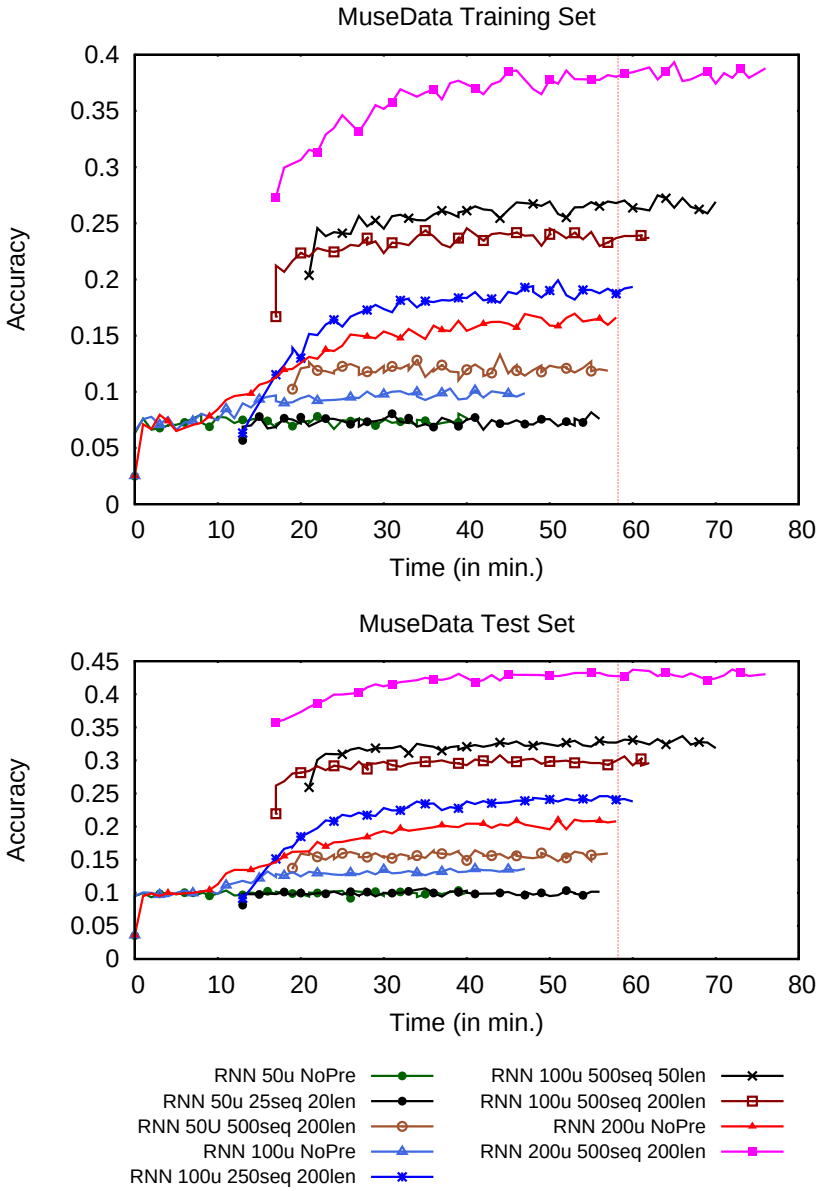

Figure 6.23: Training (top) and test (bottom) accuracy and running times for RNNs on the Muse dataset. Each curve is identified by the label `RNN` followed by three or two identifiers: the three identifiers pattern $\texttt{n}_1\texttt{U}\ \texttt{n}_2\ \texttt{n3}$ refers to our approach, where $\texttt{n}_1$ is the number of used hidden units for RNN, $\texttt{n}_2$ is the number of sequences generated by an HMM with 10 states, and $\texttt{n}_3$ is the length of such sequences; the two identifiers pattern `nU NoPre` refers to a RNN with standard random initialization and `n` hidden units. A dotted vertical line is used to mark the end of training of RNNs, with no pre-training, after 5000 epochs. The same number of epochs is used to train RNNs with pre-training.

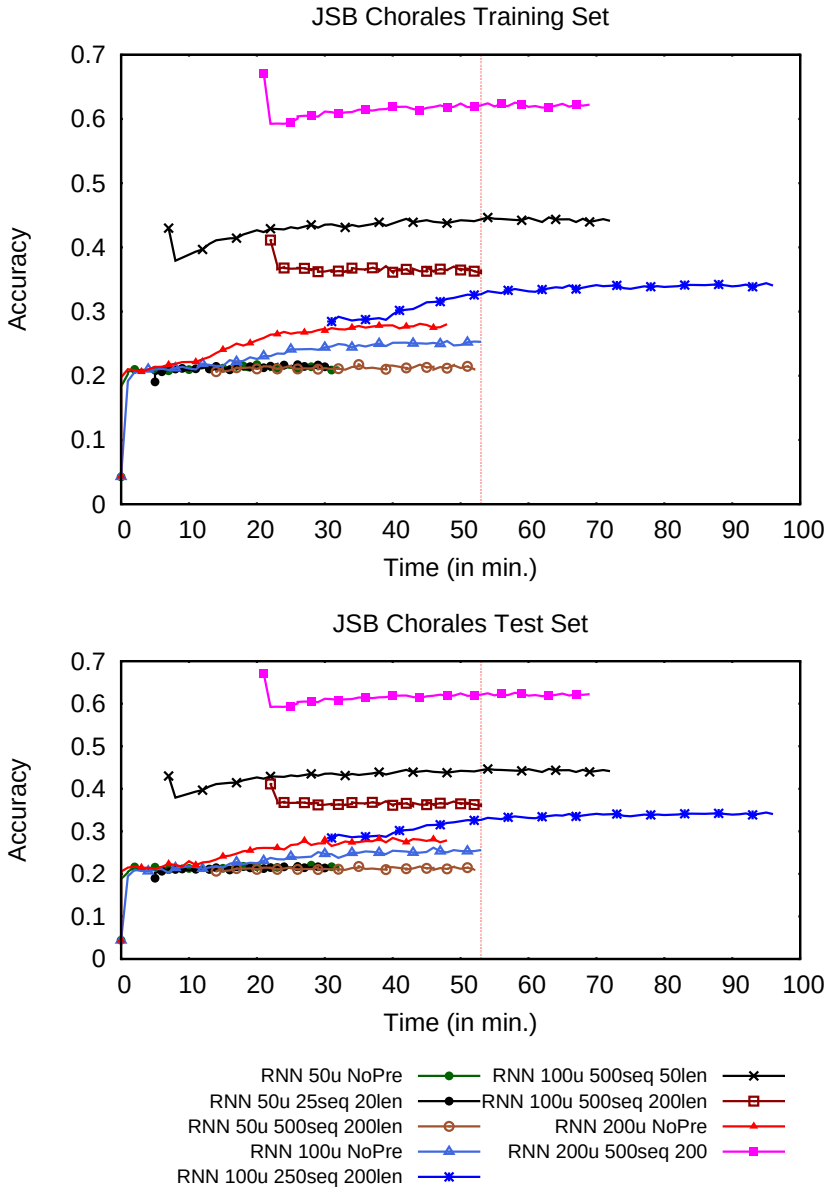

Figure 6.24: Training (top) and test (bottom) accuracy and running times for RNNs on the JSB dataset. Each curve is identified by the label `RNN` followed by three or two identifiers: the three identifiers pattern $\mathtt{n}_1\mathtt{U}\ \mathtt{n}_2\ \mathtt{n3}$ refers to our approach, where $\mathtt{n}_1$ is the number of used hidden units for RNN, $\mathtt{n}_2$ is the number of sequences generated by an HMM with 10 states, and $\mathtt{n}_3$ is the length of such sequences; the two identifiers pattern `nU NoPre` refers to a RNN with standard random initialization and `n` hidden units. A dotted vertical line is used to mark the end of training of RNNs, with no pre-training, after 5000 epochs. The same number of epochs is used to train RNNs with pre-training.

Dataset	Model	ACC%
Nottingham	GMM + HMM	59.27
	RNN (w. HF)	62.93 (66.64)
	RNN-RBM	75.40
	HMM-PreT-RNN (200U 500 200)	**80.47**
Piano-midi.de	GMM + HMM	7.91
	RNN (w. HF)	19.33 (23.34)
	RNN-RBM	28.92
	HMM-PreT-RNN (200U 500 200)	**36.51**
MuseData	GMM + HMM	13.93
	RNN (w. HF)	23.25 (30.49)
	RNN-RBM	34.02
	HMM-PreT-RNN (200U 500 200)	**44.96**
JSB Chorales	GMM + HMM	19.24
	RNN (w. HF)	28.46 (29.41)
	RNN-RBM	33.12
	HMM-PreT-RNN (200U 500 200)	**67.36**

Table 6.8: Accuracy results for state-of-the-art models [17] vs our pre-training approach. The acronym GMM + HMM is used to identify Hidden Markov Models (HMM) using Gaussian Mixture Models (GMM) indices as their state. The acronym (w. HF) is used to identify an RNN trained by Hessian Free optimization.

for all datasets, as clearly demonstrated by the networks using 100 hidden units and 3 different configurations for the generated sequences. For the Nottingham and JSB databases, pre-training seems to reach a very good starting point that is subsequently *lost* by training with the original dataset. This is an interesting behavior that needs to be more carefully investigated in future studies. At the time marked by the vertical line (i.e. when the slowest RNN without pre-training finished training), all the curves associated to RNNs adopting pre-training reached a performance that was very close to the final one. This suggests that our pre-training can reach significantly better solutions using the same amount of time used by RNNs with no pre-training.

Finally, it is interesting to compare the test performances reached by the RNNs with the results obtained in [17]. In Table 6.8 we have reported, for each dataset, the test performances of their Hidden Markov Models (HMM) using Gaussian Mixture Models (GMM) indices as their state, their RNN (also with HF training) and their RNN-RBM networks, jointly with the test performances reached by our RNNs with pre-training (HMM-PreT-RNN) as selected by using the associated validation set. For all datasets we obtain significantly better results, especially when considering the HMM-based approach. The improvement is particularly large for the JSB dataset.

It can be also observed that our pre-training on the RNN-RBM architecture did not seem to improve on test performances. However, we recall that we used a much smaller number of hidden units with respect to the network used in [17], which may explain why no significantly better results

are obtained for the Nottingham and Piano-midi.de datasets, although there was a significant improvement in training times.

Parameters setting

Even if our experimental investigation confirms the appeal of the proposed pre-training strategy, the optimal choice of the parameters involved in our method is still partially unexplored. In particular, in our experiments, we fixed many learning parameters (e.g., learning rates and number of learning epochs) and only coarsely explored the best values for the other parameters. In this final section, we briefly make some considerations that could help to better understand what is the impact of some settings that directly affect our pre-training method, in particular, the length of the sequences generated by the HMM.

As shown in Figures 6.21-6.24, the best final performance is clearly achieved when using more hidden units in the RNN (i.e., 200 units instead of 100). However, the results do not clearly characterize how the number and the length of the generated sequences affect the final RNN performance. Specifically, it seems that by sampling more sequences we usually obtain better results (compare, for example, the blue - 250 - and the red - 500 - lines in all the above-mentioned figures). A possible reason for this phenomenon is that by sampling more sequences we add more variability to the pre-training sequences, which results in a better generalization. At the same time, we face a trade-off because adding more sequences to the smooth dataset also causes an increase in pre-training times.

The parameter representing the length of the sequences sampled from the HMM, instead, appears to be more subtle to optimize. In particular, it seems that we do not need to generate very long sequences from the HMM, because good performances can already be obtained by using a length of only 50 (see black, crossed lines in the above-mentioned figures). This result might be due to the statistics of the considered datasets, because most of the structure to be learned could be encoded in short sequences. The statistics reported in Table 6.1 only report maximum, minimum and mean values for the sequences contained in the four datasets, but such measures do not fully characterize the distribution of sequences length in the different sets. Indeed, as shown in Figure 6.25 which reports the datasets cumulative distributions of sequence length, it appears that in both JSB and Nottingham datasets most of the sequences are fairly short.

The experimental results seem to suggest that a good choice for the length of the HMM generated sequences is to choose a value that covers about the 50% of the cumulative distribution over the lengths in the training set. However, the trade-off between the length of the sequences and the required time for pre-training must also be taken into account. Indeed, with longer sequences, the computational burden of pre-training in

terms of time increases significantly. Besides, using long HMM generated sequences does not guarantee that the accuracy of the pre-trained network is going to improve. In fact, from the experimental results, it can be observed that networks pre-trained with 500 sequences of length 50 (black line with crosses) get a better accuracy with respect to networks pre-trained with 500 sequences of length 200 (brown line with square).

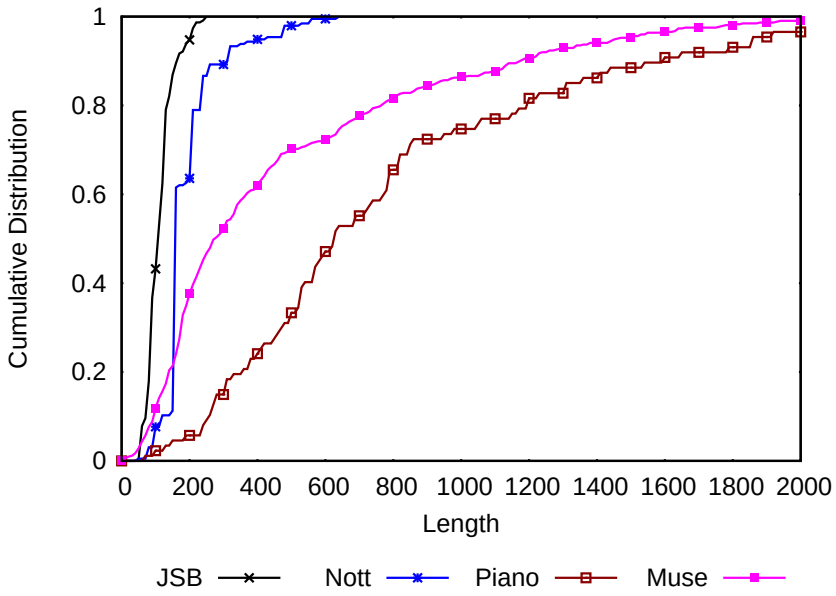

Figure 6.25: Cumulative probability distribution for the lengths of sequences contained in the four considered datasets. Please, note that, in order to have a clear separation among the four curves, only lengths up to 2,000 are considered in the plot.

A possible explanation for the worse performance obtained when generating long sequences might be found in the short memory of HMMs, which would cause longer sequences to "drift away" from the correct structure as the generation proceeds. We tested this hypothesis by plotting in Figure 6.26 the HMM[1] accuracy versus the length of sequences belonging to the Muse training and test datasets. The Muse datasets were chosen because they contain sequences with complex structures as well as a sufficient number of sequences to get reliable statistics. The plot is actually reporting the average accuracy of bins of size 50 over the length. The standard deviation for each bin is reported as well in the plot.

The plots do not seem to show the "drift away" effect described above, although it must be recognized that we do not know the memory size needed to cover all (or most) of the long-term dependencies which actually occur

[1]We recall that the HMM is using 10 states.

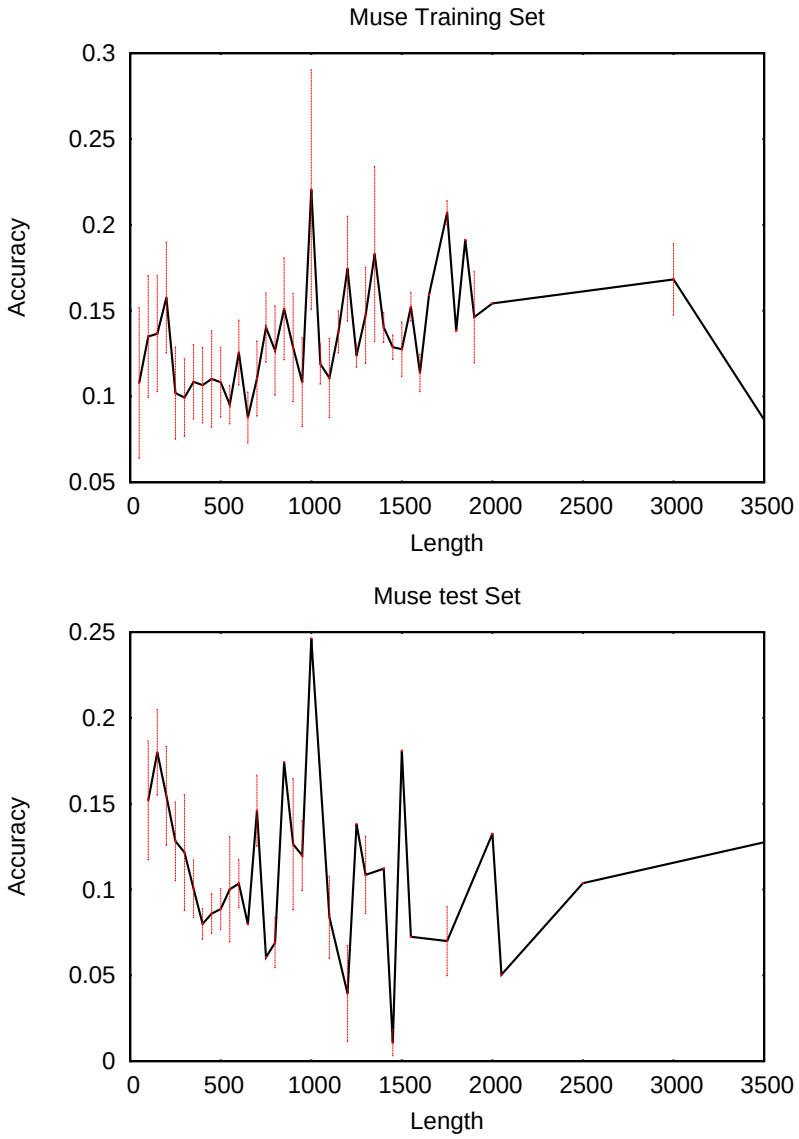

Figure 6.26: Average accuracy of HMM on the sequences contained on Muse training and test sets, grouped in bins of size 50 over the length. The standard deviation for each bin is reported as well.

into the Muse datasets, so it is difficult to evaluate how many of the long-term dependencies have been identified by the HMM. However, the fact that many long sequences have better accuracy values than the average (0.1265 for the training set, and 0.129 for the test set) seems to be a good indication that there is not a memory issue with HMM.

6.4.2 Autoencoder-based Pre-training

Experimental setup

Our Autoencoder-based pre-training approach (LA-PreT-RNN) has been assessed by using a different number of hidden units (i.e., p is set in turn to 50, 100, 150, 200, 250) and 5000 epochs of RNN training in all datasets except Muse. Indeed, due to early overfitting, for this dataset, we used 1000 epochs. The test has been executed by using a standard vanilla RNN [3] developed in Theano. The RNN is trained by using SGD on error function $E_{\mathcal{T}}$ (eq. (3.1)). Random initialization (Rnd) has also been used for networks with the same number of hidden units in order to test the effectiveness of this pre-training approach. Specifically, for networks with 50 hidden units, we have evaluated the performance of six different random initialization. Finally, in order to verify that the nonlinearity introduced by the RNN is actually useful to solve the prediction task, we have also evaluated the performance of a network with linear units (250 hidden units) initialized with our pre-training procedure (LA-PreT-Lin250).

To give an idea of the time performance of pre-training with respect to the training of a RNN, in the second column of Table 6.9 we have reported the time in seconds needed to compute pre-training matrices (Pre-) (on Intel© Xeon© CPU E5-2670 @2.60GHz with 128 GB) and to perform training of a RNN with $p = 50$ for 5000 epochs (on GPU NVidia K20). Please, note that for larger values of p, the increase in computation time of pre-training is smaller than the increment in computation time needed for training an RNN.

Experimental results

Training and test curves for all the models described in the Section 5.3 are reported in Figures 6.27, 6.28, 6.29, 6.30. In all datasets is evident that random initialisation does not allow the RNN to improve its performance in a reasonable amount of epochs. Specifically, for random initialisation with $p = 50$ (Rnd 50), we have reported the average and range of variation over the six different trails: different initial points do not change substantially the performance of RNN. Increasing the number of hidden units allows the RNN to slightly increase its performance. Using pre-training, on the other hand, allows the RNN to start training from a quite favorable point, as demonstrated by an early sharp improvement of performances. Moreover,

Dataset	(Pre-)Training Time	Model	ACC% [17]
Nottingham	seconds (226) 5837 $p = 50$ 5000 epochs	RNN (w. HF) RNN-RBM LA-PreT-RNN LA-PreT-Lin250	62.93 (66.64) **75.40** 75.23 ($p = 250$) 73.19
Piano-midi.de	seconds (2971) 4147 $p = 50$ 5000 epochs	RNN (w. HF) RNN-RBM LA-PreT-RNN LA-PreT-Lin250	19.33 (23.34) 28.92 **37.74** ($p = 250$) 16.87
MuseData	seconds (7338) 4190 $p = 50$ 5000 epochs	RNN (w. HF) RNN-RBM LA-PreT-RNN LA-PreT-Lin250	23.25 (30.49) 34.02 **57.57** ($p = 200$) 3.56
JSB Chorales	seconds (79) 6411 $p = 50$ 5000 epochs	RNN (w. HF) RNN-RBM LA-PreT-RNN LA-PreT-Lin250	28.46 (29.41) 33.12 **65.67** ($p = 250$) 38.32

Table 6.9: Pre-training statistics including computational times in seconds to perform pre-training and training for 5000 epochs with $p = 50$ (column 3), and accuracy results for state-of-the-art models [17] vs our pre-training approach (columns 3-4). The acronym (w. HF) is used to identify an RNN trained by Hessian-Free Optimization [62].

the more hidden units are used, the more the improvement in performance is obtained, till overfitting is observed. In particular, early overfitting occurs for the Muse dataset. It can be noticed that the linear model (Linear) reaches performances which are in some cases better than RNN without pre-training. However, it is important to notice that while it achieves good results on the training set (e.g. JSB and Piano-midi), the corresponding performance on the test set is poor, showing a clear evidence of overfitting. Finally, in column 4 of Table 6.9, we have reported the accuracy obtained after validation on the number of hidden units and number of epochs for our approaches (PreT-RNN and PreT-Lin250) versus the results reported in [17] for RNN (also using Hessian-Free Optimization) and RNN-RBM. In any case, the use of pre-training largely improves the performances over standard RNN (with or without Hessian-Free Optimization). Moreover, with the exception of the Nottingham dataset, the proposed approach outperforms the state-of-the-art results achieved by RNN-RBM. Large improvements are observed for the Muse and JSB datasets. Performance for the Nottingham dataset is basically equivalent to the one obtained by RNN-RBM. For this dataset, also the linear model with pre-training achieves quite good results, which seems to suggest that the prediction task for this dataset is much easier than for the other datasets. The linear model outperforms RNN without pre-training on Nottingham and JSB datasets, but shows problems with the Muse dataset.

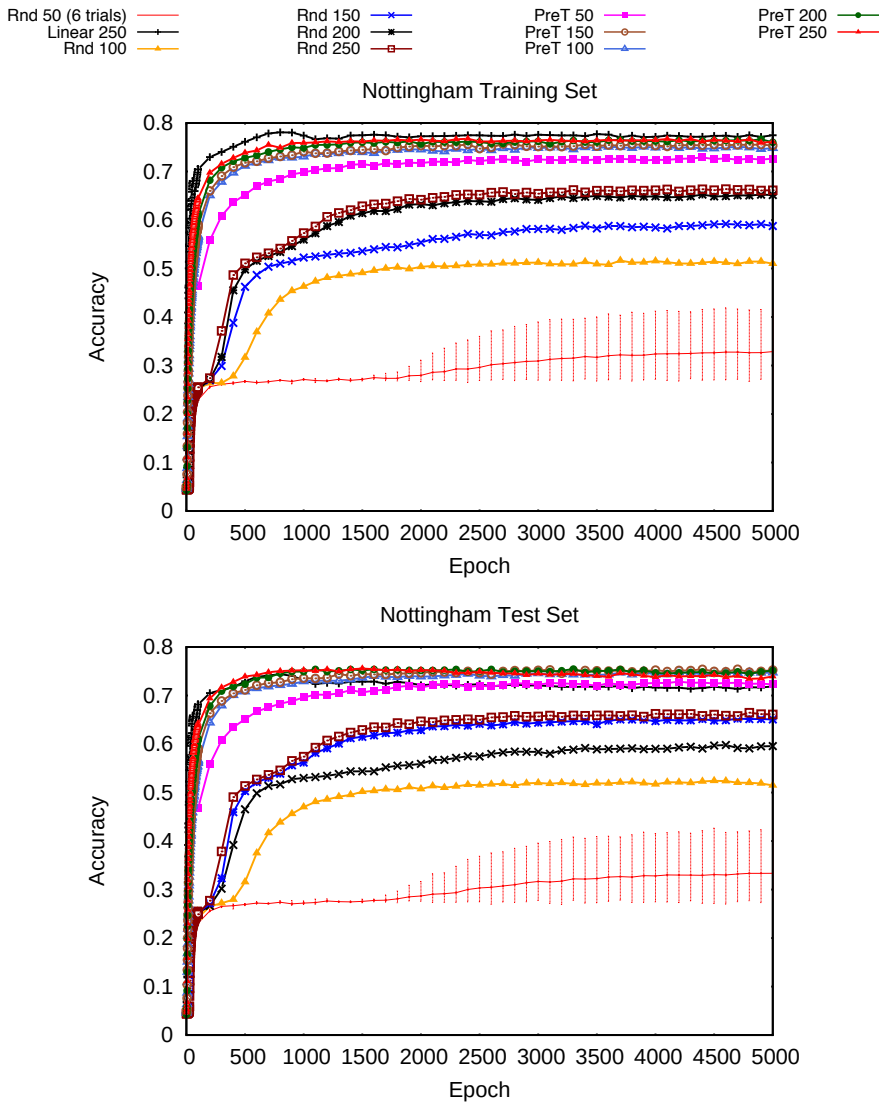

Figure 6.27: Training (top) and test (bottom) curves for the assessed approaches on the Nottingham dataset. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards.

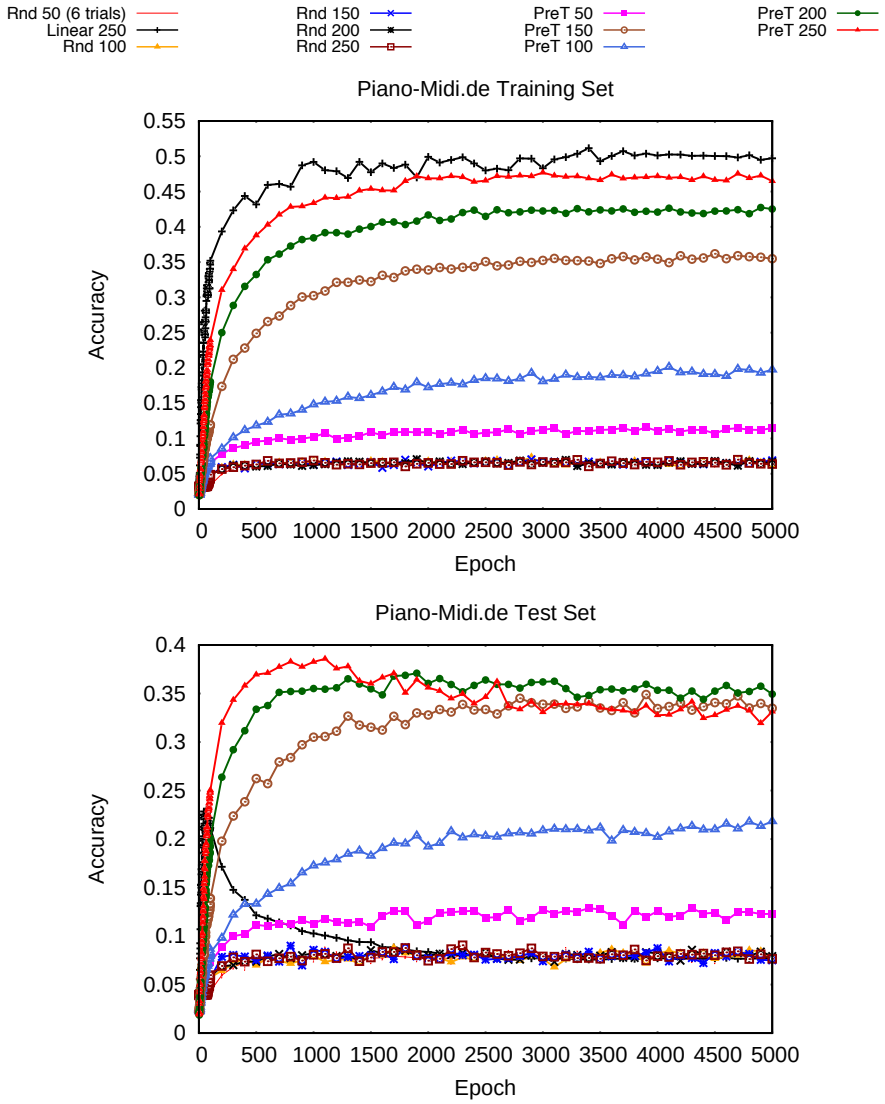

Figure 6.28: Training (top) and test (bottom) curves for the assessed approaches on the Piano-midi.de dataset. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards.

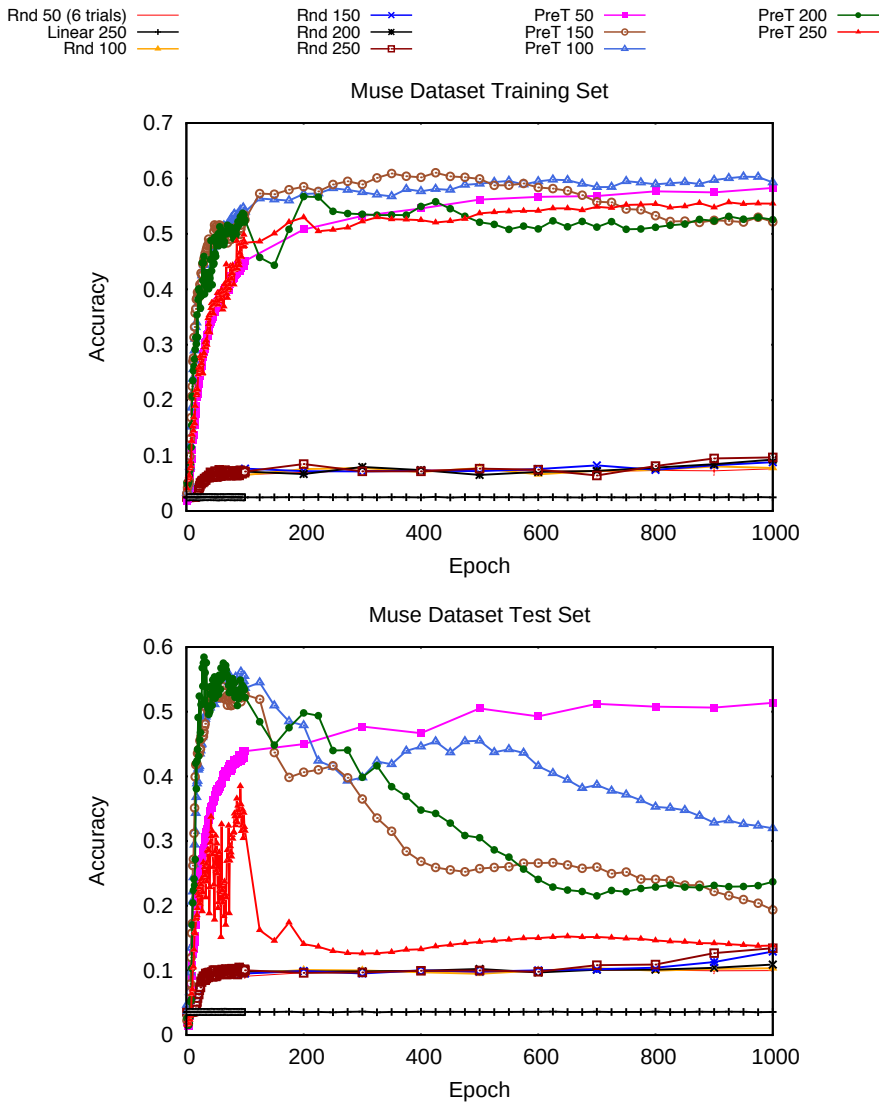

Figure 6.29: Training (top) and test (bottom) curves for the assessed approaches on the Muse dataset. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards.

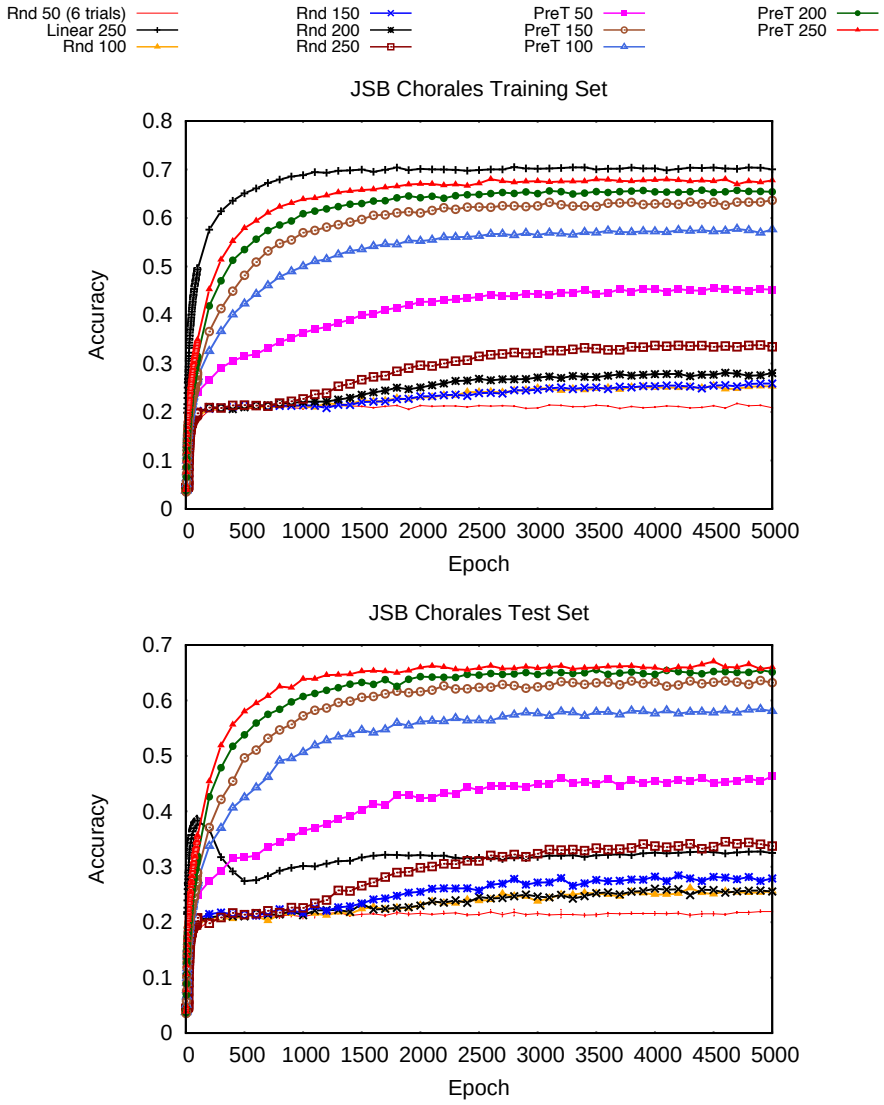

Figure 6.30: Training (top) and test (botom) curves for the assessed approaches on the JSB dataset. Curves are sampled at each epoch till epoch 100, and at steps of 100 epochs afterwards.

Chapter 7

Conclusions

In this thesis, we have addressed an issue that is often disregarded when considering the application of machine learning in sequential domains: the usefulness of linear models. We studied whether a linear model can be used in order to cope with typical complex tasks usually treated by common nonlinear models, or at least whether it is possible to use them in order to simplify and speed up the training phase of complex nonlinear models, e.g. RNN, RNN-RBM. To explore the strengths and the weaknesses of linear models we firstly considered one of the simplest of them: the Linear Dynamical System (LDS). We proposed three methods that allow to train this model on large datasets, which are typical of Machine Learning tasks. The first is inspired by Echo State Networks. It exploits the random initialization of the matrices of the LDS, and performs a simple fast phase of supervised training on the output matrix. The second consists in adapting the back-propagation algorithm, widely used to train RNN models, to the linear dynamic system. The last one exploits the capability of a Linear Dynamic Autoencoder to compute a rich internal feature representation of the input data. The idea is to use the matrices that represent a solution for an autoencoder, as input-to-state and recurrent matrices for the LDS, and performing a fast supervised training phase only for the output matrix. This method turns out to be very interesting inasmuch we proposed a new fast and efficient method to obtain a closed form solution for the definition of the "optimal" weights for linear autoencoder, which however, entails the computation of the SVD decomposition of the full data matrix. For large data matrices exact SVD decomposition cannot be achieved, so we proposed a computationally efficient procedure to get an approximation that turned out to be effective for our goals.

The second step that we have done in order to explore the potential of applying linear models to sequential domains was studying whether it is possible to derive a more powerful model by using LDSs as building blocks. We proposed several different architectures. The first one is called Linear System Network and it consists in a model that exploits the state representation computed by several LDSs fed with the same input, in order to compute a nonlinear representation of it. The main idea is to merge the various state representations in a smart way. This model presents many advantages, in particular for what concerns the training procedure. Indeed, it is possible to train each LDS that compose the architecture independently, by using one of the methods proposed in Chapter 4. The LSN uses the obtained nonlinear representation of the input to compute the model output by exploiting a simple linear function.
Another architecture that we developed is an extension of LSN, that is called Sequential-LSN. This method exploits several linear systems in order to use each of them to perform prediction/classification tasks only on a small part of the input sequence. Moreover, each LDS computes its own state that is used to compute the output of the considered part of the sequence. We saw that all proposed methods are limited due to the fact that, in order to obtain an LDS that does not diverge during the prediction task, the model has to set the recurrent matrix in a way that the system results contractive. That means that the LDS will have problems in dealing with long-term temporal dependencies. For this reason, we developed some models that make use of co-learning techniques, that involve the use of an external model in order to compute (directly or indirectly) the projection in state space of the input. We proved that the idea of using co-learning techniques to substitute the recurrent part of the model becomes ineffective when the models use a fixed output matrix. The idea is to compute the "optimal" state representation, given the target by using pseudoinverse of the output matrix. We need to compute the state in this way because we have to generate a target to train the external model. The result of our theoretical studies is that the model under these conditions becomes stateless. Finally, we studied if it is possible to use linear models to pre-train common nonlinear models e.g RNN. We proposed two novel methods to perform the pre-training phase. The first one is called HMM-based pre-training and consists in generating a smoothed, approximated dataset using a first-order HMM trained on the original data. This smoothed data are then used to pre-train a nonlinear model. It should be stressed that the proposed method does not need any *ad-hoc* adjustments of existing learning algorithms because it consists in generating a simplified dataset that is used to initialize the parameters of the learner. Our pre-training strategy is therefore very general, and its benefits could be readily extended to pre-train many other types of models for sequences.
The second proposed pre-training method is called Pre-training via Linear Autoencoder. This pre-training technique is strictly applicable to RNN. As

the name suggests, this pre-training method is based on linear autoencoder for sequences. The idea is to initialize the RNN weights matrices by using the ones obtained for a linear autoencoder. To compute the linear autoencoder matrices we used the same method proposed to LDS training. After the matrices initialization, the RNN has to be fine tuned by using a gradient base approach.

We have tested all these models and methods on a polyphonic music prediction task, that involves in predicting the notes and the cords that will be played in the next time step given the notes that were played during the previous time steps.

For what concerns the LDS we have empirically studied the performances of three training approaches and we have compared them with the results of non-linear approaches for sequence learning that have obtained very good results on the proposed task. Experimental results seem to show that linear dynamical systems may play an important role. In fact, when used directly they may by themselves return state-of-the-art performance (see for example LDS-$\mathcal{L}_3$ for the Nottingham dataset) while requiring a much lower computational effort with respect to their non-linear counterpart.

The experimental results obtained by testing LSN model show interesting results, but lower than the state-of-the-art. Moreover, the computational cost of training LSN is higher than training a single LDS. The main problem is that LSN model uses a huge projection of the input that makes the training phase very complex. The size of the obtained nonlinear projection is correlated with the number of LDSs used to compute it, and with the size of the state space of each LDS. The problem is that, in order to maintain the computational complexity under a reasonable bound, we have to limit the size of the LDS state space. But, as the results on LDS have shown, using a large state space is crucial to obtain good results. The tests on Co-learning models confirm the results of our theoretical study that reveals the limits of that type of models.

For what concerns the proposed pre-training approaches, our HMM-based pre-training applied on RNN-RBM model [17] leads to prediction accuracies significantly higher than those obtained by currently available pre-training strategies but requiring a significantly lower computational time. We also tested the method on a classic recurrent neural network, and also, in this case, the effectiveness of our approach was confirmed, obtaining a large improvement in all datasets. Although we tried to explore the joint parameters space which includes both the pre-training parameters as well as the recurrent neural network parameters, further research is needed to better understand how to reach the optimal setting for the number and length of sequences generated by the HMM. The empirical evidence we have collected in our experiments seems to suggest that using too long sequences is detrimental for the fine-tuning phase. It seems much better to use many short sequences. This may be understood as a way to avoid overfitting by

the pre-training phase: the main mode of the data can be captured by short sequences, while long-term dependencies are left to the fine-tuning phase. As a consequence of that, the empirical evidence seems to suggest that it is not important to use a Hidden Markov Model with many hidden states. In fact, having many hidden states makes pre-training too slow and prone to introduce overfitting. The results obtained by using Pre-training via Linear Autoencoder applied on the same prediction task show the usefulness of the proposed pre-training approach, since it allows to largely improve the state of the art results on all the considered datasets by using simple stochastic gradient descent for learning. Even if the results are very encouraging the method needs to be assessed on data from other application domains.

Therefore we can conclude that the linear model can obtain good results in perform learning in sequential domains. Moreover, the results obtained in this thesis showed that, by using LDSs, it is possible to obtain results that are comparable, and sometimes better, than the state-of-the-art. In addition, we prove that linear models allow to achieve very good results by limiting the computational burden required for training the model. Indeed, linear models show that it is possible to obtain good results in significantly less time, and by using fewer resources than the most common nonlinear methods. Furthermore, when the obtained results are not enough accurate, the computed configuration with the linear model can be used to help a more complex nonlinear model to compute better solutions. Therefore, we propose to use firstly a linear model also to solve a complex task. Indeed, if it is not enough to provide a reasonable solution, it is possible to resort to it to design quite effective pre-training techniques for more complex nonlinear models.

As future work, we plan to develop models that exploit the minimum degree of nonlinearity needed to obtain reasonable solutions. The idea is to use several "layers" that have an increasing degree of nonlinearity. This allows to control, and minimize the computational burden of the training phase, and to optimize the trade-off between performance and accuracy.
Moreover, we plan to extend the proposed methods and techniques, to other types of structured data. In particular, we are interested in developing and studying an extension of our works for graph-structured data.

Bibliography

[1] http://deeplearning.net/tutorial/code/rnnrbm.py.

[2] http://ghmm.sourceforge.net/.

[3] https://github.com/gwtaylor/theano-rnn.

[4] http://www.math.unipd.it/ lpasa/.

[5] G E Dahl A. Mohamed and G E Hinton. Acoustic Modeling using Deep Belief Networks. *IEEE Trans. on Audio, Speech, and Language Processing, 20, pp 14-22*, 2012.

[6] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[7] Fabio Aiolli, Giovanni Da San Martino, Markus Hagenbuchner, and Alessandro Sperduti. Learning Nonsparse Kernels by Self-Organizing Maps for Structured Data. {*IEEE*} *Transactions on Neural Networks*, 20(12):1938–1949, 2009.

[8] Kevin Ashton. That internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009.

[9] P Baldi and K Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, 2(1):53–58, 1989.

[10] Mert Bay, A F Ehmann, and J S Downie. Evaluation of Multiple-F0 Estimation and Tracking Systems. *ISMIR*, pages 315–320, 2009.

[11] Y Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[12] Y Bengio, P Simard, and P Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, January 1994.

[13] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[14] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[15] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[16] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[17] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012.

[18] H Bourlard and Y Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4-5):291–294, 1988.

[19] Jérémie Cabessa and Hava T Siegelmann. The Computational Power of Interactive Recurrent Neural Networks. *Neural Computation*, 24(4):996–1019, 2012.

[20] Yves Chauvin and David E Rumelhart, editors. *Backpropagation: Theory, Architectures, and Applications*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[21] Huanhuan Chen, Fengzhen Tang, Peter Tino, Anthony G Cohn, and Xin Yao. Model metric co-learning for time series classification. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 3387–3394. AAAI Press, 2015.

[22] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase repre-

sentations using rnn encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[23] François Chollet. Keras: Theano-based deep learning library. *Code: https://github. com/fchollet. Documentation: http://keras. io*, 2015.

[24] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[25] Nadia A Chuzhanova, Antonia J Jones, and Steve Margetts. Feature selection for genetic sequence classification. *Bioinformatics*, 14(2):139–143, 1998.

[26] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[27] 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.

[28] Corinna Cortes, Patrick Haffner, and Mehryar Mohri. Rational Kernels: Theory and Algorithms. *Journal of Machine Learning Research*, 5:1035–1062, 2004.

[29] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995.

[30] T Cover and P Hart. Nearest Neighbor Pattern Classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, sep 2006.

[31] P di Lena, K Nagata, and P Baldi. Deep architectures for protein contact map prediction. *Bioinformatics*, 28(19):2449–2457, 2012.

[32] Jeffrey L Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[33] D Erhan, Y Bengio, and A Courville. Why does unsupervised pretraining help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

[34] Paolo Frasconi, Marco Gori, Andreas Kuechler, and Alessandro Sperduti. From Sequences to Data Structures: Theory and Applications. In J Kolen and S Kremer, editors, *A Field Guide to Dynamic Recurrent Networks*, pages 351–374. 2001.

[35] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, Vol 9(5):768–785, 1998.

[36] Claudio Gallicchio and Alessio Micheli. Deep reservoir computing: A critical analysis. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2016.

[37] Zhe Gan, Chunyuan Li, Ricardo Henao, David E Carlson, and Lawrence Carin. Deep temporal sigmoid belief networks for sequence modeling. In *Advances in Neural Information Processing Systems*, pages 2467–2475, 2015.

[38] Walter R Gilks. *Markov chain monte carlo*. Wiley Online Library, 2005.

[39] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[40] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[41] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. DRAW: A recurrent neural network for image generation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1462–1471, 2015.

[42] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[43] Peter E Hart. The condensed nearest neighbor rule (Corresp.). {*IEEE*} *Transactions on Information Theory*, 14(3):515–516, 1968.

[44] G E Hinton, S Osindero, and Y W Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, 2006.

[45] G E Hinton and R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[46] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[47] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[48] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[49] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

[50] T S Jaakkola and D Haussler. Exploiting generative models in discriminative classifiers. *Advances in neural information processing systems*, pages 487–493, 1999.

[51] Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

[52] Herbert Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.

[53] I T Jolliffe. *Principal Component Analysis*. Springer-Verlag New York, Inc., 2002.

[54] Rafal Józefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2342–2350, 2015.

[55] Ross Kindermann and J. L. Snell. *Markov Random Fields and Their Applications*. AMS, 1980.

[56] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. {MIT} Press, 2009.

[57] 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.

[58] Christina S Leslie and Rui Kuang. Fast String Kernels using Inexact Matching for Protein Sequences. *Journal of Machine Learning Research*, 5:1435–1455, 2004.

[59] Lennart Ljung. System identification. In *Signal Analysis and Prediction*, pages 163–173. Springer, 1998.

[60] Mantas Lukoševičius. A practical guide to applying echo state networks. In *Neural networks: Tricks of the trade*, pages 659–686. Springer, 2012.

[61] Mantas LukošEvičIus and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[62] J Martens and I Sutskever. Learning Recurrent Neural Networks with Hessian-Free Optimization. In *International Conference on Machine Learning*, pages 1033–1040. Acm, 2011.

[63] G Martinsson and Others. Randomized methods for computing the Singular Value Decomposition (SVD) of very large matrices. In *Works. on Alg. for Modern Mass. Data Sets, Palo Alto*, 2010.

[64] Andrés Marzal and Enrique Vidal. Computation of Normalized Edit Distance and Applications. {*IEEE*} *Trans. Pattern Anal. Mach. Intell.*, 15(9):926–932, 1993.

[65] A Micheli and A Sperduti. Recursive Principal Component Analysis of Graphs. In *ICANN (2)*, pages 826–835, 2007.

[66] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[67] Luca Pasa and Alessandro Sperduti. Pre-training of recurrent neural networks via linear autoencoders. In *Advances in Neural Information Processing Systems*, pages 3572–3580, 2014.

[68] Luca Pasa and Alessandro Sperduti. Learning sequential data with the help of linear systems. In *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*, pages 3–17. Springer International Publishing, 2016.

[69] Luca Pasa, Alberto Testolin, and Alessandro Sperduti. An HMM-based Pre-training Approach for Sequential Data. pages 1–6, 2014.

[70] Luca Pasa, Alberto Testolin, and Alessandro Sperduti. Neural Networks for Sequential Data: a Pretraining Approach based on Hidden Markov Models. *Neurocomputing*, 2015.

[71] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *CoRR*, abs/1312.6026, 2013.

[72] J Ross Quinlan. {*C4.5:*} *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[73] Eran Rabani and Sivan Toledo. Out-of-Core SVD and QR Decompositions. In *PPSC*, 2001.

[74] L R Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[75] R Raina, A Madhavan, and A Y Ng. Large-scale deep unsupervised learning using graphics processors. *International Conference on Machine Learning*, pages 110–880, 2009.

[76] Eric Sven Ristad and Peter N Yianilos. Learning String-Edit Distance. {*IEEE*} *Trans. Pattern Anal. Mach. Intell.*, 20(5):522–532, 1998.

[77] Luis Javier Rodriguez and Inés Torres. Comparative study of the baum-welch and viterbi training algorithms applied to read and spontaneous speech recognition. In *Pattern Recognition and Image Analysis*, pages 847–857. Springer, 2003.

[78] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[79] Paul Smolensky. Parallel distributed processing: explorations in the microstructure of cognition, vol. 1. chapter information processing in dynamical systems: foundations of harmony theory. *MIT Press, Cambridge, MA, USA*, 15:18, 1986.

[80] A Sperduti. Exact Solutions for Recursive Principal Components Analysis of Sequences and Trees. In *ICANN (1)*, pages 349–356, 2006.

[81] A Sperduti. Efficient Computation of Recursive Principal Component Analysis for Structured Input. In *ECML*, pages 335–346, 2007.

[82] A Sperduti. Linear Autoencoder Networks for Structured Data. In *NeSy'13:Ninth International Workshop onNeural-Symbolic Learning and Reasoning*, 2013.

[83] Ron Sun and C Lee Giles, editors. *Sequence Learning - Paradigms, Algorithms, and Applications*. Springer-Verlag, London, UK, UK, 2001.

[84] I Sutskever, J Martens, G E Dahl, and G E Hinton. On the importance of initialization and momentum in deep learning. In *ICML (3)*, pages 1139–1147, 2013.

[85] Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. The recurrent temporal restricted boltzmann machine. In *Advances in Neural Information Processing Systems*, pages 1601–1608, 2008.

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

[87] Graham W Taylor and Geoffrey E Hinton. Factored conditional restricted Boltzmann machines for modeling motion style. In *Proceedings*

of the 26th annual international conference on machine learning, pages 1025–1032. ACM, 2009.

[88] Laurens van der Maaten. Learning Discriminative Fisher Kernels. In *Proceedings of the 28th International Conference on Machine Learning, {ICML} 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 217–224, 2011.

[89] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[90] T Voegtlin. Recursive principal components analysis. *Neural Networks*, 18(8):1051–1063, 2005.

[91] Li Wei and Eamonn J Keogh. Semi-supervised time series classification. In *Proceedings of the Twelfth {ACM} {SIGKDD} International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 748–753, 2006.

[92] Lloyd R Welch. Hidden Markov models and the Baum-Welch algorithm. *IEEE Information Theory Society Newsletter*, 53(4):10–13, 2003.

[93] P J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 70(10):1550–1560, 1990.

[94] Xiaopeng Xi, Eamonn J Keogh, Christian R Shelton, Li Wei, and Chotirat Ann Ratanamahatana. Fast time series classification using numerosity reduction. In *Machine Learning, Proceedings of the Twenty-Third International Conference {(ICML} 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, pages 1033–1040, 2006.

[95] Oksana Yakhnenko, Adrian Silvescu, and Vasant Honavar. Discriminatively Trained Markov Model for Sequence Classification. In *Proceedings of the 5th {IEEE} International Conference on Data Mining {(ICDM} 2005), 27-30 November 2005, Houston, Texas, {USA}*, pages 498–505, 2005.

[96] Lexiang Ye and Eamonn J Keogh. Time series shapelets: a new primitive for data mining. In *Proceedings of the 15th {ACM} {SIGKDD} International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, pages 947–956, 2009.

[97] Izzet B Yildiz, Herbert Jaeger, and Stefan J Kiebel. Re-visiting the echo state property. *Neural networks*, 35:1–9, 2012.

[98] Dong Yu, Li Deng, and Shizhen Wang. Learning in the deep-structured conditional random fields. In *Proc. NIPS Workshop*, pages 1–8, 2009.

[99] Z Zhang and H Zha. Structure and Perturbation Analysis of Truncated SVDs for Column-Partitioned Matrices. *SIAM J. on Mat. Anal. and Appl.*, 22(4):1245–1262, 2001.