

Antônio Horta Ribeiro

**Learning nonlinear differentiable models for signals and systems:
with applications**

Brazil

2020

Antônio Horta Ribeiro

**Learning nonlinear differentiable models for signals and systems:
with applications**

Thesis presented to the Universidade Federal de Minas Gerais' Postgraduate Program in Electrical Engineering as requirement for obtaining the Doctoral degree in Electrical Engineering.

Universidade Federal de Minas Gerais – UFMG

Faculdade de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica – PPGEE

Supervisor: Luis Antonio Aguirre

Co-supervisor: Thomas B. Schön

Brazil

2020

Antônio Horta Ribeiro

**Aprendendo modelos não-lineares diferenciáveis para sinais e sistemas:
com aplicações**

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais como requisito para a obtenção do grau de Doutor em Engenharia Elétrica.

Universidade Federal de Minas Gerais – UFMG

Faculdade de Engenharia

Programa de Pós-Graduação em Engenharia Elétrica – PPGEE

Orientador: Luis Antonio Aguirre

Co-orientador: Thomas B. Schön

Brazil

2020

R484I

Ribeiro, Antônio Horta.

Learning nonlinear differentiable models for signals and systems
[recurso eletrônico] : with applications / Antônio Horta Ribeiro. - 2020.
1 recurso online (171 f. : il., color.) : pdf.

Orientador: Luis Antonio Aguirre.

Coorientador: Thomas B. Schön.

Tese (doutorado) - Universidade Federal de Minas Gerais,
Escola de Engenharia.

Anexos: f. 147-153.

Bibliografia: f. 154-171.

Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia elétrica - Teses. 2. Aprendizado do computador - Teses.
3. Aprendizado profundo - Teses. 4. Identificação de sistemas - Teses.
5. Sistemas não lineares - Teses. I. Aguirre, Luis Antônio. II. Schön,
Thomas B. III. Universidade Federal de Minas Gerais. Escola de
Engenharia. IV. Título.

CDU: 621.3(043)

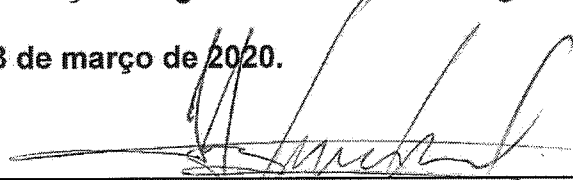
"Learning Nonlinear Differentiable Models For Signals And Systems: With Applications"

Antônio Horta Ribeiro

Tese de Doutorado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Doutor em Engenharia Elétrica.

Aprovada em 03 de março de 2020.

Por:



Prof. Dr. Luis Antonio Aguirre
DELT (UFMG) - Orientador


Prof. Dr. Thomas Schon
Department of Information Technology (Uppsala University) -
Coorientador




Prof. Dr. Eduardo Mazoni Andrade Marçal Mendes
DELT (UFMG)



Prof. Dr. Frederico Gadelha Guimarães
DEE (UFMG)



Prof. Dr. Guilherme de Alencar Barreto
Departamento de Engenharia de Teleinformática (UFC)



Prof. Dr. Leandro dos Santos Coelho
Programa de Pós-Graduação em Engenharia Elétrica no Grupo de
Pesquisa de Operação (UEPR)



Prof. Dr. Maarten Schoukens
Department of Electrical Engineering (Eindhoven Univ Technol,
Eindhoven, Netherlands)

Acknowledgements

I have learned a great deal during my time as a Ph.D. student, and I would like to acknowledge the support and encouragement I have received.

To Luis Aguirre, who has been with me since my time as an undergraduate, having introduced me both to academia and science. Thank you for encouraging my growth and teaching me so much.

To Thomas Schön, who welcomed me to his vibrant research group in Uppsala. It was a pleasure working on such diverse and exciting projects with you. Thomas had a significant influence on the way I choose scientific problems to study and the way I organize and communicate my ideas. And, for all these things, I am very grateful.

To my brother Manoel, who has been such a friend along the way. And whose influence was so relevant for the development of this thesis. Ranging from the choice of the group, I would end up spending one year with, to discussions that would change my mind about so many topics.

To my Dad, for showing me all the love and being a role model for me. It was very exciting working in the automated analysis of the electrocardiogram with you.

To my friends and colleagues in Uppsala, for making my time in Sweden so great and for making me feel so welcome. I would like to sincerely thank my co-authors Koen, Jack, Carl Andersson, and Niklas. It was a pleasure working with all of you. My office mates Carl Jidling, Ludvig, Koen, David, Fredrik, for creating such a great office environment. And, also, Helena, Anna, Fredrik, Andreas, Johan, Johannes, Pelle, Riccardo, Bengt, Daniel and all others for the friendship and hospitality. Working in such a vibrant group was essential for my formation.

To the friends and colleagues from CPH at UFMG: Arthur, Leandro, Leo, Petrus, Felipe, and Joao, thank you for creating such a friendly workplace.

To my colleagues from the CODE group: Gabi, Paulo, Derick, Jessica, Emily, and all others. I sincerely thank you all for the collaboration. I was very happy to be part of such a group.

To the entire Scipy community, for introducing me to open source development. A special thanks to Matt, Nikolay, and Ralf for mentoring me. And to Google for sponsoring me through Google Summer of Code program.

To the Brazilian taxpayers, for financially supporting me through CAPES and CNPq on which I have subsisted for the past four years, and which made my staying in Uppsala possible.

To my Mother, for showing me all love and support in the world and for always believing in me.

To my beloved girlfriend, Nicolle, for sweeping me off my feet and for being with me every day, even when I was far away.

*“Someone who breaks a puzzle because they cannot solve it
has left the Lethani. I am not such one.”*

The Wise Man’s Fear

Patrick Rothfuss

Abstract

Building empirical models from data is of fundamental importance in engineering, and pushing the boundaries of current engineering technology requires us to model and understand nonlinear systems. In this thesis, nonlinear differentiable models and its applications are studied. This class of models has gained traction in machine learning tasks with the introduction of *deep learning*. Indeed, deep models of stacked differentiable components have recently achieved super-human performance on several tasks, including computer games, image classification, and medical diagnosis. The application of nonlinear differentiable models is studied for modeling signals and systems both for engineering and machine learning applications. One central question is the role of recurrence and the pros and cons of recurrent and feedforward models. The question is approached from more than one angle: 1) by studying the effect of recurrence in neural networks in terms of robustness to noise, computational cost, and convergence; 2) by analyzing the smoothness of the cost function in nonlinear system identification problems and its relation to the model internal dynamics – and proposing the use of a technique called multiple shooting for improving the cost-function smoothness; and, 3) by investigating the interplay between the internal dynamics, the attractors and the expressiveness of the model in deep recurrent neural networks. The more applied part of the thesis consists of the use of deep neural networks to solve complex tasks and to model nonlinear behavior from real data. Data from the Telehealth Center of Minas Gerais is used to train a deep neural network capable of identifying abnormalities in the electrocardiogram exam with performance superior to the medical residents in the studied scenario. Also, a deep neural network is used for modeling an electronic oscillator and an F-16 aircraft using data from ground vibration experiments, obtaining competitive results in both cases.

Key-words: Nonlinear systems, differentiable models, deep learning, system identification, machine learning

Resumo

Construir modelos empíricos a partir de dados é de fundamental importância em engenharia e, além disso, o entendimento e a capacidade de modelar sistemas não lineares são necessários para o desenvolvimento de tecnologias de fronteira. Nesse trabalho, modelos diferenciáveis não lineares e suas aplicações são estudados. Esta classe de modelos tem ganhado força na área de aprendizado de máquina com a introdução do *aprendizado profundo*. De fato, modelos profundos de componentes diferenciáveis alcançaram, recentemente, desempenho superior ao humano em diversas tarefas, incluindo a competição em jogos digitais, classificação de imagens e diagnóstico de exames médicos. A aplicação de modelos não lineares diferenciáveis é estudada para modelar sinais e sistemas, tanto no contexto de aplicações em engenharia quanto no contexto de aprendizado de máquina. Uma questão central é o papel da recorrência, e os prós e os contras de modelos recorrentes. A questão é abordada de mais de um ângulo: 1) estudando o efeito da recorrência em redes neurais em termos da robustez a ruído, custo computacional e convergência; 2) analisando a suavidade da função de custo na identificação de sistemas não lineares e a relação com a dinâmica interna do modelo – e propondo o uso da técnica de múltiplos tiros para melhorar a suavidade da função custo; e, 3) investigando a relação entre dinâmica interna, atratores e expressividade do modelo em redes neurais recorrentes. A parte mais aplicada desta tese consiste no uso de redes neurais profundas para resolver tarefas complexas e modelar comportamento não linear a partir de dados reais. Dados do Centro de Telessaúde do estado de Minas Gerais são usados para treinar uma rede neural capaz de identificar abnormalidades no eletrocardiograma com desempenho superior ao de residentes de medicina no cenário estudado. Além disso, uma rede neural profunda é usada para modelar um oscilador eletrônico e uma aeronave F-16 usando dados de um ensaio de vibrações, obtendo resultados competitivos nos dois casos.

Palavras-chave: Sistemas não lineares, modelos diferenciáveis, aprendizado profundo, identificação de sistemas, aprendizado de máquina

List of Figures

Figure 1.1 – (Convergence rate) We show the first 10 points of sequences that converge toward zero. The sequence $(0.7)^n$ converges linearly, n^{-n} converges superlinearly and, $(0.7)^{2^n}$ converges quadratically.	40
Figure 1.2 – (Scaling) We show the contour plot and the first iterations of steepest descent algorithm with fixed step size (in red) . Figures (a) to (c) display the contour plot for the cost function $V(\theta_1, \theta_2) = \theta_1^2 + r\theta_2^2$, which has a global minimum at $(0, 0)$. We start the steepest descent algorithm from the initial point $(2, 2)$ and show the first 10 iterations. Figures (d) to (f) display the contour plot for the cost function $V(\theta_1, \theta_2) = (1 - \theta_1)^2 + r(\theta_2 - \theta_1^2)^2$, which has a global minimum at $(1, 1)$. The function is known as the <i>Rosenbrock function</i> . We start the steepest descent algorithm from the initial point $(-1, 1)$ and show the first 100 iterations. In both cases, r is used to adjust the <i>scaling</i> of the problem, with large values of r corresponding to ill-scaled problems. The fixed step size of the optimization algorithm is fixed to $0.1/r$ to avoid divergence.	41
Figure 1.3 – (Computational graph) Graph containing the partially ordered elementary operations involved in the computation of f , defined as in (1.44). The <i>leaves</i> of the computational graphs are the nodes in red and correspond to the function inputs.	43
Figure 2.1 – (Causal convolutions) Simplified diagram illustrating causal convolutions from Eq. (2.15) for kernel size $n = 3$, input vector size $n_z = 3$, output vector size $n_x = 2$. Here the input sequence has length $N = 8$ and the output sequence has length $M = N - n + 1 = 6$. The computation is shown for the first 4 output samples.	53
Figure 2.2 – (Padding) Simplified diagram illustrating the effect of padding in input and outputs for causal convolutions. In (a), no padding is used and the input sequence has length $N = 8$ and the output sequence has length $M = N - n + 1 = 6$. In (b), we use padding in order to guarantee inputs ($P = 2$) and outputs with the same length: the zero-filled input sequence has length $N + P = 8 + 2$ and the output sequence has length $M = N + P - n + 1 = 8$. In both cases, the input vector size is $n_x = 3$ and the output vector size is $n_z = 2$. The zeros included in the input are shown in black . Output elements that are being computed using these zeros as input are displayed in gray , considering the convolutional layer as a dynamic system these elements correspond to the transient response of this system from a zero-state equilibrium.	54
Figure 2.3 – (Strides) Simplified diagram illustrating causal convolutions with strides for kernel size $n = 3$, input vector size $n_x = 3$, output vector size $n_z = 2$, and downsampling factor $m = 2$. Here the input sequence \mathbf{x} has length $N = 10$ and the output sequence \mathbf{z} has length $M = (N - n + 1)/2 = 4$. The intermediary sequence \mathbf{y} has length $M = (N - n + 1) = 8$. The computation is shown for the first 4 output samples.	54
Figure 2.4 – (Dilations) Simplified diagram illustrating causal convolutions with dilations for kernel size $n = 3$, input vector size $n_x = 3$, output vector size $n_z = 2$, and dilation factor $d = 2$. Here the input sequence \mathbf{x} has length $N = 10$ and the output sequence \mathbf{z} has length $M = N - d * (n - 1) = 6$. The computation is shown for the first 4 output samples.	55
Figure 2.5 – (Polyphase decomposition) Simplified diagram illustrating the input and output of causal convolutions with dilations. Here we show the polyphase components of the input and output for $d = 2$ in (a) and (b); and for $d = 3$ in (c) and (d). In (a) and (c) we show the original signal and in (b) and (d) we show the signal decomposed into its d polyphase components. Convolutions with dilations applied to original signal (a) and (c), can be interpreted as standard convolutions applied to the polyphase components shown in (b) and (d).	55
Figure 2.6 – (Convolutional network for feature extraction) Simplified diagram illustrating a 3 layer convolutional neural network for extracting information from a sequence. The first layer has length $N^{(1)} = 22$ and $n_x^{(1)} = 1$ channels, the second layer has length $N^{(2)} = 10$ and $n_x^{(1)} = 2$ channels, the third layer has length $N^{(3)} = 4$ and $n_x^{(1)} = 4$ channels. The elements from the last layer are flattened, and fed into a fully connected layer. A color code is used to illustrate how the flattening occurs in the last layer of the neural network.	56
Figure 2.7 – (Convolutional network modeling input-output relations) Simplified diagram illustrating a 3 layer convolutional neural network for modeling the input-output relation between signals. The dilation factor for the three layers is 1, 2, 4. All layers use a kernel size $n = 2$. The input and output have dimension 1, and the intermediary layers dimension 2 and 4, respectively. The color code in (a) shows the different polyphase components (cf. Section 2.3.4) in the input, output and intermediary signals, the same colors are used in (b) to denote the same components. The purpose is to illustrate how the use of dilations is equivalent to that of downsampling and strides in the convolutional neural network of Figure 2.6.	58

Figure 3.1 – (Dynamical system) Display a schematic representation of the model used to represent a dynamical system.	59
Figure 4.1 – (Parallel and series-parallel) Display neural network training modes. In both cases, considers the dynamic system model $\mathbf{y}[k] = \mathbf{F}(\mathbf{y}[k-1], \mathbf{y}[k-2], \mathbf{u}[k-1], \mathbf{u}[k-2])$. The block q^{-1} holds and delay the output by one sample period.	70
Figure 4.2 – (Fully connected neural network) Display three-layer feedforward network.	74
Figure 4.3 – (Pilot plant validation) Displays free-run simulation in the validation window for models obtained using series-parallel (SP) and parallel (P) training. The mean square errors are $\text{MSE}_{\text{SP}} = 1144.6$; $\text{MSE}_{\text{P}} = 296.2$. The models have $n_y = n_u = 1$ and 10 nodes in the hidden layer and were trained on a two hour long dataset sampled at $T_s = 10s$. The same initial parameter guess was used for both training methods. The training was 100 epochs long, which took 3.3 and 3.9 seconds, respectively, for series-parallel and parallel training.	80
Figure 4.4 – (Series-parallel vs parallel training) Boxplots show the distribution of the free-run simulation MSE over the validation window for models trained using series-parallel (SP) and parallel (P) methods under the circumstances specified in Figure 4.3. There are 100 realizations of the training in each boxplot, in each realization the weights $w_{i,j}^{(n)}$ are draw from a normal distribution with standard deviation $\sigma = (N_{s(n-1)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros (LeCun et al., 1998). For comparison purposes, the dashed horizontal line gives the performance of an ARX linear model ($n_y = 1$ and $n_u = 1$) trained and tested under the same conditions.	80
Figure 4.5 – (Toy problem validation) Displays the first 100 samples of the free-run simulation in the validation window for models trained using series-parallel (SP) and parallel (P) methods. The mean square errors are $\text{MSE}_{\text{SP}} = 0.39$; $\text{MSE}_{\text{P}} = 0.06$. The models have $n_y = n_u = 2$ and a single hidden layer with 10 nodes. The training set has $N = 1000$ samples and was generated with (7.6) for v and w white Gaussian noise with standard deviations $\sigma_v = 0.1$ and $\sigma_w = 0.5$. The validation window is generated without the noise effect. For both, the input u is randomly generated with standard Gaussian distribution, each randomly generated value held for 5 samples. The training was 100 epochs long, which took 5.0 and 6.1 seconds for, respectively, series-parallel and parallel training.	81
Figure 4.6 – (White noise during training) Free-run simulation MSE on the validation window <i>vs</i> noise levels for series-parallel and parallel training. The main line indicates the median and the shaded region indicates interquartile range. These statistics were computed from 12 realizations. In (a) v is a Gaussian white process and $w = 0$; and, in (b) w is a Gaussian white process and $v = 0$	82
Figure 4.7 – (Colored noise during training) Free-run simulation MSE over the validation window <i>vs</i> standard deviation of colored equation error. The output error $w = 0$ and the equation error v is a colored Gaussian noise obtained by applying a 4th-order lowpass Butterworth filter with cutoff frequency ω_c to white Gaussian noise in both the forward and reverse directions. The figure shows the result for different values of ω_c , where ω_c is the normalized frequency (with $\omega_c = 1$ the Nyquist frequency). The main line indicates the median and the shaded region indicates interquartile range. These statistics were computed from 12 realizations.	83
Figure 4.8 – (Computational cost) Running time (in seconds) of training (100 epochs). The average running time of 5 realizations is displayed for series-parallel training (●) and for parallel training (◆). The neural network has a single hidden layer and a total of N_{Θ} parameters to be estimated. The training set has N samples and was generated as described in Figure 7.1. In (a), we fix $N_{\Theta} = 61$ and plot the timings as a function of the number of training samples N , a line is adjusted to illustrate the running time grows linearly with the training size for both training methods. In (b), we fix $N = 10000$ and plot the timings as a function of the number of parameters N_{Θ} , a second order polynomial is adjusted to illustrate the running time quadratic growth.	85
Figure 5.1 – (Feedforward vs Recurrent) Display feedforward and recurrent structures used by prediction error method. Prediction error methods estimate the process parameters by minimizing the error e between the measured output y and the model predicted value \hat{y} . The input-output pairs (u, y) are measured values from some dynamic process. In (a) the prediction \hat{y} depends only on measured values: ARX model; In (b) the predictor has a recurrent structure: ARMAX and OE models.	86

- Figure 5.2 – **(Multiple shooting)** Comparison between single shooting and multiple shooting in a unidimensional example ($N^x = 1$). In **black** we present the simulation of the dynamic system through the entire window length using the single initial condition x_0 (represented by \bullet). The simulated values are represented by \blacklozenge . Dividing the window length into three sub-intervals and simulating the system in each of these, for initial conditions x_0^1 , \bullet , x_0^2 , \bullet , and x_0^3 , \bullet , results in the three different simulations represented by \blacklozenge , \blacklozenge and \blacklozenge , respectively. In (a), the end of one simulation does not coincide with the beginning of the next one ($x^{i-1}[m_i] \neq x_0^i$). In (b), we show what happens as $\|x^{i-1}[m_i] - x_0^i\| \rightarrow 0$. And, in (c), we show that, when $x^{i-1}[m_i] = x_0^i$, the concatenation of these short simulations is equivalent to a single one carried out over the entire window length. 90
- Figure 5.3 – **(Logistic map parameter estimation)** *Cost function* of the optimization problem for \mathbf{x}_0^i fixed in its true values (in black) and for disturbed versions of these initial conditions (in blue). We present the result for four values of Δm_{\max} , and omit disturbed initial condition objective functions for the single shooting case ($\Delta m_{\max} = N$) to make it easier to visualize. The green circles, \bullet , indicate the pair (θ, V) corresponding to a solution found by the solver. There are 15 circles in each figure (some of them overlapping), the circles correspond to solutions for different initial guesses. As initial guesses we picked values of θ uniformly spaced between 3.2 and 3.9, with \mathbf{x}_0^i picked from randomly disturbed versions of the true initial conditions (which are known because we generated the data ourselves). The true value $\theta = 3.78$ is indicated by the dotted red vertical line. 94
- Figure 5.4 – **(Neural network multiple shooting estimation)** *Empirical cumulative distribution* of the free-run simulation MSE over the validation dataset. The results obtained in (Ribeiro and Aguirre, 2018) for an ARX neural network (NN ARX) and an (single shooting) output error neural network (NN OE - SS) are displayed together with the result obtained estimating the parameters using multiple shooting (NN OE - MS). A Linear ARX model is considered as a baseline and is displayed by the dashed line. The multiple shooting estimation uses $\Delta m_{\max} = 3$ and the training is restricted to 2000 iterations of the optimization algorithm or until either the gradient or the step size drops below 10^{-12} . The other models were estimated exactly as in (Ribeiro and Aguirre, 2018). All the neural network models have 10 nodes in the hidden layer, $n_y = n_u = 1$ and were trained with the same training dataset. Each curve is the result of 100 realizations and, for each realization, the neural network initial weights $w_{i,j}^{(n)}$ are drawn from a normal distribution with zero-mean and standard deviation $\sigma = (N_{s(n-1)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros (LeCun et al., 1998). Realizations of NN OE - SS and NN OE - MS that perform worse than the baseline are indicated respectively as blue, \bullet , and red circles, \bullet . Confidence intervals (95%) are displayed as shaded regions around the estimated cumulative distribution, these have been computed using the Dvoretzky-Kiefer-Wolfowitz inequality (Dvoretzky et al., 1956). 97
- Figure 5.5 – **(Pendulum and inverted pendulum position data)** Display the *output signal* $y[k]$ for the three different datasets used for estimating the parameters. The dataset size is $N = 1024$ samples. (a) The input applied in this case is a zero-mean Gaussian random input with standard deviation $\sigma_u = 10$, each random value being hold for 20 samples. The input in this case is unable to drive the system away from the influence of the stable fixed point $(0, 0)$. (b) The input $u[k]$ in this case is obtained by the control law: $u[k] = 40\delta e[k-1] - 78.8\delta e[k-2] + 38.808\delta e[k-3] + 1.02u[k-1] - 0.02u[k-2]$, where the error is the difference between the reference and the output: $e[k] = r[k] - y[k]$. The reference is $r[k] = \pi + \Delta r[k]$ where $\Delta r[k]$ is a zero-mean Gaussian random input with standard deviation $\sigma_r = 0.2$, each random value being held for 20 samples; and, (c) Same thing as (a) but with the larger standard deviation $\sigma_u = 50$, which is able to drive the pendulum to complete full rotations. For (a) and (c) zero-mean Gaussian white noise with standard deviation $\sigma_r = 0.03$ was added to the output. 98
- Figure 5.6 – **(Pendulum and inverted pendulum parameter estimation)** *Contour plot of the cost function*. Figures (a), (b) and (c) correspond to the cost function V from single shooting simulation for the datasets (a), (b) and (c) generated as described in Fig. 5.5 caption; and, in (d), (e) and (f) the cost function for the same problems is displayed for the multiple shooting formulation with $\Delta m_{\max} = 16$. The true parameter is indicated by a red circle, \bullet , solutions found by the solver are indicated by blue circles, \bullet . There are 25 blue circles in each figure (some of them overlapping), each circle corresponds to the solution for a different initial guess. Some solutions are outside of the displayed region and the corresponding blue dots are displayed at the edge of the plot. Initial guesses were picked values of θ uniformly spaced on a grid of points uniformly spaced in the rectangle $[20, 50] \times [0.5, 6]$. It is important to highlight that the plots show a two dimensional projection of a cost function that is defined on an extended parameter space that includes the initial conditions \mathbf{x}_0^i , $i = 1, \dots, M$ as parameters, which were fixed to the true values when generating the contour plots. 99
- Figure 6.1 – **(Chaotic LSTM)** Display: a) Bifurcation diagram; and b) cost function (mean-square error) for LSTM models with parameter vectors $\theta(s) = s\theta_{\text{true}}$ 102

Figure 6.2 – (Teacher forcing) The figure display the same recurrent neural network in two different modes. In (a), observed previous values of the output \mathbf{y}_t are used as input to the model. In (b), the neural network own outputs are feed back as inputs, instead.	103
Figure 6.3 – (Information in RNNs) Illustrate the lower bound on the entropy H_t obtained in (6.5). Starting from H_0 , the entropy H_t can only take values in the shaded region. The entropy: a) may decay over time (contractive system); b) may remain constant (e.g. for a periodic oscillator); and, c) increases when $L_f > 1$ (e.g. for a chaotic system).	105
Figure 6.4 – (Sine wave generator training history) Mean square error per epoch for the <i>sine-wave generation task</i> during training. The baseline performance is indicated by the dashed line. The final performance of the LSTM after 4,000 epochs is $4.8 * 10^{-3}$, the final performance of the stable LSTM (sLSTM) is 0.49 and the final performance of the orthogonal RNN (oRNN) is $4.1 * 10^{-3}$	109
Figure 6.5 – (The mechanism for learning oscilations) Bifurcation diagram during training for the <i>sine-wave generation task</i> showing the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that varies from 0 to 1500. The bifurcation diagram is for a constant input $\frac{\pi}{8}$, other values will yield similar plots.	109
Figure 6.6 – (Symbol classification training history) Accuracy on <i>validation data set</i> for the recurrent models trained to perform the same <i>symbol classification task</i> for two different sequence lengths. The baseline performance (always predicting $\{p, p\}$) is indicated by the dashed line. In (a) two x-axis scales co-exist in the same graph, one scale in $[0, 250]$ and other in $[250, 15250]$, with a relation 1:40 between the two scales.	110
Figure 6.7 – (The effect of the initial conditions) Accuracy on training and validation data on the symbol classification task for the LSTM model in identical scenarios but with different initial random parameter initialization (from different random seeds). In (a), the optimization procedure abruptly finds a solution that has good accuracy on both training and validation; while, in (b), the convergence is slow and steady towards a solution that has good accuracy on the training set (above 90%) but is no better than random guessing on the validation set.	111
Figure 6.8 – (The LSTM mechanism for learning to remember symbols) Bifurcation diagram for the <i>LSTM</i> model in <i>symbol classification task</i> for sequences of length 100. It shows the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that vary from 0 to 400.	111
Figure 6.9 – (The oRNN mechanism for learning to remember symbols under noise) Bifurcation diagram for the <i>oRNN</i> model in <i>symbol classification task</i> for sequences of length 100. It shows the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that vary from 0 to 400.	112
Figure 6.10 – (Word-level language model training history) Perplexity on <i>validation data set</i> for <i>word-level language model</i>	114
Figure 6.11 – (The LSTM mechanism for learning a language model) Bifurcation diagram for the <i>LSTM world-level language model</i> . For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ and its first difference $p(\mathbf{x}_t) - p(\mathbf{x}_{t-1})$ after a burnout period of 1500 samples. This burnout period is used to remove the transient reponse and yields a visualization of the system attractors, per epoch. The arrow point towards the evolution of the number of epochs, that varies from 0 to 150. In (a) and (b), we have two different realizations of the bifurcation diagram obtained from constant inputs. In (c) and (d), the diagram is generatad using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in (a) and (b), respectively. The subplots (a) to (d) use the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$. The second row, (e) to (h), and third row, (i) to (l), show the exact same experiments but for the projections in the direction of the tokens “is” and “Valkyria”, respectively.	115
Figure 6.12 – (The sLSTM mechanism for learning a language model) Bifurcation diagram for the <i>stable LSTM world-level language model</i> . For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ after a burnout period of 1500 samples. This burnout period is used to remove the transient reponse and yields a visualization of the system attractors, per epoch. In the displays (a) to (d), the diagram is generated for the same constant input. In (e) to (h), the diagram is generatad using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in the first row. The projections are: the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$; and, projections into the direction of the tokens “is”, “Valkyria” and “<unk>”.	116

Figure 6.13 –	(The oRNN mechanism for learning a language model) Bifurcation diagram for the <i>orthogonal RNN world-level language model</i> . For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ after a burnout period of 1500 samples. This burnout period is used to remove the transient reponse and yields a visualization of the system attractors, per epoch. In the displays (a) to (d), the diagram is generated for the same constant input. In (e) to (h), the diagram is generatad using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in the first row. The projections are: the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$; and, projections into the direction of the tokens “is”, “Valkyria” and “<unk>”.	117
Figure 7.1 –	(Temporal convolution network) Illustrate the temporal convolution network (TCN) with residual blocks. (a) Temporal convolutional network with dilated causal convolutions with dilation factor $d_1 = 1$, $d_2 = 2$ and $d_3 = 4$ and kernel size $n = 3$. (b) A TCN residual block. Each block consists of two dense layers and an identity (or linear) map on the skip connection. As illustrated in (a) by connection with the same color, neural network weights are shared within the same layer and invariant to time translations. This reflects the hypothesis we are modeling a time invariant system.	121
Figure 7.2 –	(Toy problem validation) Displays 100 samples of the free-run simulation TCN model vs the simulation of the true system. The kernel size for the causal convolutions is 2, the dropout rate is 0, it has 5 convolutional layers and a dilation rate of 1. The training set has 20 batches of 100 samples and was generated with (7.6) for v and w white Gaussian noise with standard deviations $\sigma_v = 0.3$ and $\sigma_w = 0.3$. The validation set has 2 batches of 100 samples. For both, the input u is randomly generated with a standard Gaussian distribution, each randomly generated value held for 5 samples.	125
Figure 7.3 –	(Ablation study) Box plots showing how different design choices affect the performance of the TCN for noise standard deviation $\sigma = 0.3$ and training data length $N = 2000$. On the y -axis the one-step-ahead RMSE on the validation set is displayed, and on the x -axis we have: in (a) the presence or absence of dilations; in (b) the dropout rate $\{0.0, 0.3, 0.5, 0.8\}$; in (c) the number of residual blocks $\{1, 2, 4, 8\}$; and, in (d) if <i>batch norm</i> , <i>weight norm</i> or nothing is used for normalizing the output of each convolutional layer. The variation in performance for the box plot quartiles is achieved through the variation for all the other hyper-parameters not fixed by the hyper-parameter choice indicated on the x -axis.	126
Figure 7.4 –	(Silverbox prediction error and data) The true output and the prediction error of the TCN model in free-run simulation for the Silverbox data. The model needs to extrapolate approximately outside the region ± 0.2 marked by the dashed lines.	127
Figure 7.5 –	(TCN depth vs performance) Box plot showing how different depths of the neural network affects the performance of the TCN in the Example 3. Should be interpreted in the same way as Fig. 7.3.	127
Figure 7.6 –	(F-16 model main resonance) We plot the model frequency response in the interval $[4.7, 11]$ Hz. The true output spectrum is in black, the noise distortion, in grey dash-dotted line, the total distortion (= noise + nonlinear distortions), in grey dotted line, error LSTM, in green, error MLP, in blue, and error TCN, in red). All tested model structures perform similar andhe error is close to the noise floor around the main resonance at 7.3 Hz.	129
Figure 8.1 –	(Abnormalities examples) A list of all the abnormalities the model classifies. We show only 3 representative leads (DII, V1 and V6).	134
Figure 8.2 –	(Precision-recall curve) Show precision-recall curve for our nominal prediction model on the test set (strong line) with regard to each ECG abnormalities. The shaded region show the range between maximum and minimum precision for neural networks trained with the same configuration and different initialization. Points corresponding the performance of resident medical doctors and students are also displayed, together with the point corresponding to the DNN performance for the same threshold used for generating Table 8.2. Gray dashed curves in the background correspond to iso- F_1 curves (i.e. curves in the precision-recall plane with constant F_1 score).	135
Figure 8.3 –	(Bootstrapped scores) Display boxplots of empirical distribution of precision, recall, specificity and F1 score on the test set. Sampling with replacement (i.e. bootstrapping) from the test set was used to generate $n = 1000$ samples. The results are given for the DNN, the medical residents and students. Source data are provided as a Source Data file. The boxplots should be read as follows: the central line correspond to the median value of the empirical distribution, the box region correspond to the range of values between the first and third quartile (also knoww as interquartile range or IQR), the whiskers extend from 1.5 IQR below and above the firsth and third quartiles, values outside of that range are considered outliers and show as diamonds.	138

Figure 8.4 – **(Heart rate vs DNN predictions)** Heart rate measured by the Uni-G software for samples in the test set is given on the y -axis. The color indicates if the *DNN* make the correct prediction or not. The x -axis separates the dataset accordingly to the presence of: SB in (a); and, ST in (b). A horizontal line show the threshold of 50 bpm for SB in (a); and, of 100 bpm for ST in (b), which delimit the consensus definition of SB and ST. Notice that most exams for which the neural network fails to get the correct result are very close to this threshold line and are the borderline cases we mentioned in the discussion. It should be highlighted that this automatic measurement system is not perfect, and measurements that may indicate some of the conditions do not necessarily agree with our board of cardiologist (e.g. there are exams with heart rate above 100 according to Uni-G that are not classified by our cardiologist as ST).¹⁴⁰

Figure 8.5 – **(DNN architecture)** The uni-dimensional residual neural network architecture used for ECG classification.¹⁴³

List of Tables

Table 1.1 – (Design choices) Summary of the design choices in the numerical examples. The Levenberg-Marquardt algorithm is a nonlinear least-squares algorithm proposed by (Marquardt, 1963). The trust-region algorithm from Chapter 5 was described by (Lalee et al., 1998) and ADAM is a stochastic first-order with momentum proposed by (Kingma and Ba, 2014). [†] In Chapter 4 we actually use a mixed approach, where we use backpropagation (which uses reverse mode automatic differentiation) to compute the derivative of the neural network, and forward differentiation to propagate the derivatives through the recurrence. [‡] We did the same for Example 2 in Chapter 5.	46
Table 4.1 – (Backpropagation computational cost) Modified backpropagation number of <i>flops</i> for a fully connected network.	76
Table 4.2 – (Computational cost) Levenberg-Marquardt number of <i>flops</i> per iteration for series-parallel training (SP), parallel training with fixed initial conditions ($P\theta$) and parallel training with extended parameter vector ($P\Phi$). A mark X signals which calculation is required in each method.	77
Table 4.3 – (Noise in frequency bands during training) Free-run simulation MSE on the validation window for parallel and series-parallel training. Both the mean and the standard deviation are displayed (30% trimmed estimation computed from 12 realizations). In situation (a) , the training data was generated with zero output error ($w = 0$) and v is a Gaussian random process. In (b) , the training data was generated with $v = 0$ and w is a Gaussian random process. The Gaussian random process has standard deviation $\sigma = 1.0$ and power spectral density confined to the given frequency band. In both situations, the rows where the frequency ranges from 0.0 to 1.0 (the whole spectrum) corresponds to white noise, in the remaining rows we apply a 4th-order lowpass (or highpass) Butterworth filter to white Gaussian noise (in both the forward and reverse directions) in order to obtain the signal in the desired frequency band. The cell of the training method with the best validation results between the two models is colored. Its colored red when the difference in the MSE is larger than the sum of standard deviations and yellow when it is not.	84
Table 5.1 – (Optimization performance in multiple shooting) Number of function evaluations and total running time to convergence for different values of Δm_{\max} . We give, the minimum, maximum and median among 15 runs for the situations presented in Figure 5.3. (*)The number of iterations is limited to 1000 and the solver is interrupted when this number is reached.	95
Table 6.1 – (Symbol classification accuracy) Comparison between the LSTM; stable LSTM (sLSTM); and orthogonal RNN (oRNN); for the symbol classification task with different sequence lengths (l). As a baseline, always predicting $\{p, p\}$ would yield an accuracy of 0.25.	113
Table 7.1 – (Performance vs dataset size vs noise intensity) One-step-ahead RMSE on the validation set for the models (MLP, LSTM and TCN) trained on datasets generated with: different noise levels (σ) and lengths (N). The standard deviation of both the process noise v and the measurement noise w is denoted by σ . We report only the best results among all hyper-parameters and architecture choices we have tried out for each entry.	126
Table 7.2 – (Silverbox benchmark with no extrapolation) Free-run simulation results for the Silverbox example on part of the test data (avoiding extrapolation).	127
Table 7.3 – (Silverbox benchmark) Free-run simulation results for the Silverbox example on the full test data. (*Computed from FIT=92.2886%).	128
Table 7.4 – (Performance in F-16 benchmark) RMSE for free-run simulation and one-step-ahead prediction for the F16 example averaged over the 3 outputs. The average RMS value of the 3 outputs is 1.0046.	129
Table 7.5 – (Toy problem model hyperparameters) Best model hyperparameters in Example 1 for: different noise levels (σ) and lengths (N). The standard deviation of both the process noise v and the measurement noise w is denoted by σ	130
Table 8.1 – (Dataset summary) Patient characteristics and abnormalities prevalence, n (%).	134
Table 8.2 – (Performance indexes) Scores of our DNN are compared on the test set with the average performance of: i) 4th year cardiology resident (<i>cardio.</i>); ii) 3rd year emergency resident (<i>emerg.</i>); and, iii) 5th year medical students (<i>stud.</i>). (PPV = positive predictive value)	135

Table 8.3 – (Error analysis) Present the analysis of misclassified exams. The errors were classified into the following categories: i) measurements errors (<i>meas.</i>) were ECG interval measurements preclude the given diagnosis by its textbook definition ; ii) errors due to <i>noise</i> , were we believe that the analyst or the DNN failed due to a lower than usual signal quality; and, iii) other type of errors (<i>unexplain.</i>). Those were further divided, for the medical residents and students, into two categories: conceptual errors (<i>concep.</i>), where our reviewer suggested that the doctor failed to understand the definitions of each abnormality, and attention errors (<i>atte.</i>), where we believe the error could be avoided if the reviewer had been more careful.	136
Table 8.4 – (Confusion matrices) Show the absolute number of: i) false positives; ii) false negatives; iii) true positives; and, iv) true negatives, for each abnormality on the test set.	137
Table 8.5 – (Kappa coefficients) Show the Kappa scores measuring the inter-rater agreement on the test set. In (a), we compare the DNN, the medical residents and the students two at a time. In (b), we compare the DNN, and the certified cardiologists that annotated the test set (certif. cardiol.). If the raters are in complete agreement then it is equal to 1. If there is no agreement among the raters other than what would be expected by chance it is equal to 0.	137
Table 8.6 – (McNemar test) Display the <i>p</i> -values for the McNemar test comparing the misclassification on the test set. The DNN, the medical residents and the students were compared two at a time. Entries with statistical significance (with 0.05 significance level) are displayed in boldface	138

Contents

Introduction	21
I BACKGROUND	28
1 MACHINE LEARNING FUNDAMENTALS	29
1.1 Parameter estimation	29
1.1.1 Maximum likelihood	30
1.1.2 Normal distribution and the least-squares cost function	31
1.1.3 Multinomial distribution and the softmax layer	31
1.1.4 Bayes parameter estimation and maximum-a-posteriori	32
1.1.5 Prior and parameter penalty	33
1.2 Model complexity	33
1.2.1 Bias-variance tradeoff	34
1.2.2 Regularization methods and hyperparameter choice	35
1.2.3 Model final performance	36
1.3 Optimization	36
1.3.1 Global vs local optimization algorithms	36
1.3.2 Taylor approximations	37
1.3.3 Convexity	38
1.3.4 Line-search and trust-region methods	38
1.3.5 Convergence rate	39
1.3.6 Scaling	40
1.3.7 Memory complexity	40
1.3.8 Stochastic optimization	41
1.4 Automatic differentiation	42
1.4.1 Forward-mode automatic differentiation	44
1.4.2 Reverse-mode automatic differentiation	44
1.4.3 Computational complexity and applications	45
2 NEURAL NETWORK MODELS FOR SEQUENCES	47
2.1 Sequence models	47
2.2 Recurrent neural networks	49
2.2.1 Training and inference using recurrent neural networks	49
2.2.2 A short note about attention-based models	50
2.2.3 Common recurrent neural networks architectures	50
2.2.4 Bidirectional neural networks	51
2.2.5 Multi-layer recurrent neural network	52
2.3 Convolutional neural networks for sequences	52
2.3.1 The uni-dimensional convolutional layer	52
2.3.2 Padding and the transient response	53
2.3.3 Downsampling in convolutional neural networks: strides and pooling	53
2.3.4 Polyphase decomposition and dilations	55
2.3.5 Deep convolutional neural networks for feature extraction	56

2.3.6	Capturing input-output relation	57
2.3.7	Skip connections and the residual neural network	58
3	SYSTEM IDENTIFICATION AND PREDICTION ERROR METHODS	59
3.1	Mathematical representation of dynamical systems	59
3.1.1	Dynamic representation	60
3.1.2	Approximation function	61
3.1.3	Noise model	61
3.2	Prediction error methods	62
3.2.1	Setup	62
3.2.2	Nonlinear ARX models	63
3.2.3	Nonlinear output error models	63
3.2.4	Nonlinear ARMAX models	63
3.2.5	General nonlinear state-space framework	64
3.2.6	Initial conditions	64
3.2.7	The data generation process	64
3.2.8	Asymptotic properties	65
3.2.8.1	Optimal output prediction	65
3.2.8.2	Optimal cost function	65
3.2.8.3	Uniform convergence of V	65
3.2.9	Prediction error and maximum likelihood	66
3.2.10	Some terminology	66
3.2.11	Computing the derivatives	66
3.2.11.1	Sensitivity equations	66
3.2.11.2	Derivatives of the cost function	67
3.3	Test design and data collection	67
II	THE TRADE-OFFS OF RECURRENCE	68
4	PARALLEL TRAINING CONSIDERED HARMFUL?	69
4.1	Introduction	69
4.2	Unifying framework	71
4.2.1	Parallel vs series-parallel training	71
4.2.2	Optimal predictor	72
4.3	Nonlinear least-squares network training	72
4.3.1	Nonlinear least-squares	73
4.3.2	Modified backpropagation	74
4.3.3	Series-parallel training	75
4.3.4	Parallel training	75
4.4	Complexity analysis	76
4.4.1	Neural network output and its partial derivatives	76
4.4.2	Number of flops for series-parallel and parallel training	76
4.4.3	Comparing methods	77
4.4.4	Memory complexity	77
4.5	Practical aspects	78
4.5.1	Convergence towards a local minima	78
4.5.2	Signal unboundedness during training	78

4.5.3	Time series prediction	79
4.6	Implementation and test results	79
4.6.1	Example 1: data from a pilot plant	79
4.6.2	Example 2: investigating the noise effect	81
4.6.3	Timings	82
4.7	Conclusion	83
5	ON THE SMOOTHNESS OF NONLINEAR SYSTEM IDENTIFICATION	86
5.1	Notation and setup	87
5.2	Smoothness of prediction error methods	88
5.3	Multiple shooting	89
5.3.1	Method formulation	89
5.3.2	Properties of the objective function	91
5.3.3	Comparison with other methods	91
5.4	Implementation	92
5.5	Numerical examples	93
5.5.1	Example 1: output error model for chaotic system	94
5.5.2	Example 2: neural network for modeling pilot plant	96
5.5.3	Example 3: pendulum and inverted pendulum	97
5.5.4	Practical considerations	99
5.6	Conclusion	100
6	ATTRACTOR AND SMOOTHNESS IN THE ANALYSIS OF RECURRENT NEU- RAL NETWORKS TRAINING	101
6.1	Introduction	101
6.2	Setup and notation	102
6.3	Information in a recurrent network	103
6.3.1	Entropy of the internal states	104
6.3.2	Contractive vs non-contractive systems	105
6.3.3	Long-term memory	105
6.4	Smoothness of the cost function	106
6.4.1	Example: chaotic LSTM	107
6.4.2	Theoretical results	107
6.5	Attractors evolution during training	108
6.5.1	Sine wave generation	108
6.5.2	Symbol classification	110
6.5.3	Word-level language model	113
6.6	Related work	116
6.7	Discussion	117
III	MODELING SIGNALS AND SYSTEMS: APPLICATIONS TO ENGINEER- ING AND HEALTH CARE	119
7	DEEP CONVOLUTIONAL NETWORKS IN SYSTEM IDENTIFICATION	120
7.1	Neural networks for temporal modeling	120
7.1.1	Temporal convolutional network	121
7.1.2	Regularization	122

7.1.3	Batch normalization	122
7.1.4	Optimization algorithms	123
7.1.5	The residual block	123
7.2	Connections to system identification	123
7.2.1	Connection with Volterra series	123
7.2.2	Connection with block-oriented models	124
7.2.3	Conclusion	124
7.3	Numerical results	124
7.3.1	Example 1: nonlinear toy problem	125
7.3.2	Example 2: Silverbox	126
7.3.3	Example 3: F-16 ground vibration test	127
7.3.4	Hyperparameter search and training time	128
7.3.4.1	Nonlinear toy problem	129
7.3.4.2	Silverbox	129
7.3.4.3	F-16 ground vibration test	130
7.4	Conclusion and future work	131
8	AUTOMATIC DIAGNOSIS OF THE 12-LEAD ECG USING A DEEP NEURAL NET- WORK	132
8.1	Results	133
8.1.1	Model specification and training	133
8.1.2	Testing and performance evaluation	135
8.2	Discussion	137
8.3	Methods	141
8.3.1	Dataset acquisition	141
8.3.2	Labelling training data from text report	141
8.3.3	Training and validation set annotation	142
8.3.4	Test set annotation	143
8.3.5	Neural network architecture and training	143
8.3.6	Hyperparameter tuning	144
8.3.7	Statistical and empirical analysis of test results	144
	Conclusion	146
	APPENDIX A – SEQUENTIAL QUADRATIC PROGRAMMING SOLVER	149
	APPENDIX B – PROOFS	152
	Bibliography	156

Introduction

Nonlinear systems and non-convex problems

Mathematical models of dynamic systems are of fundamental importance for science and engineering. Linear models of dynamic systems are extremely popular due to the strong theoretical foundations behind them. They are widely studied and are the standard choice for most engineering applications. However, pushing the forefront of current engineering technology requires us to better understand, model, and control nonlinear systems. To cite two examples: using flexible wings open possibilities for significantly more efficient airplanes (Ifju et al., 2002; He and Zhang, 2017); and, using amplifiers near the saturation region is required to meet the needs of the future generation of wireless communication networks for high-data-rate transmission, a large number of users and power-efficient devices (Singya et al., 2017). These two applications require modeling and controlling nonlinear systems, explicitly taking into account the nonlinear behavior, and both could have a very high impact on society. More efficient airplanes are needed to conciliate our need to move around the globe with the enormous impact of aviation on human emissions. It could also open exciting new possibilities, such as taking us closer to making electric airplanes commercially viable. Better wireless communication networks could have a significant impact on human society by allowing simple physical objects to become internet-connected-devices (also known as the *internet of things*), which comes with a wide new range of possibilities (Ashton, 2009; Xia et al., 2012).

The basic laws that govern many of the basic electrical, mechanical, and chemical processes that are of our interest in engineering are well understood. Complex interactions and combinations of well-understood components, however, often yield processes that cannot be easily modeled using basic principles. Hence, building simplified empirical models for these complex engineering processes directly from data recorded during experiments presents an interesting alternative.

Identifying dynamical systems from empirical data caught a lot of interest from the control community, and a field named *system identification* was a mature field by the end of the 20th century (Söderström and Stoica, 1988; Ljung, 1998). The identification of a dynamic system can be formulated as an optimization problem that tries to adjust the model predictions to a training dataset. One important class of optimization problems are convex optimization problems: for these problems, it is easier to establish uniqueness and existence of solutions. There are also efficient algorithms with strong guarantees to solve this type of problem. When identifying nonlinear models, however, there is a wide range of problems that fall into non-convex problems. Even for linear models, it is easy to fall into cases for which the optimization problem is not convex due to the recurrence required to model some types of noise that are often found in practice.

To tackle the non-convex identification of nonlinear models into its full complexity, however, can be highly rewarding and could yield powerful tools for the identification of nonlinear systems. One interesting class of models that encompass several nonlinear models of interest is that of (almost everywhere) differentiable models. The differentiability of the model translates into a differentiable optimization problem which can be solved efficiently by using local, low-order, Taylor approximations of the objective function. And, while there aren't the same guarantees as in the case of a convex optimization problem, it is still possible to find a solution that is locally optimal. Interestingly, highly nonlinear differentiable models yielding non-convex optimization problems have recently had a high impact in another related field: *machine learning*.

Teaching what you cannot explain

Mathematical empirical models have a crucial role in artificial intelligence, and they are at the core of the sub-field called *machine learning*. *Machine learning* is a thriving field that studies how to create machines capable of learning from examples. The ultimate goal is to enable computers to perform tasks usually associated with human (or animal) intelligence: such as understanding language, being able to process visual input, and being able to make decisions under complex situations. While many of these tasks come naturally to people, some are hard to describe formally, and it has been a challenge to solve them using traditional techniques.

For instance, it is easy for most of us to distinguish between a cat and a dog or to detect hostility in someone's voice. We can do that almost automatically. Nevertheless, it is hard to explain how we are doing that. Once learned, other tasks such as understanding language and driving also become natural and automatic. All these tasks are hard to define and traditional programming paradigms usually have a hard time solving such problems. Machine learning presents itself as a way to enable computers to solve such tasks. It does so by *teaching*, by examples, the computer to solve these tasks instead of trying to create a recipe for it.

Along the last decade, *machine learning* has had many successes, and there were breakthroughs in image classification (Krizhevsky et al., 2012), machine translation (Bahdanau et al., 2014; Vaswani et al., 2017) and speech recognition (Hinton et al., 2012), to cite a few. The driving force behind these recent achievements is the development of deep neural networks (LeCun et al., 2015) and the pioneer researchers in the field (Yoshua Bengio, Geoffrey Hinton, Yann LeCun) have been awarded the 2018 Turing Award recognizing the importance of deep neural networks as components in computing systems. The Turing Award is recognized as the highest distinction in computer science.

Deep neural networks are models, loosely inspired by neurological processes, composed of many stacked layers that combine linear and nonlinear transformations. The success of deep neural networks made popular a *machine learning* paradigm called *end-to-end learning*, which is often used together with these models. This paradigm feeds the raw input into the model, in contrast with the more traditional machine learning approach of using handcrafted features as inputs to the model. Take image classification as an example: the more traditional approach was to use image processing techniques to detect corners, shapes, and other image features, and using the obtained representation of the image as input to the classifier. In contrast, end-to-end learning uses the raw image as input to the classifier, and the model learns, by exposition to examples, both to extract the features and to use them to perform the classification.

Part of deep neural networks and end-to-end learning success is due to both the increasing availability of data and computational power. On the one hand, as our society became connected and transmission and storage of information became cheaper, extensive records become naturally available and could be used for training the models. On the other hand, increasing computational power made it possible to train larger models that make good use of such large datasets.

Great power and great responsibility

For better or for worse, the full development and deployment of current deep learning technology will be disruptive for our society. For instance, according to the Brazilian statistic and geographic institute IBGE, the work of more than 3.5 million people in Brazil in 2018 was related to transporting, in a vehicle, goods or people (Amorim, 2019). This is more than 1 percent of the Brazilian population. The development of self-driving cars, which use deep neural networks as a critical component, could displace many of these workers by having them competing with machines that don't sleep or eat and can perform the same job

24 hours a day, seven days a week.

Machine learning algorithms also have the potential to immensely improve several aspects of our lives, and one area where it might benefit society the most is in health care. Deep learning models have had a lot of success in automatic exams diagnosis. For instance, deep learning algorithms have recently obtained better performance than a human specialist in routine work-flow of diagnosing breast cancer (Bejnordi et al., 2017) and in detecting retinal diseases from three-dimensional optical coherence tomography scans (De Fauw et al., 2018). In Chapter 8, we will describe a model capable of outperforming cardiologist residents in the automatic diagnosis of electrocardiogram exams.

Such a system for the automatic diagnosis of the electrocardiogram, when fully developed, might lead to more reliable automatic diagnosis and improved clinical practice. Although the expert review of complex and borderline cases seems to be necessary even in this future scenario, the development of such automatic interpretation by an algorithm may expand the access of the population to this essential diagnostic exam.

The electrocardiogram exam is often performed in settings, such as in primary care centers and emergency units, where there are no specialists to analyze and interpret the exam. Primary care and emergency department health professionals have limited diagnostic abilities in interpreting this exam (Mant et al., 2007; Veronese et al., 2016). The need for an accurate automatic interpretation is most acute in low and middle-income countries, which are responsible for more than 75% of deaths related to cardiovascular disease (World Health Organization, 2014), and where the population, often, do not have access to cardiologists with full expertise in electrocardiogram diagnosis. Improvements in the automatic diagnosis of other exams might bring similar gains.

The improvement of deep learning in medicine might bring enormous gain for improving clinical practice by both allowing doctors to take more well-informed decisions with the help of artificial intelligence systems and also by eliminating time-consuming tasks from doctor schedule, allowing them to focus on doctor-patient relation (Topol, 2019). The introduction of artificial intelligence in medicine does not offer a replacement of the physician, but rather a way of improving the patient treatment. As nicely put by the neurosurgeon Antonio Di Leva: “Machines will not replace physicians, but physicians using AI will soon replace those not using it”.

Is attention all you need?

When presenting the system we developed for the automatic diagnosis of the electrocardiogram, one question would often appear: “why were we not using a recurrent neural network?”. And in fact, we have tried to employ such architecture, but without nearly as much success as we got for the convolutional architecture.

Recurrent neural networks have been the textbook solution for modeling sequences for a very long time. Since the introduction of a neural network architecture called long-short term memory, in 1997, these models gained a lot of popularity and have been successfully used to solve several sequence tasks, including machine translation (Sutskever et al., 2014), language modeling (Peters et al., 2018) and speech recognition (Graves et al., 2013), to cite a few. Having been the *de facto* state-of-the-art architecture to solve sequence learning tasks until very recently, these models, however, are now being challenged by new, high-performance, feedforward architectures.

In 2017, a paper called “Attention is All You Need” (Vaswani et al., 2017), introducing a feed-forward architecture for language translation. The so-called *transformer*, presented in this very influential paper, would then become a fundamental component for understanding and generating language. By the

moment this thesis was written, transformer-based architectures dominated the leaderboards, achieving top performance in natural language processing and understanding tasks, leaving recurrent architectures in the second plane for those tasks.

Convolutional architectures for sequences are also gradually occupying a place previously dominated by recurrent models (Bai et al., 2018b) and have matched or even outperformed the recurrent architectures performance in language and music modelling (Bai et al., 2018b; van den Oord et al., 2016; Dauphin et al., 2017), text-to-speech conversion (van den Oord et al., 2016), machine translation (Kalchbrenner et al., 2016; Gehring et al., 2017) and other sequential tasks (Bai et al., 2018b).

The role of recurrence

The role of recurrence and recurrent architectures in learning sequences, signals, and systems is a central question for machine learning, system identification, and other related fields. And the same dichotomy between recurrent and feedforward models, take slightly different forms in twin areas of the literature and have been extensively discussed.

More than three decades ago, Narendra and Parthasarathy (1990) proposed the dichotomy between parallel and series-parallel training modes. The first training mode uses recurrence during training, and the second, a feedforward setup. The paper defends training the neural network in the feedforward, or “series-parallel”, setup. Chapter 4 discusses the arguments presented by them in detail and show that they are not precise. This recommendation of using “series-parallel” training mode, however, was carried over to Matlab neural network toolbox ¹, which made the choice quite popular among engineers until very recently. In the system identification literature, the dichotomy between free-run simulation minimization and one-step-ahead minimization takes a different name, but refer to the same problem. And many papers in this area show strengths of minimizing free-run simulation, which also results in a recurrent model during training (Piroddi and Spinelli, 2003; Aguirre et al., 2010; Farina and Piroddi, 2012).

Indeed, the recent success of feedforward architectures (i.e., convolutional and attention-based architectures) in machine learning, was preceded by a wave of interest in recurrent models, and in 2015, a leading researcher Karpathy (2015) wrote a blog post named “The Unreasonable Effectiveness of Recurrent Neural Networks”, defending that recurrent models would “*become a pervasive and critical component to intelligent systems*”.

Outline of this work

This work investigates how nonlinear differential models can be used to model sequence, signals, and systems. The problem is central both for machine learning and systems identification. A fundamental question, studied from more than one angle in this work, is the role of recurrence and recurrent architecture in modeling sequences, and how much recurrence is needed, be it during the training or inference. Finally, we present applications modeling signals and systems. These applications could have a high impact on society by improving our capability to model nonlinear systems in engineering and to extract information from non-stationary signals relevant to health care.

The work is divided into three parts. Part I gives the background needed to understand the subsequent sections. It starts by presenting machine learning fundamentals in Chapter 1. Then, in Chapter 2, it presents common machine learning tasks related to sequences, i.e. *sequence learning*, focusing on neural network architectures that can be used to solve such tasks. Finally, Chapter 3 presents system

¹ The new deep learning toolbox does not contain such a suggestion. But up to 2018, it could be found in Matlab documentation.

identification methods from a nonlinear perspective and highlights connections between nonlinear system identification and sequence learning.

Part II presents the theoretical and methodological contributions of this work to nonlinear system identification and neural networks. We start, in Chapter 4, by studying the effect of recurrence in training shallow neural networks. We show how recurrence in neural networks can make the neural network more robust to some types of noise contamination and improve asymptotic properties. We also study computational aspects and simple examples of neural networks for modeling nonlinear systems. In Chapter 5, we present the downside of recurrence, from a system identification point of view, showing how it might negatively affect the optimization procedure. We offer a solution based on a method known as multiple shooting, which avoids propagating effects for too long. In Chapter 6, we extend the result from the previous chapter, on the impact of recurrence in the cost function, from a system identification perspective into a deep recurrent neural network perspective. The results we found for nonlinear system identification extend a fundamental principle deep and recurrent neural networks: that of exploding gradients. We also study the expressiveness of recurrent neural networks in terms of the dynamic attractors they can represent. Orthogonal and stable recurrent neural networks, which have been recently proposed, are studied in this setup.

Part III studies two applications. In Chapter 7, we study the application of convolutional neural networks in nonlinear system identification problems. We focus on openly available nonlinear system identification benchmarks: modeling an F-16 aircraft from its ground vibration test and modeling the *silverbox* electronic oscillatory circuit. It also includes the performance comparison between convolutional and recurrent models. In Chapter 8, we present the automatic diagnosis of the electrocardiogram. Showing how a convolutional neural network can outperform medical cardiology residents.

Publications related to this thesis

This thesis features content from the following publications:

Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness, *Antônio H. Ribeiro, Koen Tiels, Luis A. Aguirre and Thomas B. Schön*. To appear in the Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS), 2020.

Automatic Diagnosis of the 12-Lead ECG using a Deep Neural Network, *Antônio H. Ribeiro, Manoel Horta Ribeiro, Gabriela M.M. Paixão, Derick M. Oliveira, Paulo R. Gomes, Jéssica A. Canazart, Milton P. S. Ferreira, Carl R. Andersson, Peter W. Macfarlane, Wagner Meira Jr., Thomas B. Schön, Antonio Luiz P. Ribeiro*. Nature Communications, 2020. v. 11(1) n. 1760. doi: 10.1038/s41467-020-15432-4

On the Smoothness of Nonlinear System Identification, *Antônio H. Ribeiro, Koen Tiels, Jack Umenberger, Thomas B. Schön, Luis A. Aguirre*. Provisionally accepted for publication in Automatica, 2020.

Deep Convolutional Networks in System Identification, *Carl Andersson*, Antonio H. Ribeiro*, Koen Tiels, Niklas Wahlström and Thomas B. Schön (* Equal contribution)*. Proceedings of the 58th IEEE Conference on Decision and Control (CDC), 2019. pp. 3670–3676. doi: 10.1109/CDC40024.2019.9030219

“Parallel Training Considered Harmful?” : Comparing Series-Parallel and Parallel Feedforward Network Training, *Antônio H. Ribeiro, Luis A. Aguirre*. Neurocomputing, 2018. v. 316 (17) pp. 222–231. doi: 10.1016/j.neucom.2018.07.071

Shooting Methods for Parameter Estimation of Output Error Models, *Antonio H. Ribeiro, L.A. Aguirre*. Proceedings of the IFAC World Congress, 2017. IFAC-PapersOnLine v. 50 (1), pp. 13998-14003. doi: 10.1016/j.ifacol.2017.08.2421

To a lesser extent, the thesis was also influenced by and contains some ideas from other publications I co-authored during my Ph.D.:

SciPy 1.0—Fundamental Algorithms for Scientific Computing in Python, *Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J van der Walt, Matthew Brett, Joshua Wilson, K Jarrod Millman, Nikolay Mayorov, Andrew R.J. Nelson, Eric Jones, Robert Kern, Eric Larson, C.J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, EA Quintero, Charles R Harris, Anne M. Archibald, **Antônio H. Ribeiro**, Fabian Pedregosa, Paul van Mulbregt, SciPy 1.0 Contributors*. Nature Methods, 2020. v. 17 (3) pp. 261-272. doi: 10.1038/s41592-019-0686-2

Evaluation of mortality in bundle branch block patients from an electronic cohort: Clinical Outcomes in Digital Electrocardiography (CODE) study, *Gabriela M.M. Paixão, Emilly M. Lima, Paulo R. Gomes, Milton P. F. Ferreira, Derick M. Oliveira, Manoel H. Ribeiro, **Antônio H. Ribeiro**, Jamil S. Nascimento, Jéssica A. Canazart, Leonardo B. Ribeiro, Antonio L. P. Ribeiro*. Journal of Electrocardiology, 2019. v. 57 pp. S56-S60. doi: 10.1016/j.jelectrocard.2019.09.004

Tele-electrocardiography and Bigdata: The CODE (Clinical Outcomes in Digital Electrocardiography) Study, *Antonio Luiz P. Ribeiro, Gabriela M.M. Paixão, Paulo R. Gomes, Manoel Horta Ribeiro, **Antônio H. Ribeiro**, Jéssica A. Canazart, Derick M. Oliveira, Milton P. Ferreira, Emilly M. Lima, Jermana Lopes de Moraes, Nathalia Castro, Leonardo B. Ribeiro, Peter W. Macfarlane*. Journal of Electrocardiology, 2019. v. 57 pp. S75-S78. doi: 10.1016/j.jelectrocard.2019.09.008

Lasso Regularization Paths for NARMAX Models via Coordinate Descent, *Antônio H. Ribeiro, Luis A. Aguirre*. Proceedings of the American Control Conference, 2018. pp. 5268-5273. doi: 10.23919/ACC.2018.8430924

Other appearances

The paper *Automatic Diagnosis of the 12-Lead ECG using a Deep Neural Network* was presented at the SciLifeLab Science Summit in Uppsala, Sweden (2019), where it received the best poster award. And at the Swedish Symposium on Deep Learning - SSDL (2019), also as a poster. It was preceded by the pre-print:

Automatic Diagnosis of Short-Duration 12-Lead ECG using a Deep Convolutional Network *Antônio H. Ribeiro, Manoel Horta Ribeiro, Gabriela Paixão, Derick Oliveira, Paulo R. Gomes, Jéssica A. Canazart, Milton Pifano, Wagner Meira Jr., Thomas B. Schön, Antonio Luiz Ribeiro*. 2018. <https://arxiv.org/abs/1811.12194>

This pre-print was presented as a poster in the peer-reviewed non-archival workshop “Machine Learning for Health (ML4H)” at NeurIPS 2018, where it received a travel award. Earlier versions were also presented in the Brazilian Congress of Cardiology in 2018 and in the Brazilian Congress of Informatics in Health in 2018.

The paper *Deep Convolutional Networks in System Identification* was presented, as a poster, at the *European Research Network on System Identification, ERNSI*, (2019) and at the *Nonlinear System Identification Benchmarks Workshop* (2019), as an oral presentation under the name *Deep Convolutional Networks are useful in System Identification*. The paper *On the Smoothness of Nonlinear System Identification* was also presented as a poster at the *European Research Network on System Identification, ERNSI* (2019).

Besides that, the optimization algorithm described in Appendix A was implemented in Python as part of my Google Summer of Code project and integrated to the SciPy library². Scipy is one of the core scientific libraries in Python and is openly available under a permissive free-software license ([Virtanen et al., 2020](#)).

² `scipy.optimize.minimize(..., method='trust-constr')`

Part I

Background

1 Machine learning fundamentals

Machine learning is a branch of artificial intelligence that studies methods to allow computers to learn tasks by examples. For these methods, the task is solved not by pre-determined instruction, but rather, by exposure to examples. There is a significant intersection between machine learning and statistics, and many machine learning methods have been influenced by existing statistical methods. Nevertheless, machine learning approaches are usually more performance-driven and inclined toward black-box approaches. Statistical modeling, on the other hand, often assumes the (stochastic) data generating process is known and builds methods on top of the underlying assumptions on how the data was generated (Breiman, 2001). The distinction between two fields, however, is blurry. Another twin area is *system identification*, which studies modeling in the context of dynamical systems from a system theory background (cf. Chapter 3). In the end, the difference between these fields comes down to the terminology, perspective, and background of the researchers involved.

The focus of this thesis will be on *supervised learning*. Supervised learning is a branch of machine learning that studies methods for modeling the relation between a set of *inputs* (also known as *predictors* or *independent variables*) and *outputs* (also known as *responses* or *dependent variables*). The algorithm learns this relation by optimizing the distance between predicted and observed outputs in a *training set* (for which both the inputs and outputs are known). One example is the problem that will be studied in Chapter 8, for which we have as input the points of an electrocardiogram signal and as output indicators of possible abnormalities.

Two relevant branches of machine learning that will not be explored in this thesis are the so-called *unsupervised learning* and *reinforcement learning*. For unsupervised learning algorithms, there is not a clear distinction between inputs and outputs, and the goal is to find associations and patterns on the available dataset. Reinforcement learning, on the other hand, does not learn patterns but rather actions to take. The actions are not pre-specified, and the method must discover the actions that yield the largest reward by trying them out (Sutton and Barto, 2011). Unlike supervised and unsupervised learning, these methods learn the task, not by being presented to a dataset, but rather, by being presented to a problem with well-defined rules with a reward assigned to each possible outcome and multiple trials to solve it.

In this chapter, we present a short introduction to *supervised learning*. It is not our goal to cover supervised learning from a broad perspective, and we will restrict our presentation to a somewhat limited setup, we refer the reader to (Friedman et al., 2001; Bishop, 2006; Murphy, 2012) for a more extensive treatment of the subject.

1.1 Parameter estimation

Given a *training dataset* containing input-output pairs $(\mathbf{u}_i, \mathbf{y}_i)$ for $i = 1, 2, \dots, N$ and a parameterized model, dependent on the parameter vector $\boldsymbol{\theta}$, relating the inputs and outputs. We estimate the parameters using the following setup: we define a cost function $V(\boldsymbol{\theta}) \in \mathbb{R}$ that provides a measure of how well the model fits the data. Point estimation of the parameter $\boldsymbol{\theta}$ is obtained by optimizing the cost function in an exact or approximate manner. Some basic frameworks for defining this cost function are described next.

1.1.1 Maximum likelihood

The maximum likelihood estimation finds the parameter vector that maximizes the *likelihood* $\mathcal{L}(\boldsymbol{\theta})$ of the observations:

$$\mathcal{L}(\boldsymbol{\theta}) = p_{\boldsymbol{\theta}}(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N \mid \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N), \quad (1.1)$$

where $p_{\boldsymbol{\theta}}(\cdot \mid \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N)$ is the joint probability density of the outputs conditioned on the observed inputs. This approach is quite natural, as it finds the parameters that make the observed data most likely to appear within the family of distributions $p_{\boldsymbol{\theta}}(\cdot)$ being considered. The cost function is often defined as:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log \mathcal{L}(\boldsymbol{\theta}), \quad (1.2)$$

which can be minimized in order to find the parameter vector that maximizes the likelihood.

The likelihood might be decomposed into a chain of conditional distributions:

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{i=1}^N p_{\boldsymbol{\theta}}(\mathbf{y}_i \mid \mathbf{y}_1, \dots, \mathbf{y}_{i-1}, \mathbf{u}_1, \dots, \mathbf{u}_N). \quad (1.3)$$

This decomposition is not unique and alternative orders would also be possible. One special case occurs when the output samples are independent and identically distributed (i.i.d.). In this case:

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{i=1}^N p_{\boldsymbol{\theta}}(\mathbf{y}_i \mid \mathbf{u}_i). \quad (1.4)$$

Hence, in both cases:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{i=1}^N \log p_{\boldsymbol{\theta}}(\mathbf{y}_i \mid \mathbf{x}_i), \quad (1.5)$$

where \mathbf{x}_i is either equal to \mathbf{u}_i or to $(\mathbf{y}_1, \dots, \mathbf{y}_{i-1}, \mathbf{u}_1, \dots, \mathbf{u}_N)$ depending on which case we are dealing with.

Since very often we choose $p_{\boldsymbol{\theta}}(\cdot)$ to belong to an exponential family of distributions, defining the cost function in terms of the the logarithm of the likelihood (*log-likelihood*), as we did in (1.2), often simplifies the computations significantly. Exponential families are sets of distributions $p_{\boldsymbol{\theta}}(\mathbf{z})$ that can be written as:

$$p_{\boldsymbol{\theta}}(\mathbf{z}) = \frac{1}{Z(\boldsymbol{\theta})} A'(\mathbf{z}) \exp(\boldsymbol{\eta}(\boldsymbol{\theta})^T \mathbf{T}(\mathbf{z})), \quad (1.6)$$

for $\boldsymbol{\theta}$ in the convex set Θ . Where $\mathbf{T}(\mathbf{z})$ are called *sufficient statistics*. A set of statistics are sufficient for a family of probability distributions if the data from which it is calculated gives no additional information than does the statistic. And $Z(\boldsymbol{\theta})$ is called *partition function* and it is the normalization constant that guarantees that the distribution $p_{\boldsymbol{\theta}}(\mathbf{z})$ sums to one. Examples of exponential families include the normal, exponential, multinomial with fixed number of trials, gamma, chi-squared, beta, Dirichlet, Bernoulli and Poisson family of distributions.

Let $\mathbf{z} = (\mathbf{y}_i, \mathbf{x}_i)$, we can just write the conditional distribution as:

$$p_{\boldsymbol{\theta}}(\mathbf{y}_i \mid \mathbf{x}_i) = \frac{1}{Z(\boldsymbol{\theta})} A(\mathbf{z}) \exp(\boldsymbol{\eta}(\boldsymbol{\theta})^T \mathbf{T}(\mathbf{z})) \quad (1.7)$$

where: $A(\mathbf{z}) = A'(\mathbf{z})/p_{\boldsymbol{\theta}}(\mathbf{x}_i)$.

In this case, the presence of the logarithm allows the the cost function (1.2) to be written as:

$$V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N (\log(Z(\boldsymbol{\theta})) - \boldsymbol{\eta}(\boldsymbol{\theta})^T \mathbf{T}(\mathbf{z})), \quad (1.8)$$

where we omitted the term $\log A(\mathbf{z})$, since it does not alter the maximum of $V(\boldsymbol{\theta})$.

Next, we present the two very relevant examples of parameter estimation in the exponential family, namely the estimation of parameters of normal and multinomial distributions. For simplicity, we will not put these examples in the canonical form presented above and just work with the most convenient form.

1.1.2 Normal distribution and the least-squares cost function

Assume that we have a normal distribution with mean $\boldsymbol{\mu}_i = \mathbf{f}(\mathbf{x}_i; \boldsymbol{\beta})$ and variance σ^2 .

$$p_{\boldsymbol{\theta}}(\mathbf{y}_i | \mathbf{x}_i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2} \|\mathbf{y}_i - \boldsymbol{\mu}_i\|^2\right). \quad (1.9)$$

where \mathbf{f} is some linear or nonlinear parametrized function and n is the dimension of \mathbf{y} . The maximum likelihood estimation of $\boldsymbol{\theta} = [\boldsymbol{\beta}, \sigma^2]$ is given by the minimum of:

$$V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \left(\frac{1}{2\sigma^2} \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i; \boldsymbol{\beta})\|^2 + \frac{n}{2} \log(\sigma^2) \right) = \frac{1}{2\sigma^2} V'(\boldsymbol{\beta}) + \frac{n}{2} \log(\sigma^2), \quad (1.10)$$

again, we omitted constant additive terms. The minimizer of $V(\boldsymbol{\theta})$ will be $\hat{\boldsymbol{\theta}} = [\hat{\boldsymbol{\beta}}, \hat{\sigma}^2]$. Where the optimal value $\hat{\boldsymbol{\beta}}$ does not depend on the variance and can be computed minimizing the *least-square problem*:

$$V'(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^N \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i; \boldsymbol{\beta})\|^2. \quad (1.11)$$

And the variance can be computed as the minimum of $\frac{1}{\sigma^2} V'(\hat{\boldsymbol{\beta}}) + n \log(\sigma^2)$ which occurs at: $\hat{\sigma}^2 = \frac{1}{n} V'(\hat{\boldsymbol{\beta}}) = \frac{1}{nN} \sum_{i=1}^N \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i; \hat{\boldsymbol{\beta}})\|^2$.

1.1.3 Multinomial distribution and the softmax layer

Assume the output is a scalar categorical variable $y_i \in \{0, 1, \dots, k\}$ where the probability of the outcomes $1, 2, \dots, k$ is given by the elements of $\mathbf{p}_i = [p_i^1, p_i^2, \dots, p_i^k]$. Be $I_{j,y}$ the indicator function, defined as:

$$I_{j,y} = \begin{cases} 1 & j = y; \\ 0 & j \neq y. \end{cases} \quad (1.12)$$

We can define the probability of the output assuming a given value y_i by:

$$p(y_i) = (p_i^1)^{I_{1,y_i}} (p_i^2)^{I_{2,y_i}} \dots (p_i^k)^{I_{k,y_i}}. \quad (1.13)$$

Now assume we will define $\mathbf{p}_i = \mathbf{p}(\mathbf{x}_i; \boldsymbol{\theta})$. Where $\mathbf{p}(\bullet; \boldsymbol{\theta}) : \mathbb{R}^{N_x} \rightarrow \mathbb{S}$ is a parametrized function that maps into the set $\mathbb{S} = \{\mathbf{p} \in \mathbb{R}^k \mid \sum_{j=1}^k p^j = 1 \text{ and } 0 \leq p^j \leq 1 \text{ for all } j\}$. The restrictions that define the set \mathbb{S} are needed so \mathbf{p} can be interpreted as a vector of probabilities. The maximum likelihood estimation of $\boldsymbol{\theta}$ is given by the minimum of:

$$V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N (I_{1,y_i} \log(p^1(\mathbf{x}_i; \boldsymbol{\theta})) + I_{2,y_i} \log(p^2(\mathbf{x}_i; \boldsymbol{\theta})) + \dots + I_{k,y_i} \log(p^k(\mathbf{x}_i; \boldsymbol{\theta}))). \quad (1.14)$$

We define $\mathbf{p}(\mathbf{x}; \boldsymbol{\theta})$ as the composition of two functions: the 1-1 map $\mathbf{s} : \mathbb{R}^{k-1} \rightarrow \mathbb{S}$ and the parametrized function $\mathbf{f}(\bullet; \boldsymbol{\theta}) : \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{k-1}$. Such that $\mathbf{p}(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{s}(\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}))$ has domain \mathbb{S} . By far the most common

option for the map $\mathbf{s}(\bullet)$ is:

$$\begin{aligned} s^1(\mathbf{z}) &= \frac{\exp(z^1)}{1 + \sum_{i=1}^{k-1} \exp(z^i)} \\ s^2(\mathbf{z}) &= \frac{\exp(z^2)}{1 + \sum_{i=1}^{k-1} \exp(z^i)} \\ &\vdots \\ s^{k-1}(\mathbf{z}) &= \frac{\exp(z^{k-1})}{1 + \sum_{i=1}^{k-1} \exp(z^i)} \\ s^k(\mathbf{z}) &= \frac{1}{1 + \sum_{i=1}^{k-1} \exp(z^i)}. \end{aligned}$$

Putting this expression of \mathbf{p} back into (1.1.3), we obtain:

$$V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \left(I_{1,y_i} f^1(\mathbf{x}_i; \boldsymbol{\theta}) + I_{2,y_i} f^2(\mathbf{x}_i; \boldsymbol{\theta}) + \cdots + I_{k-1,y_i} f^{k-1}(\mathbf{x}_i; \boldsymbol{\theta}) - \log \left(1 + \sum_{j=1}^{k-1} \exp(f^j(\mathbf{x}_i; \boldsymbol{\theta})) \right) \right),$$

which can be minimized in order to obtain the maximum likelihood estimation of $\boldsymbol{\theta}$.

The map \mathbf{s} , which we have defined above, is related to the so-called *softmax layer*, which is often used as a neural network layer. More precisely, we have used $\mathbf{s}(\mathbf{z}) = \mathbf{S}([\mathbf{z}, 0])$ where we have set the last component of \mathbf{S} equal to zero to guarantee that $\mathbf{s}(\mathbf{z})$ sums to 1.

Definition 1.1 (Softmax). *The softmax layer is a map $\mathbf{S} : \mathbb{R}^k \rightarrow \{\mathbf{p} \in \mathbb{R}^k \mid 0 \leq p^j \leq 1, \forall j\}$. For which:*

$$S^j(\mathbf{z}) = \frac{\exp(z^j)}{1 + \sum_{l=1}^k \exp(z^l)}. \quad (1.15)$$

□

1.1.4 Bayes parameter estimation and maximum-a-posteriori

An alternative view on the parameter estimation problem is to consider that there is no true parameter and, rather, that $\boldsymbol{\theta}$ is a random variable. The idea is that there is a *prior distribution* $p(\boldsymbol{\theta})$, which summarize our knowledge of this parameter $\boldsymbol{\theta}$ before any observation. How to decide this prior is a widely studied problem. Some approaches try to use uninformative priors, that convey as little information as possible. Other approaches try to reflect a preference for parsimonious and “simpler” solutions.

Assume that we have the training set $\mathbb{Z} = \{(\mathbf{u}_i, \mathbf{y}_i) \text{ for } i = 1, 2, \dots, N\}$ and that the parameter $\boldsymbol{\theta}$ is independent of the input \mathbf{u}_i for every i , we can compute the probability of $\boldsymbol{\theta}$ given the observed data using the Bayes rule:

$$p(\boldsymbol{\theta} \mid \mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N, \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N) = \frac{p(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N \mid \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N; \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N \mid \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N)}, \quad (1.16)$$

where $p(\boldsymbol{\theta} \mid \mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N, \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N)$ is called *posterior* distribution and the *likelihood* is denoted by $p(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N \mid \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N; \boldsymbol{\theta})$ (using a notation slightly different from Section 1.1.1). Here, the *posterior* distribution gives the updated belief about the parameters after taking into account the observed data.

In a fully Bayesian approach, we use this posterior distribution in all predictions, which might be challenging computationally. Even though there are methods for doing this efficiently, we will not study them here. Instead, we will give a brief description of a method called *maximum-a-posteriori*,

which computes a point estimation $\hat{\boldsymbol{\theta}}$ by maximizing the posterior distribution. Which is equivalent to minimizing:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log p(\boldsymbol{\theta} \mid \mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N, \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_N). \quad (1.17)$$

Using (1.16) and omitting additive constants that do not depend on $\boldsymbol{\theta}$ the above cost function simplifies to:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log \mathcal{L}(\boldsymbol{\theta}) - \frac{1}{N} \log p(\boldsymbol{\theta}), \quad (1.18)$$

which is the combination of the cost function in the maximum likelihood plus an penalty term depending on the prior. Next we study the effect of two priors on the maximum a posteriori estimation: namely a Gaussian prior and a Laplacian prior.

1.1.5 Prior and parameter penalty

Assume that the prior distribution on $\boldsymbol{\theta}$ is a distribution centered at zero with the following form:

$$p(\boldsymbol{\theta}) = K \exp(-\lambda(\|\boldsymbol{\theta}\|_q)^q). \quad (1.19)$$

where K is some normalizing constant that does not depend on $\boldsymbol{\theta}$ and $\|\boldsymbol{\theta}\|_q = (\sum_i |\theta^i|^q)^{1/q}$ is the q -norm of $\boldsymbol{\theta}$. For this prior, Eq. (1.18) simplifies to:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log \mathcal{L}(\boldsymbol{\theta}) + \frac{\lambda}{N} (\|\boldsymbol{\theta}\|_q)^q. \quad (1.20)$$

For instance if $q = 2$, $p(\boldsymbol{\theta})$ is distributed according to a normal distribution, and $V(\boldsymbol{\theta})$ is equal to:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log \mathcal{L}(\boldsymbol{\theta}) + \frac{\lambda}{N} \|\boldsymbol{\theta}\|^2. \quad (1.21)$$

Which is basically the minimization of the log-likelihood with an additional l_2 parameter penalty. In the context of linear least-square problems this penalty is called *Tikhonov regularization* or *ridge regression*. And, in the context of neural networks, it is often called *weight decay*.

Another popular choice is to use $q = 1$, which yields a Laplacian prior distribution and $V(\boldsymbol{\theta})$ equals to:

$$V(\boldsymbol{\theta}) = -\frac{1}{N} \log \mathcal{L}(\boldsymbol{\theta}) + \frac{\lambda}{N} \|\boldsymbol{\theta}\|_1. \quad (1.22)$$

Unlike l_2 penalization, this formulation produces sparse solutions due to the non-differentiability of the function being minimized. That is, some of the coefficients will be *exactly* zero, which may: i) enhance the model interpretability; ii) yield cheaper computational evaluation of the model, since many of the coefficients will be zero.

For l_1 regularization the number of non-zero terms depends on the value of λ . The larger the value of λ , the fewer degrees of freedom are allowed to the solution. And again, it becomes possible to tune the model complexity by adjusting the value of λ . In the context of linear models, this l_1 regularization is often called *Lasso regression*¹ (Tibshirani, 1996).

1.2 Model complexity

A central problem in machine learning is how to obtain a model that does perform well not just in the training set but also generalize well to new inputs. A model that has too much flexibility may *overfit* the training set, by adjusting itself to the random effects from the dataset it was trained; on the other hand, a model that is not sufficiently flexible may fail to capture essential system characteristics, which is known as *underfitting*.

¹ Lasso stands for *Least absolute shrinkage and selection operator*.

1.2.1 Bias-variance tradeoff

When creating a model, it is impossible to account for all possible factors that might interfere in the outcome. For instance, when building a model to predict the orbit of the earth around the sun, the sun and all planets in the solar systems might be taken into account, but it is unfeasible to account for the gravitational attraction of all bodies in the universe. It is not very productive either because these often have a very small effect on what the model is trying to predict.

The approach used in statistics and machine learning to take into account the effects that are not being directly modeled is to consider the presence of random variables, described by a distribution of values rather than a single deterministic value. Due to the randomness, there is a model prediction error that will inevitably arise, even if for the best conceivable model. There are two other error components. The first is due to the structural inflexibility of the model class being considered. That is, within a model class, even when the parameters are set to their optimal values, the model might still not be the best possible choice. In this case, a broader class of models might be able to capture the underlying behavior better. The second factor that causes errors is the model learning random effects present in the training data. These two components are some times called *bias error* and *variance error*.

The twin goal of reducing these two errors has a conflicting nature. To reduce the *bias error*, one should increase the model flexibility, making it possible to better approximate the system. However, this usually causes the *variance error* to increase since when more flexibility is given to the model, it becomes more subject to learning random effects from the training data. The problem of finding a tradeoff between these conflicting goals is called *bias-variance tradeoff*.

Under appropriated assumptions, the model error can be decomposed as additive components. Assume that we have an output \mathbf{Y} and an input \mathbf{X} , both vectorial random variables. And that we have used some method to estimate a function $f(\bullet)$, which tries to capture the relation between input and output. Let us define the expected prediction error (EPE) as:

$$\text{EPE} = E_{X,Y} \{ \|\mathbf{Y} - f(\mathbf{X})\|^2 \}, \quad (1.23)$$

where $E_{X,Y}\{\bullet\}$ is the expectation regarding the joint distribution (\mathbf{X}, \mathbf{Y}) .

$$\begin{aligned} & E_{X,Y} \{ \|\mathbf{Y} - f(\mathbf{X})\|^2 \} \\ &= E_{X,Y} \left\{ \left\| (\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\}) - (f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}) \right\|^2 \right\} \\ &= E_{X,Y} \left\{ \|\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\}\|^2 + \|f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}\|^2 - 2(\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\})^T (f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}) \right\} \\ &= E_{X,Y} \{ \|\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\}\|^2 \} + E_X \{ \|f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}\|^2 \} - 2E_{X,Y} \{ (\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\})^T (f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}) \}. \end{aligned}$$

The third term of this decomposition $E_{X,Y} \{ (\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\})^T (f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}) \}$ can be expanded as:

$$\begin{aligned} & E_X E_{Y|X} \{ \mathbf{Y}^T f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}^T f(\mathbf{X}) - \mathbf{Y}^T E_{Y|X}\{\mathbf{Y}\} + E_{Y|X}\{\mathbf{Y}\}^T E_{Y|X}\{\mathbf{Y}\} \} \\ &= E_X \{ E_{Y|X}\{\mathbf{Y}\}^T f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}^T f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}^T E_{Y|X}\{\mathbf{Y}\} + E_{Y|X}\{\mathbf{Y}\}^T E_{Y|X}\{\mathbf{Y}\} \} \\ &= 0. \end{aligned}$$

Hence, the expected prediction error simplifies to:

$$\text{EPE} = E_X \{ E_{Y|X} \{ \|\mathbf{Y} - E_{Y|X}\{\mathbf{Y}\}\|^2 \} + \|f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}\|^2 \}. \quad (1.24)$$

Notice that the first term is independent of f , and, hence, the value of f for which the expected prediction error is minimum occur for the minimizer of:

$$E_X \{ \|f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}\|^2 \}.$$

Let us define a function $f^*(\mathbf{x}) = \arg \min_c \|c - E_{Y|X}\{\mathbf{Y}\}\|^2 = E_{Y|X}\{\mathbf{Y}\}$. This function minimizes the prediction error pointwise, and, hence, it will minimize $E_X \{\|f(\mathbf{X}) - E_{Y|X}\{\mathbf{Y}\}\|^2\}$ and the expected prediction error. Using this definition of optimal f^* we can rewrite (1.24) as:

$$\text{EPE} = E_{X,Y} \{\|\mathbf{Y} - f^*(\mathbf{X})\|^2\} + E_X \{\|f(\mathbf{X}) - f^*(\mathbf{X})\|^2\}. \quad (1.25)$$

Using a procedure entirely analogous to the one we have used to arrive at this expression we can show that:

$$E_X \{\|f(\mathbf{X}) - f^*(\mathbf{X})\|^2\} = E_X \{\|f(\mathbf{X}) - E_X\{f(\mathbf{X})\}\|^2\} + \|f^*(\mathbf{X}) - E_X\{f(\mathbf{X})\}\|^2.$$

Hence, the expected prediction error can be written as:

$$\text{EPE} = \underbrace{E_{X,Y} \{\|\mathbf{Y} - f^*(\mathbf{X})\|^2\}}_{\text{irreducible error}} + \underbrace{E_X \{\|f(\mathbf{X}) - E_X\{f(\mathbf{X})\}\|^2\}}_{\text{variance}} + \underbrace{\|f^*(\mathbf{X}) - E_X\{f(\mathbf{X})\}\|^2}_{\text{bias}}. \quad (1.26)$$

Hence, for this formulation, the error can be decomposed as the three component mentioned earlier in this section: the unavoidable error that arise for any model f due to the randomness of the data; the variance of the model due to random effects in the training dataset being captured by the model and the distance between the current and the best possible model that could be used.

1.2.2 Regularization methods and hyperparameter choice

Techniques used for controlling the model complexity to make it generalize well for new inputs are collectively known as *regularization methods*. One commonly used technique is to include an additive penalty term that penalizes the norm of the parameter vector:

$$V(\boldsymbol{\theta}) = V_U(\boldsymbol{\theta}) + \frac{\lambda}{N} \|\boldsymbol{\theta}\|_p^p, \quad (1.27)$$

where V_U is a cost function measuring the model performance in the training set. And λ can be use to control the complexity of the model. A large value of λ indicates that complicated models (with large parameters) will be strongly penalized and the optimization of V will yield simpler models. Such that, as $\lambda \rightarrow \infty$ the estimated parameter vector tends to zero and as $\lambda \rightarrow 0$ the solution will approach the minimizer of V_U .

In Section 1.1.5 we have arrived at a cost function very similar to that of (1.27) using a maximum-a-posteriori formulation. Indeed, prior parameter distributions can be used to infuse knowledge about the desired model in the estimation. And the choice of priors centered around zero, which decay as we move away from the center, serves the purpose of specifying that we would prefer to have small parameters (which usually correspond to simpler models).

An appropriate choice of the factor λ will yield a model that not only optimizes the performance in the training set but also that gives a good generalization to unseen situations, by optimizing the performance at the same time it penalizes model complexity. We should highlight that any choice of $\lambda \neq 0$ will result in a model that does not optimize only the performance and, hence, that will have worse accuracy **on the training set** than the solution of V_U . That is because evaluating the model performance on the training set only allows us to assess the bias error (i.e., the inflexibility of the model class to capture the input-output relation). However, the variance error, which appears due to the model capturing spurious random effects, will not reflect on the training set performance. Hence, when choosing the regularization parameter λ , we want to find the value that optimizes both components of the error: the variance and the bias error.

Hence, one cannot choose λ or other parameters that control model complexity by trying to find its optimal value on the training set. In a data-rich situation, one may set aside a *development dataset*

and use it for evaluating the performance of models resulting from different choices, choosing the one with the best performance on this dataset. *Cross-validation* is an alternative that does not require a separate development dataset (Friedman et al., 2001), but require extra computational power, since the dataset will be subdivided into K -folds and the model will be trained multiple times alternating which folds are used as training and which are used to evaluate the performance, the final choice of regularization parameters will be the average of the different realizations, which are used in a final stage, for which the model is trained using the entire set. Alternatives, for linear models are the *Akaike's Information Criterion* (Akaike, 1974) and *Bayesian Information Criteria* (Schwarz, 1978), which provide an optimal model complexity under restrictive assumptions.

In the neural network literature, it is usual to call the parameters that cannot be set by optimizing the training dataset as *hyperparameters*. A separate *development dataset* is, currently, the usual choice for setting these *hyperparameters* for large neural networks since *cross-validation* might be too computationally expensive. *Hyperparameters* include the regularization parameters (such as λ in the above example), model structure choices and also the *optimization parameters*. Optimization parameters might affect the final result significantly when $V(\boldsymbol{\theta})$ is not convex since the optimizer is not guaranteed to find the optimal value.

1.2.3 Model final performance

As discussed in the previous section, assessing the model performance on the same set that was used to estimate its parameters, will not give an accurate estimate of what will be the performance of the model on unseen data. To a lesser extension, evaluating the performance on the development dataset will also give an over-optimistic evaluation of the performance since this dataset was used to choose the *hyperparameters*. Hence, to properly evaluate the model performance, a separate *test set* may be set aside and used to assess the performance on new data.

Hence, in a data-rich situation, one may divide the dataset into three parts: a *training set*, a *development set* and a *test set*. The parameters are estimated to fit the *training set*; the *development set* is used for model and hyperparameter selection; and, the *test set* should be set aside and brought out only for the assessment of the final chosen model, to provide a reliable estimation of the model error.

1.3 Optimization

Here we study some aspects of algorithms for finding the minimum of the cost function $V(\boldsymbol{\theta})$, $V : \mathbb{R}^{N_{\boldsymbol{\theta}}} \rightarrow \mathbb{R}$.

1.3.1 Global vs local optimization algorithms

We make the distinction between *local solutions* and *global solutions* of V . A *global solution* satisfies:

$$V(\boldsymbol{\theta}^*) \leq V(\boldsymbol{\theta}), \quad (1.28)$$

for all $\boldsymbol{\theta}$. A *local solution*, on the other hand, only needs to satisfy the above inequality in a ball with radius r near the solution, that is it satisfies the above inequality for all values inside the set:

$$\{\boldsymbol{\theta} : \|\boldsymbol{\theta}^* - \boldsymbol{\theta}\| < r\}, \quad (1.29)$$

for some value of r . Hence, all global solutions are also local solutions, but not all local solutions are global solutions.

Local optimization algorithms are those that we considered having converged once a local solution is found. No matter if this solution is also a global solution or not. *Global optimization algorithms* are those that might not have their convergence criteria satisfied even after founding a local solution. It is usually hard to establish if a global solution has been indeed found in a high dimensional space, though, so they typically converge to approximate solutions.

Populational algorithms is a class of global optimization algorithms. This class of methods instantiate a population of solutions θ_i , $i = 1, 2, 3, \dots, n$. At each iteration, it computes the value of the cost function at those points and uses the information to propose a new set of individuals θ_i . *Genetic algorithms* are probably the most famous algorithms within this class [Goldberg \(2000\)](#). Another related approach is *Bayesian optimization* [Mokus \(1974\)](#), which sets a prior over the objective function value and update this prior as the cost function is actually evaluated in points along the parameter space. Both approaches try to explore the parameter space, while also intensifying around regions where good solutions are located.

Local algorithms start with an initial solution guess θ_0 and update the solution iteratively obtaining a sequence of solutions $\{\theta_n\}$ that should approach a local solution. At iteration n_* , the algorithm is considered to have converged when θ_{n_*} is sufficiently close to a local solution. When V is diferentiable at θ_{n_*} , this criteria can be easily verified by checking if the gradient $\nabla V(\theta_{n_*})$ is sufficiently close to zero, i.e. $\|\nabla V(\theta)_{n_*}\| < 10^{-8}$.

1.3.2 Taylor approximations

If V is continuous differentiable, we have that:

$$V(\theta + D) = V(\theta) + \nabla V(\theta)^T D + \mathcal{O}(\|D\|^2), \quad (1.30)$$

further more if V is twice continuous differentiable:

$$V(\theta + D) = V(\theta) + \nabla V(\theta)^T D + \frac{1}{2} D^T \nabla^2 V(\theta) D + \mathcal{O}(\|D\|^3). \quad (1.31)$$

Where $\mathcal{O}(\|D\|^n)$ is a term that is proportional to the norm of D to the power of n . Hence, if $\|D\|$ is sufficiently small, we can ignore the term $\mathcal{O}(\|D\|^2)$ and approximate the cost function by its *first-order approximation*:

$$V(\theta + D) \approx V(\theta) + \nabla V(\theta)^T D. \quad (1.32)$$

Or, more accurately, by its *second-order approximation*:

$$V(\theta + D) \approx V(\theta) + \nabla V(\theta)^T D + \frac{1}{2} D^T \nabla^2 V(\theta) D, \quad (1.33)$$

where $\nabla V(\theta)$ is the gradient and $\nabla^2 V(\theta)$ the Hessian of V evaluated at θ .

Methods that use a first-order Taylor approximation to compute the step direction are called *first-order methods*. *Second-order methods* are those that use the second-order Taylor approximations in order to compute the step direction. Computing the Hessian might be too expensive or demand too much memory, and some algorithms use an approximation of the Hessian in the computation the step direction. We will call this class of methods *inexact second-order methods*.

The *steepest descent* algorithm is a *first-order method*. This algorithm update the solution using the following rule:

$$\theta_{n+1} = \theta_n - \eta \nabla V(\theta_n), \quad (1.34)$$

where η is called the *step size* or *learning rate*. This iterative rule follows from simply considering, at each iteration, a first-order approximation (1.32) around θ_n and choosing an update D which minimizes $V(\theta_n) + \nabla V(\theta_n)^T D$ subject to $\|D\| = \eta \|\nabla V(\theta_n)\|$.

The *Newton method* algorithm is a *second-order method*. It use the update rule:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - (\nabla^2 V(\boldsymbol{\theta}_n))^{-1} \nabla V(\boldsymbol{\theta}_n), \quad (1.35)$$

which follows simply from considering, at each iteration, a second-order approximation (1.33) around $\boldsymbol{\theta}_n$ and choosing an update \mathbf{D} which minimizes $V(\boldsymbol{\theta}_n) + \nabla V(\boldsymbol{\theta}_n)^T \mathbf{D} + \frac{1}{2} \mathbf{D}^T \nabla^2 V(\boldsymbol{\theta}_n) \mathbf{D}$.

Finally, *quasi-Newton methods* are *inexact second-order methods*. They use an approximation $B_n \approx (\nabla^2 V(\boldsymbol{\theta}_n))^{-1}$ and uses the following update rule:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - B_n \nabla V(\boldsymbol{\theta}_n). \quad (1.36)$$

1.3.3 Convexity

The function V is said to be *convex* if it satisfies the following inequality for every $\boldsymbol{\theta}$, $\boldsymbol{\phi}$, and for every $\alpha \in [0, 1]$:

$$V(\alpha \boldsymbol{\theta} + (1 - \alpha) \boldsymbol{\phi}) \leq \alpha V(\boldsymbol{\theta}) + (1 - \alpha) V(\boldsymbol{\phi}). \quad (1.37)$$

The function is said to be *strictly convex* if the equality holds only for $\alpha = 0$ or $\alpha = 1$.

The next theorem gives one important property, which immensely simplifies the optimization of convex functions:

Theorem 1.1. *When V is convex, any local minimizer of V is a global minimizer.*

Proof. See Nocedal and Wright (2006, Theorem 2.5). □

Furthermore, for convex functions for which ∇V is Lipschitz continuous² with constant L , the steepest descent method with fixed step size, given by the update rule (1.34), always converge to the solution if the step size respect the following inequality: $\eta < 2/L$ (cf. Poljak (1987)). Furthermore, for convex functions for which $\nabla^2 V$ is Lipschitz continuous, the Newton method (1.35) is also guaranteed to converge to a solution (Nocedal and Wright, 2006, Theorem 3.5).

1.3.4 Line-search and trust-region methods

For convex and smooth cost functions, the previous section gives conditions to the update rules (1.34) and (1.35) converge to a solution. When working with non-convex functions, however, these simple update rules are not guaranteed to converge to a local solution. There are two strategies that can guarantee the convergence to a local solution in the non-convex case, namely *line-search methods* and *trust-region methods*.

Line-search methods, at each interaction, first choose a direction \mathbf{D}_n and then, in a second stage, choose a step size $\eta_{(n)}$. This direction is $\mathbf{D}_n = -\nabla V(\boldsymbol{\theta}_n)$ or $\mathbf{D}_n = -\nabla^2 V(\boldsymbol{\theta}_n) \nabla V(\boldsymbol{\theta}_n)$ for first and second-order methods, respectively. The step size $\eta_{(n)} > 0$ is chosen in a way that *approximately* minimizes $V(\boldsymbol{\theta}_n + \eta \mathbf{D}_n)$. If the step size is choosen in such way that satisfy some basic conditions (such as the Wolf conditions) the algorithm is guaranteed to converge to a local soluton for first and second-order methods. (Nocedal and Wright, 2006)

Trust-region methods, at each iteration, construct a local approximation $M_n(\boldsymbol{\theta})$ whose behaviour near the current point $\boldsymbol{\theta}_n$ is close to that of the function V . The first-order approximation (1.32) or the second-order approximation (1.33) around the current iterate $\boldsymbol{\theta}_n$ are both examples of local models that can be used. The next iterate $\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \mathbf{D}_n$ is computed as the minimizer of this local approximation.

² ∇V is Lipschitz continuous with constant L if and only if there exist a constant L such that $\|\nabla V(\boldsymbol{\theta}) - \nabla V(\boldsymbol{\phi})\| < L \|\boldsymbol{\theta} - \boldsymbol{\phi}\|$

This local approximation, however, might not be a good approximation far from θ_n , hence we restrict the search for minimizers within some region \mathcal{F}_n around θ_n in which we trust our local model. Hence the update will be computed by, approximately, solving:

$$\min_{\mathbf{D}} M_n(\theta_n + \mathbf{D}) \text{ subject to } (\theta_n + \mathbf{D}) \in \mathcal{F}_n. \quad (1.38)$$

If the candidate solution does not provide a sufficient decrease, we conclude that the trust-region is too large, and we shrink it and re-solve (1.38). It is very common to define the trust-region as a ball centered in θ_n , that is: $\mathcal{F}_n = \{(\theta_n + \mathbf{D}) : \|\mathbf{D}\| < \Delta_n\}$, for some value of Δ_n . Elliptical and box-shaped trust-regions are also used in some algorithms.

We refer the reader to Nocedal and Wright (2006) for an extensive treatment of both line-search and trust-region methods.

1.3.5 Convergence rate

Be $\{\theta_n\}$ a sequence that converges to a point θ^* . The convergence is:

- *linear* if, there exist $0 < r < 1$ and N , such that, for all $n > N$:

$$\frac{\|\theta_{n+1} - \theta^*\|}{\|\theta_n - \theta^*\|} \leq r. \quad (1.39)$$

- *quadratic* if, there exist $0 < r < 1$ and N , such that, for all $n > N$:

$$\frac{\|\theta_{n+1} - \theta^*\|}{\|\theta_n - \theta^*\|^2} \leq r. \quad (1.40)$$

- *superlinear* if for all values of r we can always find N , such that, for all $n > N$:³

$$\frac{\|\theta_{n+1} - \theta^*\|}{\|\theta_n - \theta^*\|^2} \leq r. \quad (1.42)$$

We give examples of sequences that converge linearly, superlinearly, and quadratically in Figure 1.1. Any sequence that converges quadratically also converges superlinearly. And, in turn, every superlinear convergent sequence also converges linearly.

Consider a sequence of solutions $\{\theta_n\}$ produced by a first-order, second-order, and inexact second-order optimization methods. For a large class of methods, the convergence rate of these algorithms is, respectively, linear, quadratic, and superlinear. We refer the reader to (Nocedal and Wright, 2006) for a complete description of the algorithms and conditions for this to hold.

The faster convergence rate is one of the strongest advantages of second-order methods over inexact second-order methods. And, in turn, of inexact second-order methods over linear methods. It is important to keep in mind, however, that this is an asymptotic convergence rate, and the number of iteration before achieving this convergence rate is not specified. The asymptotic speed of convergence also depends on r , whose values depend not only on the algorithm but also on the properties of the optimization problem being solved. This dependency is weaker for quadratically convergent sequences. And, while a quadratically convergent sequence will, eventually, always converge faster than a linear, this might take many iterations.

³ That is equivalent to:

$$\lim_{n \rightarrow \infty} \frac{\|\theta_{n+1} - \theta^*\|}{\|\theta_n - \theta^*\|^2} = 0. \quad (1.41)$$

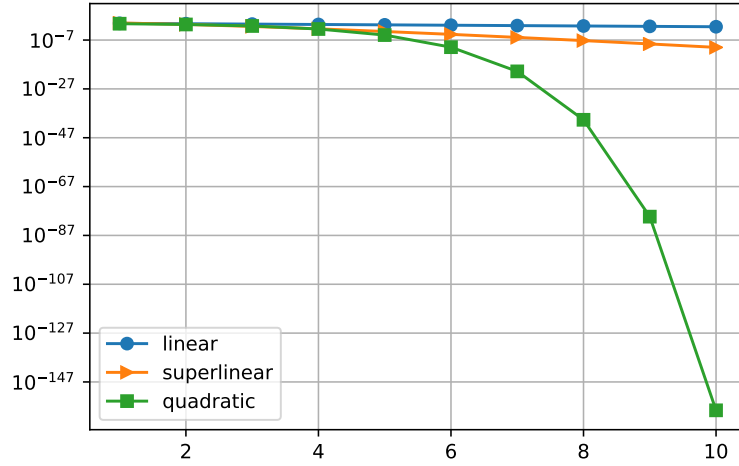


Figure 1.1 – **(Convergence rate)** We show the first 10 points of sequences that converge toward zero. The sequence $(0.7)^n$ converges linearly, n^{-n} converges superlinearly and, $(0.7)^{2^n}$ converges quadratically.

1.3.6 Scaling

Figure 1.2 illustrates different cost functions with the same final solution. We also show the first iterations of the steepest descent algorithm. The performance of the algorithm varies a lot. The difference in performance is due to problem formulation and *scaling*. A problem is said to be *poorly scaled* if a change in one direction produces a much larger variation in the cost function than changes in another direction.

Scale invariance is the property of an optimization method not being affected by scaling. It is a much-desired property that allows the algorithm to be robust and not loose performance, even when faced with a problem containing effects happening at different scales. While the steepest descent algorithm is greatly affected by poor scaling, the Newton method is less affected by it, since information about curvature is being captured by the second-order derivatives. Inexact second-order methods usually fair better than the first-order methods in this regard as well.

1.3.7 Memory complexity

At each iteration, first-order methods only need to store the gradient while second-order methods also have to store the Hessian. Hence, be N_θ the number of parameters, while first-order methods require an storage of $\mathcal{O}(N_\theta)$, second-order methods have a requirement of $\mathcal{O}(N_\theta^2)$. This quadratic dependence on the number of parameters might make the use of second-order methods unfeasible for many applications. In Chapters 4 and 5 we use an inexact second-order method that is based on the nonlinear least-square structure of the problem, and approximate the Hessian by $\mathbf{J}\mathbf{J}^T$, where $\mathbf{J} \in \mathbb{R}^{N_\theta \times N_\theta}$ is the Jacobian matrix of the error and hence has a memory requirement $\mathcal{O}(N * N_\theta)$, where N_θ is the number of parameters, and N is the number of data points.

Particularly, the quadratic memory complexity of exactly second-order methods or even the complexity of $\mathcal{O}(N * N_\theta)$ from nonlinear least-square methods is not consistent with the needs of deep learning applications, where it is very usual to use models with a very large number of parameters (For instance, the high-performance deep-learning-based chatbot Meena has 2.6 billion parameter (Adiwardana et al., 2020)). There are inexact second-order methods that fare much better in this aspect, though. For instance, the L-BFGS algorithm approximates the Hessian using the L last values of the gradient and has

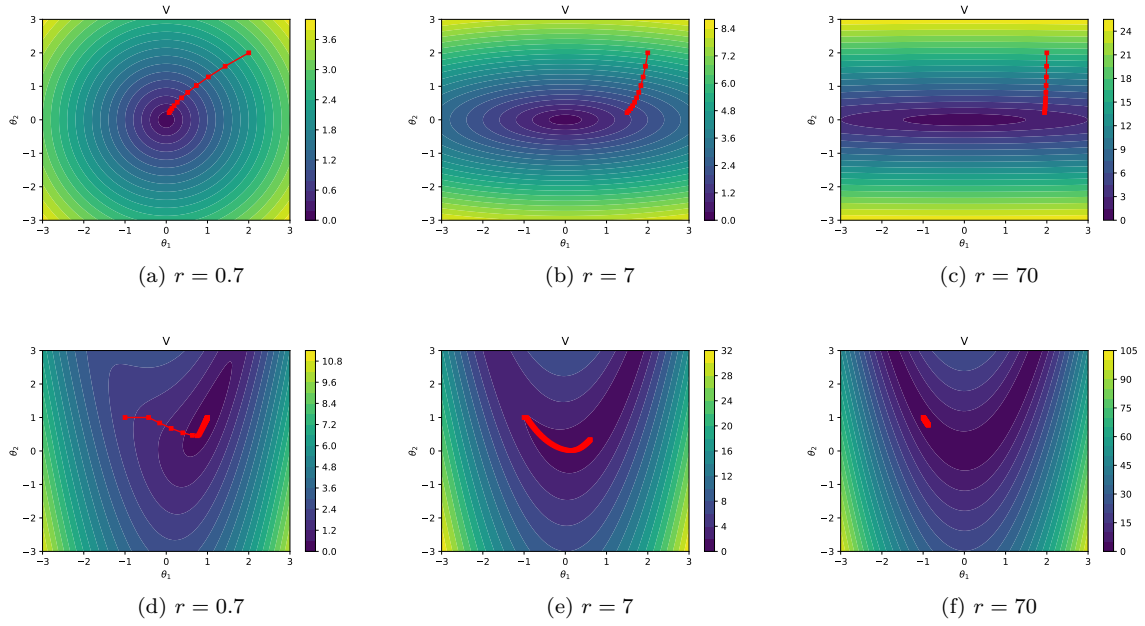


Figure 1.2 – **(Scaling)** We show the contour plot and the first iterations of steepest descent algorithm with fixed step size (in red). Figures (a) to (c) display the contour plot for the cost function $V(\theta_1, \theta_2) = \theta_1^2 + r\theta_2^2$, which has a global minimum at $(0, 0)$. We start the steepest descent algorithm from the initial point $(2, 2)$ and show the first 10 iterations. Figures (d) to (f) display the contour plot for the cost function $V(\theta_1, \theta_2) = (1 - \theta_1)^2 + r(\theta_2 - \theta_1^2)^2$, which has a global minimum at $(1, 1)$. The function is known as the *Rosenbrock function*. We start the steepest descent algorithm from the initial point $(-1, 1)$ and show the first 100 iterations. In both cases, r is used to adjust the *scaling* of the problem, with large values of r corresponding to ill-scaled problems. The fixed step size of the optimization algorithm is fixed to $0.1/r$ to avoid divergence.

a memory requirement of $\mathcal{O}(LN_\theta)$, and might yield good results even for small L (Liu and Nocedal, 1989).

1.3.8 Stochastic optimization

Let the cost function be defined by:

$$V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{k=1}^N l_k(\boldsymbol{\theta}), \quad (1.43)$$

where l_k is the *loss function*. We assume we have a training dataset containing N samples and that l_k is related to the k -th entry. See Section 1.1.1 to see one examples where such cost function might arise. It is also possible to easily incorporate a *penalty term* $\Omega(\boldsymbol{\theta})$, for regularization, but we omit such term in the discussion for simplicity.

The steepest descent algorithm with fixed step size, i.e. (1.34), applied to this cost function could be implemented as the following sequence of operations:

ALGORITHM 1.1 (BATCH STEEPEST DESCENT). Fix the step-size η . Repeat the following, till $\|\nabla V(\boldsymbol{\theta})\| < 1 \times 10^{-8}$:

1. For $k = 1$ to N compute: $\mathbf{d}_k \leftarrow \nabla l_k(\boldsymbol{\theta}_n)$;
2. Compute: $\mathbf{D} \leftarrow \frac{1}{N} \sum_{k=1}^N \mathbf{d}_k$;
3. Update: $\boldsymbol{\theta}_{n+1} \leftarrow \boldsymbol{\theta}_n - \eta \mathbf{D}$;

4. Update: $n \leftarrow n + 1$.

□

The major drawback of that algorithmic approach is that it requires us to compute the gradient of all entries (i.e. compute $\nabla l_k(\boldsymbol{\theta}_n)$ for $k = 1, \dots, N$) before updating the parameter. This might be inefficient for large datasets: Few samples might already give a good update direction for a small fraction of the computational cost and an approach called *stochastic gradient descent* is a more efficient alternative, as demonstrated in (Bottou et al., 2018). This approach computes the derivative of a *mini-batch* containing n_b samples and use it to compute the update direction, doing that for all the samples N of the dataset, n_b samples at a time. The method is summarized in Algorithm 1.2. Another advantage of using small mini-batches is that the stochastic nature of the algorithm might help escaping *sharp local minima* and yield solutions that generalize better in unseen datasets (Keskar et al., 2016).

ALGORITHM 1.2 (STOCHASTIC GRADIENT DESCENT). Fix the step-size η . Repeat the following, till $\|\nabla V(\boldsymbol{\theta})\| < 1 \times 10^{-8}$:

1. Shuffle data samples;
2. for $b = 1$ to $\left\lceil \frac{N}{n_b} \right\rceil$:
 - a) For $k = 1$ to n_b compute: $\mathbf{d}_k \leftarrow \nabla l_{(k+b*n_b)}(\boldsymbol{\theta}_n)$;
 - b) Compute: $\mathbf{D} \leftarrow \frac{1}{n_b} \sum_{k=1}^{n_b} \mathbf{d}_k$;
 - c) Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{D}$;
3. Update: $n \leftarrow n + 1$.

□

Stochastic exact and inexact second-order approaches also exist and process the input one mini-batch at a time. Taking into account both the cost per iteration and convergence rate, stochastic gradient descent provides a better time-complexity than, stochastic and batch, exact second-order methods (Bousquet and Bottou, 2008). Inexact stochastic second-order methods present an interesting alternative that is still being studied (Bottou et al., 2018), but have not gained much traction in deep learning applications yet despite the advantages mentioned in Section 1.3.5 and 1.3.6. Reasons range from the popularity of function that are piecewise differentiable such as rectified linear unit (ReLU) and might affect the performance of second-order methods negatively, to the fact that first-order algorithms are simpler to implement and understand. To the moment this thesis was written, stochastic gradient descent and its variations with momentum are still the most popular choice in deep learning applications.

1.4 Automatic differentiation

As discussed above, many optimization methods require us to compute derivatives. Fortunately, there exist algorithmic approaches that allow the exact computation of the derivatives of an arbitrary function. These algorithmic approaches are extremely useful when estimating parameters and optimizing a cost functions V and high-quality libraries implementing such methods were developed to fulfill the needs of deep learning implementations and are continuously evolving, such as the libraries PyTorch (Paszke et al., 2017) and Tensorflow (Abadi et al., 2015). Another focus of such libraries is to provide easily available interfaces to graphical processing units (GPUs) to speed up the computation.

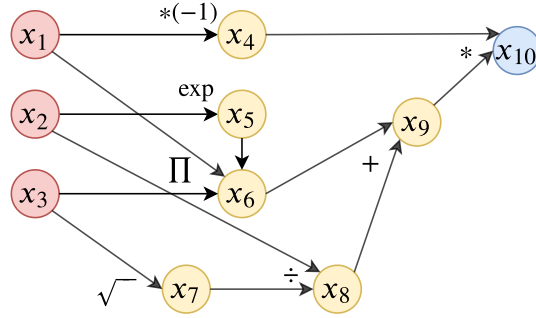


Figure 1.3 – **(Computational graph)** Graph containing the partially ordered elementary operations involved in the computation of f , defined as in (1.44). The *leaf*s of the computational graphs are the nodes in red and correspond to the function inputs.

Let \mathbf{f} be an arbitrary function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the computation of this function can be broken down into elementary operations. For instance, given the function:

$$f(x_1, x_2, x_3) = -x_1 * \left(x_1 * e^{x_2} * x_3 + \frac{\sqrt{x_3}}{x_2} \right), \quad (1.44)$$

we can define intermediary variables containing results from the computations and break down the entire function evaluation into elementary operations, as follows:

$$x_4 = -x_1 \quad (1.45a)$$

$$x_5 = e^{x_2} \quad (1.45b)$$

$$x_6 = x_1 * x_5 * x_3 \quad (1.45c)$$

$$x_7 = \sqrt{x_3} \quad (1.45d)$$

$$x_8 = x_7 / x_2 \quad (1.45e)$$

$$x_9 = x_6 + x_8 \quad (1.45f)$$

$$x_{10} = x_4 * x_9. \quad (1.45g)$$

And $f = x_{10}$. This sequence of partially ordered operations can be put together as a *computational graph*. Any dependency $x_i \rightarrow x_j$ indicated in the graph corresponds to the dependencies in the computation. Let us call x_j a *parent node*, the *child nodes* are all the nodes that point to it in the graph (i.e. all the elements that are required to compute this intermediary elements). The computational graph for the example is given in Figure 1.3. The computational graph consists of a *directed acyclic graph* and a list of elementary operations. The elementary operations are well defined enough so, given all the child nodes, we can compute the value for the parent node, that is be $\mathbf{x}_{C(j)}$ a vector of child nodes, then $x_j = o_j(\mathbf{x}_{C(j)})$.

A *topological order* is an ordering for the nodes in the graph such that if $x_i \rightarrow x_j$ then $j > i$. Notice that the nodes $\{x_1, x_2, \dots, x_{10}\}$ in Figure 1.3 and in Equations (1.46) are indexed according to a topological ordering.

Given a computational graph two algorithmic approaches can be used for computing the derivatives: *forward-mode* and *reverse-mode automatic differentiation*. Mixed approaches are also possible. For all approaches, all the operations must have well defined derivatives, so be a pair of child and parent node $x_i \rightarrow x_j$. Then $\partial x_j / \partial x_i$ can always be computed given the set of values from the child nodes. For instance, in the example, $x_6 = x_1 * x_2 * x_3$. Hence, given that $(x_1, x_2, x_3) = (3, 6, 12.5)$, we can compute the partial

derivatives:

$$\frac{\partial x_6}{\partial x_1} = x_2 * x_3 = 6 * 12.5 = 75 \quad (1.46a)$$

$$\frac{\partial x_6}{\partial x_2} = x_1 * x_3 = 3 * 12.5 = 37.5 \quad (1.46b)$$

$$\frac{\partial x_6}{\partial x_3} = x_1 * x_2 = 3 * 6 = 18. \quad (1.46c)$$

1.4.1 Forward-mode automatic differentiation

Let x_j be any node in the computational graph, let us call the gradient of this intermediary element, with regard to the inputs $\mathbf{x} = (x_1, \dots, x_n)$, by:

$$\nabla x_j(\mathbf{x}) = \left(\frac{\partial x_j}{\partial x_1}(\mathbf{x}), \dots, \frac{\partial x_j}{\partial x_n}(\mathbf{x}) \right). \quad (1.47)$$

Forward-mode automatic differentiation associate with each node the value-gradient pair $(x_j, \nabla x_j(\mathbf{x}))$. This value-gradient pair will be evaluated and carry forward in the computational graph.

ALGORITHM 1.3 (FORWARD-MODE AUTOMATIC DIFFERENTIATION). Order the nodes in topological order $\{x_1, x_2, \dots\}$. Be n the number of inputs. For $j = 1, 2, \dots$:

1. If $j \leq n$, x_j is one of the leafs (input of the function), its value is known and the gradient is given by:

$$\nabla x_j(\mathbf{x}) = \mathbf{e}_j = (0, \dots, 0, 1, 0, \dots, 0); \quad (1.48)$$

2. If $j > n$, be $C(j)$ the set of child nodes of j , and $\mathbf{x}_{C(j)} = (x_i : i \in C(j))$ a vector containing them. The value of this node can be computed using the elementary operation $x_j = o_j(\mathbf{x}_{C(j)})$ and the gradient can be using the chain rule:

$$\nabla x_j(\mathbf{x}) = \sum_{i \in C(j)} \frac{\partial x_j}{\partial x_i}(\mathbf{x}_{C(j)}) \nabla x_i(\mathbf{x}). \quad (1.49)$$

□

Notice that since the nodes are indexed in topological order, the values required to carry the computation, i.e. 1) the values of the child nodes $\mathbf{x}_{C(j)}$; and, 2) the gradients $\nabla x_i(\mathbf{x})$ for $i \in C(j)$, will already have been computed once we arrive at the computation of $(x_j, \nabla x_j(\mathbf{x}))$.

1.4.2 Reverse-mode automatic differentiation

Reverse-mode automatic differentiation does not perform function and gradient evaluations concurrently. Instead, it performs a two-stage computation. In a forward pass, it computes the values of all intermediary nodes. In a backward pass, it computes the derivatives.

Let $\mathbf{x} = (x_1, \dots, x_n)$ be the input, in a first pass the algorithm compute the value of all intermediary nodes $x_j = o_j(\mathbf{x}_{C(j)})$. Let $\mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathbb{R}^m$ be the output and x_j any node in the computational graph. Let us call $\frac{\partial \mathbf{y}}{\partial x_j}(\mathbf{x})$ the derivative of the output with regard to this intermediary element. The algorithm proceed as follows

ALGORITHM 1.4 (REVERSE-MODE AUTOMATIC DIFFERENTIATION). Order the nodes in topological order $\{x_1, x_2, \dots\}$. Be m the number of outputs and N the total number of nodes.

1. For $j = 1, 2, \dots, N$:

- a) Compute $x_j \leftarrow o_j(\mathbf{x}_{C(j)});$
- 2. For $i = N, N - 1, \dots, 1:$
 - a) If $N - i \leq m$, x_i is one of outputs of the function, and the partial derivative is given by:

$$\frac{\partial \mathbf{y}}{\partial x_i}(\mathbf{x}) = \mathbf{e}_i = (0, \dots, 0, 1, 0, \dots, 0); \quad (1.50)$$

- b) If $N - i > m$, the partial derivative can be computed using the chain rule:

$$\frac{\partial \mathbf{y}}{\partial x_i}(\mathbf{x}) = \sum_{j \in P(i)} \frac{\partial \mathbf{y}}{\partial x_j}(\mathbf{x}) \frac{\partial x_j}{\partial x_i}(\mathbf{x}). \quad (1.51)$$

where $P(i)$ are all the parent nodes of the node x_i .

□

1.4.3 Computational complexity and applications

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be the function to be evaluated, and $n_e = \sum_j \sum_{i \in C(j)} 1 = \sum_i \sum_{j \in P(i)} 1$ the number of edges in its computational graph, n the number of inputs, and m the number of outputs. Forward-mode automatic differentiation has a computational cost $\mathcal{O}(n_e n)$. Reverse-mode differentiation, on the other hand, has the computational cost $\mathcal{O}(n_e m)$. Hence, while the forward-mode has a computational complexity that scales with the number of inputs, reverse-mode has a computational complexity that scales with the number of outputs.

The main appeal of reverse-mode automatic differentiation when optimizing a scalar cost function $V : \mathbb{R}^{N_\theta} \rightarrow \mathbb{R}$ is its low computational cost. The extra arithmetics associated with the gradient computation is usually no more than 4 or 5 times the arithmetic needed to evaluate the function alone. This approach is very popular in neural network applications and is sometimes called *backpropagation*. The fact that the computational cost does not scale with the number of parameters allows models with a large number of parameters to be estimated.

In nonlinear least-squares problems, the Jacobian matrix of an error function can provide a good approximation of the Hessian of the cost function and speed up the convergence of the optimization algorithm. This type of optimization problem minimizes $\|\mathbf{e}\|^2$, where $\mathbf{e} : \mathbb{R}^{N_\theta} \rightarrow \mathbb{R}^N$. The major advantage is that the computational cost will not depend on the number of datapoints N , and this might yield efficient implementations. The computational cost will still depend on the number of parameters N_θ , which might be a limitation for problems with a very large number of parameters.

In Chapter 4, we use a mixed approach that uses reverse-mode automatic differentiation for computing the derivative of the neural network and a forward mode automatic differentiation for propagating the derivative through the recurrence equation, this enables the efficient computation of the Jacobian matrix of the error vector. In Chapter 5, we used a similar approach for some of the examples. When dealing with many parameters, however, even this approach might become unfeasible, and in the implementation of Chapters 6, 7 and 8, we always use reverse-mode automatic differentiation since we are dealing with large neural networks with many parameters.

	automatic differentiation		optimization method		input processing	
	reverse	forward	1st order	inex. 2nd order	batch	stochastic
Chapter 4		\times^\dagger		Levenberg-Marquardt	\times	
Chapter 5		\times^\ddagger		trust-region method	\times	
Chapter 6	\times		ADAM			\times
Chapter 7	\times		ADAM			\times
Chapter 8	\times		ADAM			\times

Table 1.1 – **(Design choices)** Summary of the design choices in the numerical examples. The Levenberg-Marquardt algorithm is a nonlinear least-squares algorithm proposed by (Marquardt, 1963). The trust-region algorithm from Chapter 5 was described by (Lalee et al., 1998) and ADAM is a stochastic first-order with momentum proposed by (Kingma and Ba, 2014). [†]In Chapter 4 we actually use a mixed approach, where we use backpropagation (which uses reverse mode automatic differentiation) to compute the derivative of the neural network, and forward differentiation to propagate the derivatives through the recurrence. [‡]We did the same for Example 2 in Chapter 5.

2 Neural network models for sequences

Machine learning has had many recent successes, achieving great performance in tasks such as image classification (Krizhevsky et al., 2012), machine translation (Bahdanau et al., 2014; Vaswani et al., 2017) and speech recognition (Hinton et al., 2012), to cite a few. The driving force behind many of these recent achievements is the development of deep neural networks (LeCun et al., 2015): models composed of stacked layers that combine linear and nonlinear transformations. There are great expectations about the development of this technology, which may help to transform health care and clinical practice (Stead, 2018; Naylor C, 2018; Hinton, 2018; Topol, 2019), allow the full development of autonomous vehicles (Geiger et al., 2012) and dramatically improve how computers understand and generate text (Devlin et al., 2018).

Artificial neural network development dates back from the 1940s. Earlier biological inspired models of neurons with weights set by human operators, such as the McCulloch-Pitts neuron (McCulloch and Pitts, 1943), served as an inspiration to models that could learn the weights from inputs to solve certain tasks, such as the perceptron (Rosenblatt, 1958) and the ADALINE (Widrow and Hoff, 1960). These basic units connected in parallel and stacked in layers would render successful models two decades latter (Hinton, 1986; Rumelhart et al., 1988) and also popularize the estimation of these models by stochastic gradient descent, with the gradient of the error computed using reverse-mode automatic differentiation, in a procedure known as *backpropagation* that is still used in many state-of-the-art recent models.

Technical developments made it possible to train models with a larger number of parameters, and the digitalization of our society resulted in extensive records that could be organized in large datasets for training and testing machine learning models. These developments rendered *end-to-end* models successful in the last decade. End-to-end models represent a change of paradigm for modeling text, image, and other types of unstructured data. In this paradigm, raw data is provided as input to the model, in contrast to the more traditional approach for which the data is pre-processed, features are extracted and used as input to the machine learning algorithm. And it is in this end-to-end setup that deep neural networks have been thriving. The impressive results in computer vision (Krizhevsky et al., 2012) and natural language processing (Devlin et al., 2018), to cite a few, have been driving a new wave of interest in neural networks. And the applications of these learning methods have been rebranded as *deep learning*.

In this chapter, we present a short introduction to deep learning and neural networks. Our presentation will be largely focused on models that use sequences as inputs. Several problems with relevant applications fall into this category, and the methods discussed here can be used in a large range of applications, such as processing and automatically analyzing biomedical signals, understanding and generating text, and modeling industrial processes, aircrafts, and the most diverse dynamical systems.

2.1 Sequence models

A sequence is an enumerated collection of objects which may be finite, $\{x_k\}_{k=1}^N$, or infinite, $\{x_k\}_{k=1}^\infty$. Sequence models are models that have sequences as inputs or outputs. In practical models, we will always work with finite sequences, but it might be useful to take into consideration an infinite sequence to establish asymptotic properties of the method being analyzed. Next, we present relevant black-box tasks which require sequence models. The tasks are to:

1. **capture input-output relations.** The model tries to capture the relation between the input sequence $\{x_k\}_{k=1}^N$ and the output sequence $\{y_k\}_{k=1}^N$ given a dataset containing pairs of input-output

sequences of the same length. In this case, there is a one-to-one correspondence between input and output entries and the obtained model must be able to produce a prediction of the output y_k given the input and, in some cases, other observed elements from the output sequence. Doing that for $k = 1, \dots, N$ if required.

2. **capture autoregressive relations.** Here there is a single sequence: $\{y_k\}_{k=1}^N$ and the model tries to predicting the k -th element of the sequence given all the other sequence elements (or a subset of it). Doing that for all values of k . In some sense, this is the same as the previous task, but without an input sequence.
3. **determine sequence-to-sequence transformation.** The model tries to capture the relation between input and output for sequences of different lengths. It uses data to build a model that takes as input the sequence $\{x_k\}_{k=1}^N$ and tries to predict the output sequence $\{y_k\}_{k=1}^M$, where $M \neq N$. There is no direct correspondence between individual sequence elements.
4. **extract information from a sequence.** The model learns to extract information from a sequence. It builds a model that has as input a sequence $\{x_k\}_{k=1}^N$ and as output the element Y containing information about the given sequence.

The elements x_k , y_k and Y above are problem-specific. To cite a few examples, they might be scalars in \mathbb{R} ; vectors in \mathbb{R}^n ; a word in a dictionary of possible words; or, a vector of probabilities that sum to one. **Input-output models** are fairly common in engineering and system identification deals almost exclusively with this type of models (cf. Chapter 3). In engineering, many variables of interest are continuous variables sampled at a fixed rate. In this case, it is common to sample the input and output at the same rate (or resample the variables at a common rate for modeling the process), hence, $x_k = x(kT)$ and $y_k = y(kT)$, where T is the sampling time and $x(t)$ and $y(t)$ are continuous signals at time t . Examples of models of this type of model are given in Chapter 7, where we model the relation between the voltage on two different points of an oscillatory circuit and the relation between the force applied by a shaker and the acceleration at different points of an F-16 aircraft wing during a ground vibration test.

Input-output models can be **causal** or **noncausal**. The model is **causal** if the output y_k depends only on past or current elements and to predict the sequence at time k the model take into consideration only $\{x_i\}_{i=1}^k$ and $\{y_i\}_{i=1}^{k-1}$. Otherwise, we will say the model is **non-causal**. The engineering models mentioned above, for which k correspond to the temporal evolution of the variables, are examples of causal models. This definition can be easily extended to autoregressive models. One example of non-causal autoregressive models is the so-called *language models*. Assume that the sequence $\{y_k\}_{k=1}^N$ correspond to a sentence and y_k is the k -th word of this sentence. The language model tries to predict the probability distribution for the k -th word given the other words in the sentence. Since most languages do not have a structure where cause necessarily precedes the consequence, non-causal models usually yield the best performance in this task. Language models are basic blocks for building algorithms for understanding and generating text.

Another highly relevant task for natural language processing is the so-called *machine translation*. Assume an input sequence $\{x_k\}_{k=1}^N$ representing a phrase in English. The task is to find an adequate sentence $\{y_k\}_{k=1}^M$ in Portuguese, which contains the same meaning as the original phrase. In the above classification, this would be a **sequence transformation** problem. Notice that unlike input-output models, there is no need for a one-to-one correspondence between elements of the input sequence and elements of the output sequence.

Finally, sometimes it is interesting to **extract information from a sequence**. That is, we want to develop a model for which the input is a sequence, and the output is a vector containing relevant information extracted from this sequence. For instance, if $\{x_k\}_{k=1}^N$ corresponds to words of an e-mail, it might be interesting to have as output the probability of this mail is spam. Or, if $\{x_k\}_{k=1}^N$ correspond to words in a comment about a movie (or about a product that is for sale), it might be interest to automatically identify from the text if the comment corresponds to a positive, negative or neutral opinion about the content: a task known as *sentiment analysis*. Another example that we will study directly in this thesis is the problem of automatic diagnosis of the electrocardiogram (cf. Chapter 8). In this case, assume $\{x_k\}_{k=1}^N$ is the signal obtained from the electrocardiogram exam, corresponding to the cardiac electrical activity of a person during a 10 seconds exam. An output of interest is if this exam contains an abnormality that might indicate some heart disease (and which type of abnormality).

2.2 Recurrent neural networks

Consider the input sequence $\{\mathbf{z}_k\}_{k=1}^N$, the recurrent neural network is a model that generates a sequence of *internal states* $\{\mathbf{x}_k\}_{k=1}^N$. It does so by propagating an initial state \mathbf{x}_0 through the recursive relation:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{z}_{k+1}; \boldsymbol{\theta}), \quad (2.1)$$

where $\boldsymbol{\theta} \in \mathbb{R}^{N_\theta}$ is a parameter vector parametrizing the (nonlinear) function \mathbf{f} .

2.2.1 Training and inference using recurrent neural networks

Next, we describe how these internal states propagated by the recurrent neural network are used to solve the most diverse tasks.

Capturing input-output relations: For a given training pair of input-output sequences $\{\mathbf{u}_k\}_{k=1}^N$ and $\{\mathbf{y}_k\}_{k=1}^N$ (both with the same length), we can define the input \mathbf{z}_k to the recurrent neural network either as $\mathbf{z}_k = \mathbf{u}_k$ or as $\mathbf{z}_k = (\mathbf{u}_k, \mathbf{y}_k)$. The internal states can be computed according to (2.1) and the neural network prediction $\hat{\mathbf{y}}_k$ is computed from the internal states according to:

$$\hat{\mathbf{y}}_{k+1} = \mathbf{g}(\mathbf{x}_k; \boldsymbol{\theta}). \quad (2.2)$$

The optimization problem tries to make those predictions close (in some sense) to the observed values by minimizing a cost function V . The cost function can be defined as:

$$V = \frac{1}{N} \sum_{k=1}^N l(\mathbf{y}_k, \hat{\mathbf{y}}_k). \quad (2.3)$$

The parameter vector $\boldsymbol{\theta}$ is estimated (i.e. the neural network is trained) by minimizing V or, in the case of multiple independent training sequences, minimizing a weighted average of many V s defined as in (2.3). Here, l is the loss function. Common choices for regression and classification problems are the squared error and cross entropy loss (both follow from the maximum likelihood setup discussed in Section 1.1.1).

Capturing autoregressive relations: Here the recurrent neural network receives as input previous output values, i.e. $\mathbf{z}_k = \mathbf{y}_{k-1}$. The internal states at time k are then used to produce a prediction $\hat{\mathbf{y}}_k$:

$$\hat{\mathbf{y}}_{k+1} = \mathbf{g}(\mathbf{x}_k; \boldsymbol{\theta}). \quad (2.4)$$

We estimate the parameter vector by defining and minimizing a cost function analogous to (2.3). Again this cost function encodes some notion of proximity between observed and predicted values.

Extracting information from a series: Given training pairs of input sequences $\{\mathbf{u}_k\}_{k=1}^N$ and output information vectors \mathbf{Y} , we define a model that gives a prediction $\hat{\mathbf{Y}}$, either from a combination of all internal states of the neural network $\hat{\mathbf{Y}} = \phi(\mathbf{x}_1, \dots, \mathbf{x}_N; \boldsymbol{\theta})$ or just from the last one $\hat{\mathbf{Y}} = \phi(\mathbf{x}_N; \boldsymbol{\theta})$. Be N_{seq} the number of sequences in the training set, the parameter vector might be estimated (i.e. the neural network trained) by minimizing:

$$V = \frac{1}{N_{\text{seq}}} \sum_{s=1}^{N_{\text{seq}}} l(\mathbf{Y}_s, \hat{\mathbf{Y}}_s). \quad (2.5)$$

Determining sequence-to-sequence transformation. Given training pair of input-output sequences $\{\mathbf{u}_k\}_{k=1}^N$ and $\{\mathbf{y}_k\}_{k=1}^M$ with different lengths there are several ways that have been published in the literature to solve this same problem using recurrent and feedforward neural networks. Here we mention a few interesting examples that came from machine translation.

One of the simplest architectures is the so-called encoder-decoder architecture. The encoder is a recurrent neural network:

$$\mathbf{x}_{k+1} = \mathbf{f}_{\text{enc}}(\mathbf{x}_k, \mathbf{u}_k; \boldsymbol{\theta}), \quad (2.6)$$

which reads the input sequence $\{\mathbf{u}_k\}_{k=1}^N$ and produces one final state $\mathbf{x}_{\text{final}} = \mathbf{x}_{N+1}$. The decoder is another recurrent neural network:

$$\mathbf{h}_{k+1} = \mathbf{f}_{\text{dec}}(\mathbf{h}_k, \mathbf{y}_k; \boldsymbol{\theta}), \quad (2.7a)$$

$$\hat{\mathbf{y}}_k = \mathbf{g}_{\text{dec}}(\mathbf{h}_k; \boldsymbol{\theta}), \quad (2.7b)$$

which receives the final state from the other recurrent neural network, $\mathbf{x}_{\text{final}}$, and uses it as initial state. Producing predictions to the other sequence $\hat{\mathbf{y}}_k$. This architecture is used, for instance by (Cho et al., 2014b; Sutskever et al., 2014).

2.2.2 A short note about attention-based models

In the encoder-decoder architecture from (Cho et al., 2014b; Sutskever et al., 2014), the entire input sentence is condensed to a single state vector and fed into the decoder. Later, a mechanism called *attention* was introduced in order to improve this (Bahdanau et al., 2014; Luong et al., 2015). The basic idea behind this mechanism is that *all* internal states of the encoder $\{\mathbf{x}_k\}_{k=1}^N$ are used together with the decoder state \mathbf{h}_{k-1} to produce a context vector \mathbf{c}_k that is used as input to the next decoder step. This attention mechanism helps with the performance of this model for long sentences.

Interestingly, this attention mechanism was so successful that later it was shown that an encoder-decoder architecture entirely based on attention (without any recurrent component) could achieve state-of-art-results in machine translation (Vaswani et al., 2017). These impressive results would generate a lot of interest in attention-based models, and the basic attention unit proposed in (Vaswani et al., 2017) would later become a basic building block for many language processing tasks (Devlin et al., 2018).

These attention-based models have been receiving a lot of attention from the research community lately and achieving impressive results in several natural language understanding and generation tasks. Nevertheless, we will limit ourselves to this short note about this class of model and focus on recurrent and convolutional models.

2.2.3 Common recurrent neural networks architectures

Next, we give the example of three recurrent neural network architectures that are popular among deep learning practitioners.

1. The simplest one is the *Elman* recurrent neural network. Which (for the single layered case) has the propagation formula:

$$\mathbf{x}_{k+1} = \tanh(W_{xx}\mathbf{x}_k + W_{zx}\mathbf{z}_{k+1} + b), \quad (2.8)$$

and have as trainable parameters the concatenation of weights and bias: $\theta = (W_{ss}, W_{hh}, b)$.

2. Arguably the most popular architecture at the moment is the *long-short term memory* (LSTM) recurrent neural network (Hochreiter and Schmidhuber, 1997). The internal state consist of two components: $\mathbf{x}_k = (\mathbf{h}_k, \mathbf{c}_k)$, these two components, together with the input are then used to compute the following intermediary values:

$$\mathbf{i}_{k+1} = \sigma(W_{zi}\mathbf{z}_{k+1} + W_{hi}\mathbf{h}_k + \mathbf{b}_i); \quad (2.9a)$$

$$\mathbf{f}_{k+1} = \sigma(W_{zf}\mathbf{z}_{k+1} + W_{hf}\mathbf{h}_k + \mathbf{b}_f); \quad (2.9b)$$

$$\mathbf{g}_{k+1} = \tanh(W_{zg}\mathbf{z}_{k+1} + W_{hg}\mathbf{h}_k + \mathbf{b}_g); \quad (2.9c)$$

$$\mathbf{o}_{k+1} = \sigma(W_{zo}\mathbf{z}_{k+1} + W_{ho}\mathbf{h}_k + \mathbf{b}_o), \quad (2.9d)$$

which are used to compute \mathbf{h} and \mathbf{c} at the next instant:

$$\mathbf{c}_{k+1} = \mathbf{f}_{k+1} \odot \mathbf{c}_k + \mathbf{i}_{k+1} \odot \mathbf{g}_{k+1}; \quad (2.10a)$$

$$\mathbf{h}_{k+1} = \mathbf{o}_{k+1} \odot \tanh(\mathbf{c}_{k+1}), \quad (2.10b)$$

where σ is the sigmoid function, \odot operator represents the elementwise product between the vectors and the trainable parameters are: $\theta = (W_{zi}, W_{hi}, W_{zf}, W_{hf}, W_{zg}, W_{hg}, W_{zo}, W_{ho}, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_g, \mathbf{b}_o)$.

3. Another architecture that has become popular lately, is the *gated recurrent unit* (GRU) (Chung et al., 2014; Cho et al., 2014b), which, at instant k , computes intermediary values:

$$\mathbf{r}_{k+1} = \sigma(W_{zr}\mathbf{z}_{k+1} + W_{xr}\mathbf{x}_k + \mathbf{b}_r); \quad (2.11a)$$

$$\mathbf{f}_{k+1} = \sigma(W_{zf}\mathbf{z}_{k+1} + W_{xf}\mathbf{x}_k + \mathbf{b}_f); \quad (2.11b)$$

$$\mathbf{n}_{k+1} = \tanh(W_{zn}\mathbf{z}_{k+1} + W_{xn}\mathbf{x}_k + \mathbf{b}_n), \quad (2.11c)$$

and uses them to compute the state at the next instant $k + 1$:

$$\mathbf{x}_{k+1} = (1 - \mathbf{f}_{k+1}) \odot \mathbf{n}_{k+1} + \mathbf{z}_{k+1} \odot \mathbf{x}_k. \quad (2.12)$$

This architecture has as trainable parameters the concatenation of weights and bias: $\theta = (W_{zr}, W_{xr}, W_{zf}, W_{xf}, W_{zn}, W_{xn}, \mathbf{b}_r, \mathbf{b}_f, \mathbf{b}_n)$.

2.2.4 Bidirectional neural networks

The recurrent neural network defined in (2.1) - and in the architectures defined above - has an intrinsic causal structure. That is, the output sequence at time k depends only on past values of the input sequence. For problems where a non-causal structure is needed, that is, we want the internal state at time k to depend both on past, and future inputs, a commonly used architecture is a bidirectional recurrent neural network:

$$\mathbf{x}_{k+1}^{(f)} = \mathbf{f}^{(f)}(\mathbf{x}_k^{(f)}, \mathbf{z}_k; \theta); \quad (2.13a)$$

$$\mathbf{x}_{k-1}^{(b)} = \mathbf{f}^{(b)}(\mathbf{x}_k^{(b)}, \mathbf{z}_k; \theta), \quad (2.13b)$$

which propagates the set of states $\mathbf{x}_k^{(f)}$ in the *forward* direction (i.e. $1, 2, \dots, N - 1, N$) and the set of states $\mathbf{x}_k^{(b)}$ in the *backward direction* (i.e. $N, N - 1, \dots, 2, 1$). These two states are then concatenated: $\mathbf{x}_k = (\mathbf{x}_k^{(f)}, \mathbf{x}_k^{(b)})$.

2.2.5 Multi-layer recurrent neural network

It is common to define multi-layer recurrent neural networks. This multi-layer model uses stacked recurrent layers: The input $\{\mathbf{z}\}_{k=1}^N$ is fed to the first layer, which generates a sequence. This sequence produced by the 1-st layer $\{\mathbf{x}^{(1)}\}_{k=1}^N$ is the input to the second layer. Which, in turn, produces a new sequence $\{\mathbf{x}^{(2)}\}_{k=1}^N$ which is fed into the next layer, and so on. For unidirectional layers, this would result in the following propagation formula:

$$\mathbf{x}_{k+1}^{(1)} = \mathbf{f}^{(1)}(\mathbf{x}_k^{(1)}, \mathbf{z}_{k+1}; \boldsymbol{\theta}^{(1)}); \quad (2.14a)$$

$$\mathbf{x}_{k+1}^{(2)} = \mathbf{f}^{(2)}(\mathbf{x}_k^{(2)}, \mathbf{x}_k^{(1)}; \boldsymbol{\theta}^{(2)}); \quad (2.14b)$$

$$\vdots$$

$$\mathbf{x}_{k+1}^{(L)} = \mathbf{f}^{(L)}(\mathbf{x}_k^{(L)}, \mathbf{x}_k^{(L-1)}; \boldsymbol{\theta}^{(L)}), \quad (2.14c)$$

where L is the total number of layers.

The multi-layer model presented above is actually a special case of the more general model presented by the recursive relation (2.1). The multi-layer model can be put into the more general format by defining $\mathbf{x}_k = (\mathbf{x}_k^{(1)}, \mathbf{x}_k^{(2)}, \dots, \mathbf{x}_k^{(L)})$ and $\boldsymbol{\theta} = (\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \dots, \boldsymbol{\theta}^{(L)})$.

It is not uncommon to use the same architectures and sizes for all layers: $\mathbf{f}^{(1)} = \mathbf{f}^{(2)} = \dots = \mathbf{f}^{(L)}$. The architectures described in Section 2.2.3 are common choices for each layer. LSTM architecture, usually does not propagate the full state $\mathbf{x}_k = (\mathbf{h}_k, \mathbf{c}_k)$, but rather just \mathbf{h}_k to the next layer. And for non-causal task a bidirectional structure (cf. Section 2.2.4) might be used in each layer.

2.3 Convolutional neural networks for sequences

Convolutional neural networks (Lecun et al., 1998) had a very strong impact on computer vision, achieving state-of-the-art-results in tasks such as image classification (Krizhevsky et al., 2012), segmentation (Long et al., 2015) and object detection (Redmon et al., 2016). Interestingly, it has recently been shown that the architecture is also highly useful also for sequence learning tasks (Bai et al., 2018b). More specifically, recent results show the architecture can match recurrent architectures in language and music modelling (Bai et al., 2018b; van den Oord et al., 2016; Dauphin et al., 2017), text-to-speech conversion (van den Oord et al., 2016), machine translation (Kalchbrenner et al., 2016; Gehring et al., 2017) and other sequential tasks (Bai et al., 2018b). In this section, we focus on uni-dimensional convolutional neural networks for sequential tasks, trying to build the basic convolutional neural network blocks from signal processing and dynamical system theory.

2.3.1 The uni-dimensional convolutional layer

The convolutional layer has as input the sequence $\{\mathbf{z}_k\}_{k=1}^N$, for which $\mathbf{z} \in \mathbb{R}^{n_z}$, and produces as output another sequence $\{\mathbf{x}_k\}_{k=1}^M$, with $\mathbf{x} \in \mathbb{R}^{n_x}$. The relation between output and input sequences is given by:

$$\mathbf{x}_k = \mathbf{f}_a(\mathbf{W}_0 \mathbf{z}_k + \mathbf{W}_1 \mathbf{z}_{k-1} + \dots + \mathbf{W}_{(n-1)} \mathbf{z}_{k-(n-1)} + \mathbf{b}), \quad (2.15)$$

where $\mathbf{f}_a: \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_x}$ is the activation function and $\mathbf{W}_i \in \mathbb{R}^{n_x \times n_z}$ are the weight matrices. The number of terms, n , is usually referred as *kernel size*. Check Figure 2.1 for a visual explanation of the convolution operation.

Let $\mathbf{W}_i^T = [\mathbf{w}_i^0; \mathbf{w}_i^1; \dots; \mathbf{w}_i^{(n-1)}]$, from a dynamical system theory perspective, each output x_k^j can be interpreted as the output of a linear time-invariant dynamical system:

$$x_k^j = \mathbf{w}_0^j \mathbf{z}_k + \mathbf{w}_1^j \mathbf{z}_{k-1} + \dots + \mathbf{w}_{(n-1)}^j \mathbf{z}_{k-(n-1)}, \quad (2.16)$$

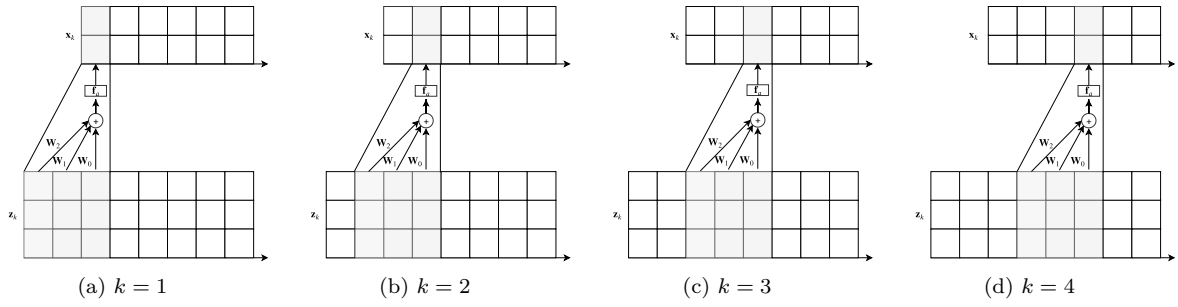


Figure 2.1 – **(Causal convolutions)** Simplified diagram illustrating causal convolutions from Eq. (2.15) for kernel size $n = 3$, input vector size $n_z = 3$, output vector size $n_x = 2$. Here the input sequence has length $N = 8$ and the output sequence has length $M = N - n + 1 = 6$. The computation is shown for the first 4 output samples.

shifted by an offset b^i and followed by a nonlinear static transformation \mathbf{f}_a . The output of linear time-invariant systems is equal to the *convolution* between the impulse response (the dynamical system response to a unitary impulse) and the input applied to the system. The output of the system (2.16) depends only on a finite number of past terms and, hence, the system described by (2.16) has finite impulse response (i.e., if you apply an impulse in this system the response will be zero after a finite number of terms). This class of systems, it happens, is widely studied in the signal processing literature and known as *finite impulse response* filter. It is often applied for removing frequency components from a signal. Hence, each convolutional layer can be actually interpreted as a bank of finite impulse response filters with trainable parameters, followed by a nonlinear transformation. This explains why the name *convolutional neural network* is appropriate for neural networks that make use of this layer.

The convolution presented above is *causal*, and the output only depends on its past values. A non-causal alternative is to include a dependence both on past and future inputs:

$$\mathbf{x}_k = \mathbf{f}_a(\mathbf{W}_{-(\frac{n-1}{2})}\mathbf{z}_{k+(\frac{n-1}{2})} + \cdots + \mathbf{W}_{-1}\mathbf{z}_{k+1} + \mathbf{W}_0\mathbf{z}_k + \mathbf{W}_1\mathbf{z}_{k-1} + \cdots + \mathbf{W}_{(\frac{n-1}{2})}\mathbf{z}_{k-(\frac{n-1}{2})} + \mathbf{b}), \quad (2.17)$$

where the kernel size n must be an odd number for the above equation to make sense.

2.3.2 Padding and the transient response

In a convolutional layer such as (2.15), input and output sequences do not necessarily have the same length. Let N be the length of the input sequence; the output sequence has length $M = N - n + 1$, where n is the kernel size. It is usual, however, to fill the input sequence with zeros at the extremity. That is, we can redefine the input sequence as $\{0, 0, \dots, 0, \mathbf{z}_1, \dots, \mathbf{z}_N\}$. The new, zero-filled, input sequence will have length $N + P$, where P is the number of zeros. And the output will have length $M = N + P - n + 1$. This trick, called *padding*, makes it possible to have $N = M$ by setting $P = n - 1$, which might be interesting for many applications. See Figure 2.2 for a visual explanation of the padding.

From a dynamical system theory perspective, by inserting zeros at the beginning of the sequence, there is a transient response, for which the system is leaving a zero-state equilibrium and adapting to the input during the first P steps.

2.3.3 Downsampling in convolutional neural networks: strides and pooling

A common operation in signal processing is decimation (or downsampling). Given a sequence $\{\mathbf{z}_k\}_{k=1}^N$, to decimate this sequence by a factor m means to keep only every m -th sample and discard the others, obtaining a new sequence with $\frac{1}{m}$ -th of the length. This new sequence is defined as $\mathbf{q}_k = \mathbf{z}_{km}$ for

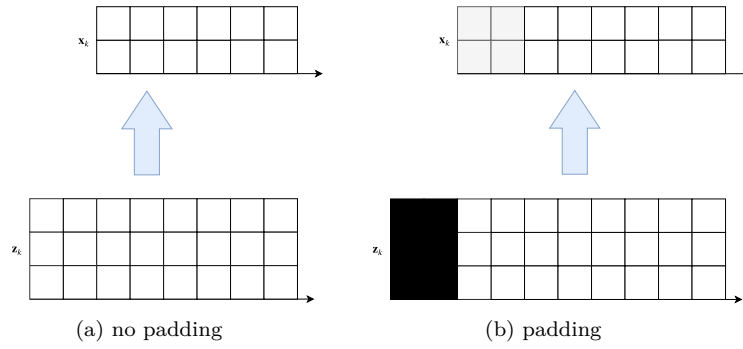


Figure 2.2 – **(Padding)** Simplified diagram illustrating the effect of padding in input and outputs for causal convolutions. In (a), no padding is used and the input sequence has length $N = 8$ and the output sequence has length $M = N - n + 1 = 6$. In (b), we use padding in order to guarantee inputs ($P = 2$) and outputs with the same length: the zero-filled input sequence has length $N + P = 8 + 2$ and the output sequence has length $M = N + P - n + 1 = 8$. In both cases, the input vector size is $n_x = 3$ and the output vector size is $n_z = 2$. The zeros included in the input are shown in **black**. Output elements that are being computed using these zeros as input are displayed in **gray**, considering the convolutional layer as a dynamic system these elements correspond to the transient response of this system from a zero-state equilibrium.

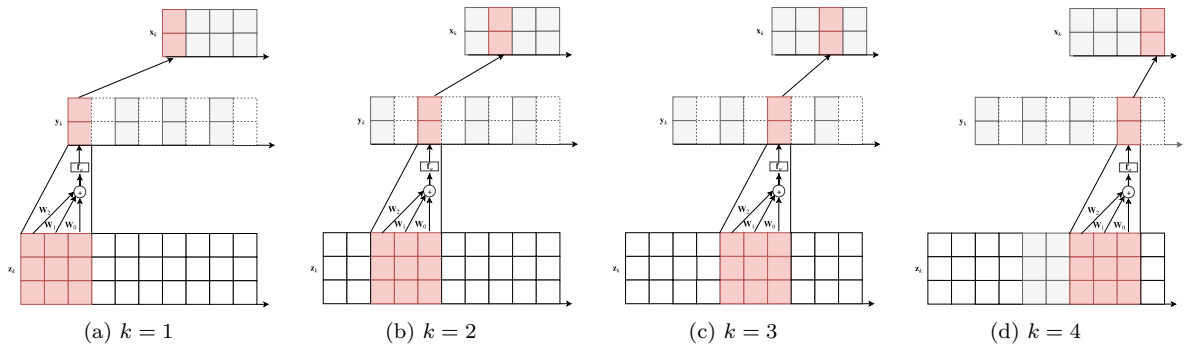


Figure 2.3 – **(Strides)** Simplified diagram illustrating causal convolutions with strides for kernel size $n = 3$, input vector size $n_x = 3$, output vector size $n_z = 2$, and downsampling factor $m = 2$. Here the input sequence \mathbf{x} has length $N = 10$ and the output sequence \mathbf{z} has length $M = (N - n + 1)/2 = 4$. The intermediary sequence \mathbf{y} has length $M = (N - n + 1) = 8$. The computation is shown for the first 4 output samples.

$k = 1, \dots, \lfloor \frac{N}{m} \rfloor$. Different operations that take place in common convolutional neural network architectures are related to decimation.¹

Strides in a convolutional layer could be interpreted as decimation applied to the signal obtained after convolution. Hence a convolutional layer with stride m can be written as:

$$\mathbf{y}_k = \mathbf{f}_a(\mathbf{W}_0 \mathbf{z}_k + \mathbf{W}_1 \mathbf{z}_{k-1} + \dots + \mathbf{W}_{(n-1)} \mathbf{z}_{k-(n-1)} + \mathbf{b}); \quad (2.18a)$$

$$\mathbf{x}_k = \mathbf{y}_{(km)}, \text{ for } k = 1, \dots, \left\lfloor \frac{N}{m} \right\rfloor. \quad (2.18b)$$

A visualisation of this two step procedure is provided in Figure 2.3.

¹ A common concern in signal processing is to avoid aliasing (i.e., the signal distortion that occurs during decimation where high frequencies appear as low frequencies, preventing the signal from being reconstructed.). Usually, by using an anti-aliasing filter before decimation. To the best of our knowledge such practice is not common when working with convolutional neural networks.

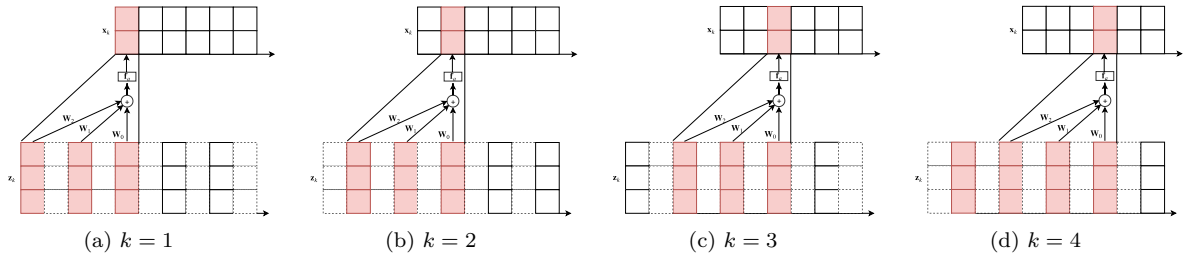


Figure 2.4 – **(Dilations)** Simplified diagram illustrating causal convolutions with dilations for kernel size $n = 3$, input vector size $n_x = 3$, output vector size $n_z = 2$, and dilation factor $d = 2$. Here the input sequence \mathbf{x} has length $N = 10$ and the output sequence \mathbf{z} has length $M = N - d * (n - 1) = 6$. The computation is shown for the first 4 output samples.

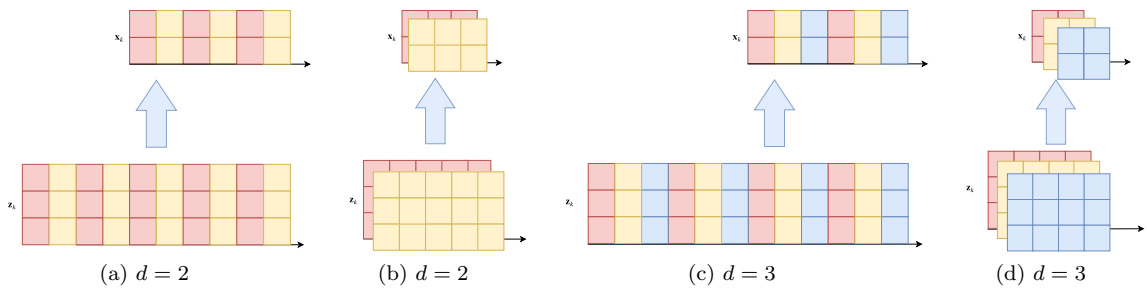


Figure 2.5 – **(Polyphase decomposition)** Simplified diagram illustrating the input and output of causal convolutions with dilations. Here we show the polyphase components of the input and output for $d = 2$ in (a) and (b); and for $d = 3$ in (c) and (d). In (a) and (c) we show the original signal and in (b) and (d) we show the signal decomposed into its d polyphase components. Convolutions with dilations applied to original signal (a) and (c), can be interpreted as standard convolutions applied to the polyphase components shown in (b) and (d).

Another common block often used in convolutional neural networks that is also related with downsampling is the so-called *max pooling layer*. The max pooling layer also reduces the dimension of the sequence by a factor m . It does that by picking the maximum of each m consecutive samples:

$$\mathbf{q}_k = \max(\mathbf{y}_{(km)}, \mathbf{y}_{(km+1)}, \dots, \mathbf{y}_{((k+1)m-1)}), \text{ for } k = 1, \dots, \left\lfloor \frac{N}{m} \right\rfloor. \quad (2.19)$$

2.3.4 Polyphase decomposition and dilations

Dilations is another operation commonly associated with the convolutional layer. The convolutional layer (2.15) with dilation d can be expressed as:

$$\mathbf{x}_k = \mathbf{f}_a(\mathbf{W}_0 \mathbf{z}_k + \mathbf{W}_1 \mathbf{z}_{k-d} + \mathbf{W}_2 \mathbf{z}_{k-2d} + \dots + \mathbf{W}_{(n-1)} \mathbf{z}_{k-d(n-1)} + \mathbf{b}). \quad (2.20)$$

A visualisation of this operation is provided in Figure 2.4.

Interestingly, dilations can also be traced back to a signal processing technique. The so-called *polyphase decomposition*, commonly used in filter implementations (Oppenheim, 1999), decomposes a signal $\{\mathbf{z}_k\}$ into d components, being the i component given by:

$$\mathbf{z}_k^i = \mathbf{z}_{(kd+i)}, \text{ for } k = 1, \dots, \left\lfloor \frac{N+i}{d} \right\rfloor. \quad (2.21)$$

Given the d polyphase components the above equation can also be used to reconstruct the original signal. Hence, Equation (2.20) can be understood as the following sequence of operations:

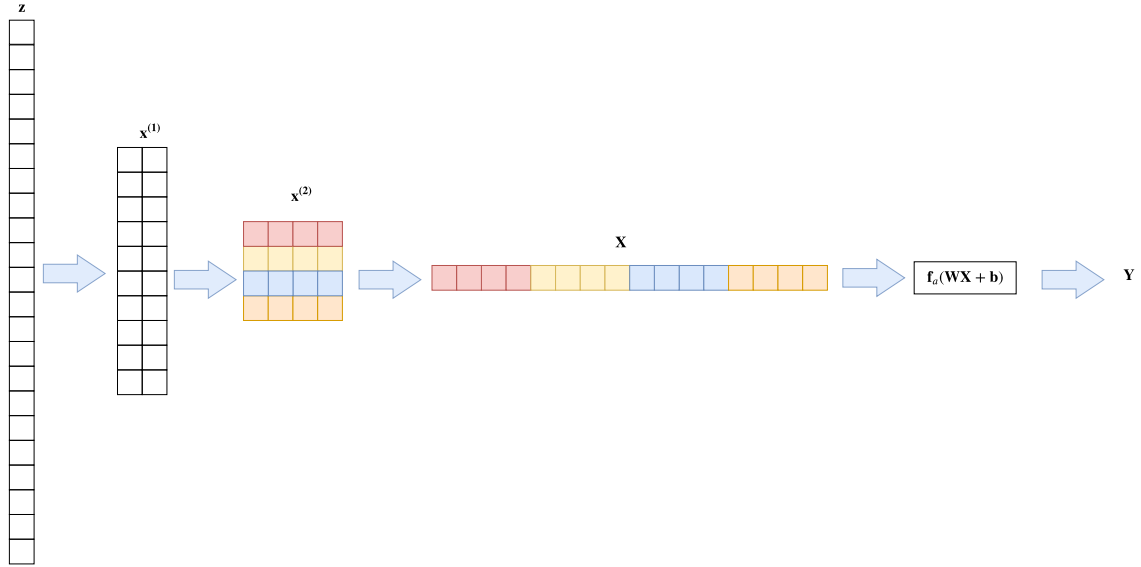


Figure 2.6 – **(Convolutional network for feature extraction)** Simplified diagram illustrating a 3 layer convolutional neural network for extracting information from a sequence. The first layer has length $N^{(1)} = 22$ and $n_x^{(1)} = 1$ channels, the second layer has length $N^{(2)} = 10$ and $n_x^{(1)} = 2$ channels, the third layer has length $N^{(3)} = 4$ and $n_x^{(1)} = 4$ channels. The elements from the last layer are flattened, and fed into a fully connected layer. A color code is used to illustrate how the flattening occurs in the last layer of the neural network.

1. Decompose the signal $\{\mathbf{z}_k\}$ into d different components using polyphase decomposition;
2. Apply the convolutional layer (2.15) to all polyphase components \mathbf{z}_k^i . With the same convolutional weights for applied all components.² The result should be d polyphase components \mathbf{x}_k^i .
3. Reconstruct the output signal \mathbf{x}_k from its d polyphase components.

This interpretation is illustrated in Figure 2.5.

2.3.5 Deep convolutional neural networks for feature extraction

To train deep neural networks with many hidden layers is a notoriously hard optimization problem. The challenges include the risk of getting stuck in bad local minima, exploding or vanishing gradients, and dealing with large-scale datasets. It is only over the past decade that these challenges have been addressed, with improved hardware and algorithms, to the extent that training truly deep neural networks became feasible.

Here we discuss the use of convolutional neural networks with many layers for *extracting information from a sequence* (as in the setup described in Section 2.1). Consider the neural network is fed with an input sequence $\{\mathbf{z}_k\}_{k=1}^N$ and yields the output vector $\hat{\mathbf{Y}}$, containing some information about the series.

Image classification was the problem for which convolutional neural networks were originally proposed (Lecun et al., 1998). And, arguably, an area for which its use is very mature: there is a massive amount of literature on the subject, openly available datasets (Krizhevsky et al., 2012) and popular architectures that yield impressive results and have been employed on several related computer vision problems (He et al., 2016a; Szegedy et al., 2015). Increasing the number of channels, while downsampling, as we progress in the deep convolutional network is something shared by most of these popular deep

² It would also be possible to use (2.17) here, when dealing with a noncausal problem.

convolutional networks. This architecture originally designed for image classification can be directly translated into convolutional neural networks for unidimensional signals.

In a multi-layer convolutional neural network the input sequence $\{\mathbf{z}_k\}$ is used as input to the first layer, which produces an intermediary sequence $\{\mathbf{x}_k^{(1)}\}$, which, in turn, is used as input to the next layer that produces the next sequence $\{\mathbf{x}_k^{(2)}\}$, and so on. Until the layer immediately before the last one produces a sequence $\{\mathbf{x}_k^{(L-1)}\}$. This sequence is put together as a vector $\mathbf{X} = (\mathbf{x}_1^{(L-1)}, \mathbf{x}_2^{(L-1)}, \dots, \mathbf{x}_{N_L}^{(L-1)})$ and fed to a final, fully connected, neural network layer, which produces the prediction $\hat{\mathbf{Y}}$. The full network can be written as:

$$\begin{aligned}\{\mathbf{x}_k^{(1)}\} &= \mathcal{F}^{(1)}\{\mathbf{z}_k\} \\ \{\mathbf{x}_k^{(2)}\} &= \mathcal{F}^{(2)}\{\mathbf{x}_k^{(1)}\}. \\ &\vdots \\ \{\mathbf{x}_k^{(L-1)}\} &= \mathcal{F}^{(L-1)}\{\mathbf{x}_k^{(L-2)}\} \\ \mathbf{X} &= (\mathbf{x}_1^{(L-1)}, \mathbf{x}_2^{(L-1)}, \dots, \mathbf{x}_{N_L}^{(L-1)}), \\ \hat{\mathbf{Y}} &= \mathbf{f}_a(\mathbf{W}\mathbf{X} + \mathbf{b})\end{aligned}$$

where $\mathcal{F}^{(l)}$ represents the l -th layer (or block) of the neural network. A convolutional neural network with $L = 3$ layers is illustrated in Figure 2.6. For the l -th layer, let us call the sequence length of the $N^{(l)}$ and the number of channels by $n_x^{(l)}$, i.e. $\mathbf{x}_k^{(l)} \in \mathbb{R}^{n_x^{(l)}}$. As mentioned above, in most architecture, the layers are chosen in such way that the sequence length $N^{(l)}$ decreases (by downsampling the sequence using convolutions with strides) and the number of channels $n_x^{(l)}$ increases with the index l .

2.3.6 Capturing input-output relation

Capturing input-output relations, as described in Section 2.1, consists of trying to predict a sequence $\{\mathbf{y}_k\}$ from a input sequence $\{\mathbf{x}_k\}$. Naturally, the architecture described in the previous section, which predicts a single vector from a given input sequence, could be used here just by using it to predict every k -th value of the output. If the system is time-invariant, the same weights could be used for predicting all output samples. Dilations offer an efficient way to implement something that is roughly equivalent to this.

The neural network for capturing input-output relations uses padding and dilations in order to have all signals from intermediary layers with exactly the same length, cf. Figure 2.7(a). Hence the input sequence $\{\mathbf{z}_k\}$ is used as input to the first layer, which produces an intermediary sequence $\{\mathbf{x}_k^{(1)}\}$ with the same length which, in turn, is used as input to the next layer that produces the next sequence $\{\mathbf{x}_k^{(2)}\}$ with the same length, and so on. Until the last layer produces a sequence $\{\hat{\mathbf{y}}_k\}$, corresponding to the output prediction. The full network can be written as:

$$\begin{aligned}\{\mathbf{x}_k^{(1)}\} &= \mathcal{F}^{(1)}\{\mathbf{z}_k\} \\ \{\mathbf{x}_k^{(2)}\} &= \mathcal{F}^{(2)}\{\mathbf{x}_k^{(1)}\}. \\ &\vdots \\ \{\mathbf{x}_k^{(L-1)}\} &= \mathcal{F}^{(L-1)}\{\mathbf{x}_k^{(L-2)}\} \\ \{\hat{\mathbf{y}}_k\} &= \mathcal{F}^{(L)}\{\mathbf{x}_k^{(L-1)}\}.\end{aligned}$$

The idea is to increase the dilation factor d as the index l increases, i.e. set the dilation factor of the l -th layer as $d = 2^l$. And, at the same time, increase the number of channels $n_x^{(l)}$ with the number of layers.

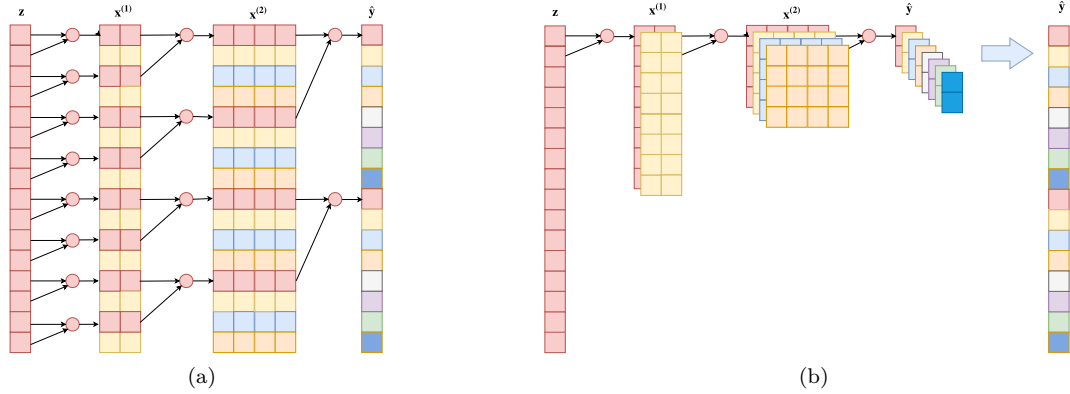


Figure 2.7 – **(Convolutional network modeling input-output relations)** Simplified diagram illustrating a 3 layer convolutional neural network for modeling the input-output relation between signals. The dilation factor for the three layers is 1, 2, 4. All layers use a kernel size $n = 2$. The input and output have dimension 1, and the intermediary layers dimension 2 and 4, respectively. The color code in (a) shows the different polyphase components (cf. Section 2.3.4) in the input, output and intermediary signals, the same colors are used in (b) to denote the same components. The purpose is to illustrate how the use of dilations is equivalent to that of downsampling and strides in the convolutional neural network of Figure 2.6.

The polyphase interpretation of dilations is useful here: so as we progress through the layers, we increase the number of polyphase components, and this is, to some extent, equivalent to successively downsampling the signal, this is illustrated in Figure 2.7(b).

2.3.7 Skip connections and the residual neural network

The residual network (He et al., 2016a) is quite popular among practitioners and has the residual block as its key component. A residual block adds a skip connection to a convolutional layer. Thus:

$$z^{(l+p)} = \mathcal{F}(z^{(l)}) + z^{(l)}. \quad (2.24)$$

This connection adds the value from the input of the block to its output. The purpose of the skip connection is to let the layers learn deviations from the identity map rather than the whole transformation. This property is beneficial, especially in deep networks, and allows the information to gradually change from the input to the output as we proceed through the layers. There is also some evidence that this makes it easier to train deeper neural networks (He et al., 2016a).

3 System identification and prediction error methods

Models of dynamical systems are of fundamental importance in engineering. In some situations, the model can be obtained exclusively from the basic physical laws describing the process. There are some situations, however, where this approach may not be possible or may yield unsatisfactory results: the system may be too complex, or the behavior of some parts of the process may be unknown. The alternative approach is to collect data and create a model consistent with the observed behavior. Building empirical mathematical models from data is relevant for the most diverse areas, and akin field of studies have arisen from the more diverse communities. Machine learning, mentioned in Chapter 1, is one of such fields.

System identification (Söderström and Stoica, 1988; Ljung, 1998; Aguirre, 2004; Nelles, 2013) arises from the need of the control community to estimate mathematical models to control and monitor dynamical processes. It is strongly focused on building mathematical models for *dynamic* processes. The model can be built exclusively from data (*black-box modeling*), or, alternatively, we may try to include prior knowledge about the process in the model (*grey-box modeling*) (Aguirre, 2019). In this case, the observation data may be used to refine a physical model of the process; or, conversely, knowledge about the process may be used to obtain better models when building a model from data.

In this chapter, we present a short introduction to system identification, covering topics that are relevant to this thesis. The fundamentals of system identification overlap with the fundamentals of machine learning in many aspects. Hence almost all the topics presented in Chapter 1 could easily fit here as well, maybe with different nomenclature. One parameter estimation technique that arises in the context of system identification is that of *prediction error methods*. This class of methods can be understood as a special case of maximum likelihood estimation and has some interesting properties that will be further discussed in this chapter.

3.1 Mathematical representation of dynamical systems

There are different ways to represent a dynamic system, and it is important to choose a representation that has the capability of capturing the main features of the process being modeled.

A mathematical model of a system is represented in Figure 3.1. In this figure, the system is represented as an operator $T\{\bullet\}$ that relates the inputs \mathbf{u} and the outputs \mathbf{y} . The model is *dynamic* if the output at a given moment depends on its previous values (it has memory) or future values.

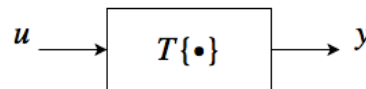


Figure 3.1 – **(Dynamical system)** Display a schematic representation of the model used to represent a dynamical system.

In Section 3.1.1 the available choices of dynamic representation will be explored; in Section 3.1.2 we explain the use of approximation functions in order to yield flexible models that are capable of fitting the data; and, how disturbances and uncertainties can be taken into account in the model is explained in Section 3.1.3.

3.1.1 Dynamic representation

There are many different ways to represent a dynamic system. The representations can be classified into discrete and continuous-time. For discrete-time models the input and output are sequences, $\{\mathbf{y}[k]\}$ and $\{\mathbf{u}[k]\}$, defined only at discrete times. For continuous-time models, on the other hand, the input and output, $\mathbf{y}(t)$ and $\mathbf{u}(t)$, are defined on a continuum of times.

Continuous-time models may be represented as an *explicit ordinary differential equation*:

$$\frac{d^n \mathbf{y}(t)}{dt^n} = \mathbf{f} \left(t, \mathbf{u}(t), \frac{d\mathbf{u}(t)}{dt}, \frac{d^2 \mathbf{u}(t)}{dt^2} \dots, \frac{d^m \mathbf{u}(t)}{dt^m}, \mathbf{y}(t), \frac{d\mathbf{y}(t)}{dt}, \frac{d^2 \mathbf{y}(t)}{dt^2} \dots, \frac{d^{n-1} \mathbf{y}(t)}{dt^{n-1}} \right), \quad (3.1)$$

for which $\mathbf{u}(t) \in \mathbb{R}^{N_u}$ and $\mathbf{y}(t) \in \mathbb{R}^{N_y}$ are vectors containing the system inputs and outputs at a given time instant t . The first order derivative of \mathbf{u} is denoted by $\frac{d\mathbf{u}(t)}{dt}$ and the m -th order derivative by $\frac{d^m \mathbf{u}(t)}{dt^m}$. And \mathbf{f} is a function that gives the relation between the derivatives.

Another, commonly used continuous representation is the *state-space representation*:

$$\begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t), t), \\ \mathbf{y}(t) &= \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t), \end{aligned} \quad (3.2)$$

where $\mathbf{x}(t) \in \mathbb{R}^{N_x}$ is the so-called state vector.

The explicit dependence of the model with time may make the system identification problem much more difficult, so it is a common design choice not to take into account this dependence. A state-space *time-invariant* model can be written as:

$$\begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t)), \\ \mathbf{y}(t) &= \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)). \end{aligned} \quad (3.3)$$

The *discrete-time* state-space representation is similar to the continuous one, and maybe written, for the time-invariant case, as:

$$\begin{aligned} \mathbf{x}[k] &= \mathbf{h}(\mathbf{x}[k-1], \mathbf{u}[k]), \\ \mathbf{y}[k] &= \mathbf{g}(\mathbf{x}[k], \mathbf{u}[k]). \end{aligned} \quad (3.4)$$

Another common representation for discrete-time models are the so-called *difference equations*, which are the discrete analogous of differential equations. A time-invariant explicit difference equation could be written as follows:

$$\mathbf{y}[k] = \mathbf{f}(\mathbf{y}[k-1], \dots, \mathbf{y}[k-n_y], \mathbf{u}[k-\tau], \dots, \mathbf{u}[k-n_u]), \quad (3.5)$$

where the positive integer τ is the input-output delay, which is included in the model to reflect that changes in the input may take some time to affect the output. And n_u and n_y are the maximum input and output lags, respectively.

This dissertation is interested mostly in discrete-time models. The data available for system identification is usually a discrete sequence of sampled values and it is practical to build discrete-time models (even for inherently continuous systems).

To make it easier to write large difference equations the following convention will be adopted.

Notation 3.1: This dissertation will use the notation:

$$\begin{aligned} \underline{\mathbf{y}}_{k-i} &= [\mathbf{y}[k-i]^T \mathbf{y}[k-i-1]^T \dots \mathbf{y}[k-n_y]^T]^T; \\ \underline{\mathbf{u}}_{k-i} &= [\mathbf{u}[k-i]^T \mathbf{u}[k-i-1]^T \dots \mathbf{u}[k-n_u]^T]^T. \end{aligned}$$

Hence, Equation (3.5) may be rewritten as: $\mathbf{y}[k] = \mathbf{f}(\underline{\mathbf{y}}_{k-1}, \underline{\mathbf{u}}_{k-\tau})$.

3.1.2 Approximation function

In the previously section, the functions \mathbf{f} , \mathbf{g} and \mathbf{h} were left unspecified. Usually, in system identification, the functions are (partially) unknown, and the problem to be solved is to determine them in order to fit the training data. They are usually chosen to be functions that, depending on the choice of parameters, can approximate a large class of behaviors.

A very usual choice is an affine function: $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ where \mathbf{A} is a matrix and \mathbf{b} is a vector. This, linear function, for the right choice of parameters, can approximate *locally* any differentiable function and much of the classic theory in system identification is focused in estimating linear representations (Ljung, 1998).

There are, however, several dynamic behaviors that cannot be represented using a linear representation (e.g., chaotic behavior, presence of limit cycles and multiple equilibrium points). As stated, in (Ljung, 2010, Sec. 4.1), the use of nonlinear representations is one of the most active areas of research in system identification and will be the main focus of this dissertation.

There are several nonlinear functions with approximation capabilities: Polynomials, due to Taylor theorem, can be used to approximate analytic functions and are widely used in system identification (Chen et al., 1989; Paduart et al., 2010; Farina and Piroddi, 2010, 2012); Feedforward networks, radial basis functions networks and fuzzy systems are universal approximators¹ and, because of that, are also widely applicable in system identification (Chen et al., 1990a; Narendra and Parthasarathy, 1990; Nørgaard et al., 2000; Kayacan et al., 2015). Refer to (Nelles, 2013) for a complete list of nonlinear representations often used on system identification. Specificities of each approximation function will be detailed when needed, along with the dissertation.

3.1.3 Noise model

So far, we said nothing about the disturbances and noise interfering with the system. But, usually, how to take those into account is also a design choice, and different assumptions about the *data generation process* will yield different estimation procedures.

For instance, three types of errors are typically considered for the difference equation (3.5): output error, equation error and error-in-variables. The error-in-variables and the output error are usually related to noise corrupting the input and output measurements. The equation error, on the other hand, accounts for unmeasured disturbance and unknown dynamics. More specifically, given the *data generator system*:

$$\begin{aligned}\mathbf{u}[k] &= \mathbf{u}^*[k] + \mathbf{s}[k]; \\ \mathbf{y}^*[k] &= \mathbf{f}^*(\mathbf{y}_{k-1}^*, \mathbf{u}_{k-\tau}^*) + \mathbf{v}[k]; \\ \mathbf{y}[k] &= \mathbf{y}^*[k] + \mathbf{w}[k],\end{aligned}\tag{3.6}$$

the final goal of the identification process is to determine a function \mathbf{f} as similar to \mathbf{f}^* as possible within a set of possibilities, from the noisy measured values, $\mathbf{u}[k]$ and $\mathbf{y}[k]$. Here, \mathbf{f}^* is the function that describes the system, \mathbf{s} , \mathbf{v} and \mathbf{w} are random processes representing, respectively, error-in-variables, equation error and output error. Noise-free input and output are represented by $\mathbf{u}^*[k]$ and $\mathbf{y}^*[k]$. Different assumptions about the *data generation process* and the noise processes, however, will yield different estimation procedures. This will be described in detail in the next section.

¹ These function can approximate a certain set of continuous function with arbitrary accuracy ϵ on compact sets (Hornik et al., 1989a; Park and Sandberg, 1991; Tikk et al., 2003).

3.2 Prediction error methods

Prediction error methods are a class of methods for estimation of parameters of dynamical systems, and this section complements Section 1.1 with the description of this class of methods. Early works on prediction error methods formulate these methods in very general terms in a framework that holds both for linear and nonlinear systems very well, e.g. (Ljung, 1976, 1978; Ljung and Caines, 1980). Due to the great importance of linear prediction error methods, however, classical textbooks on system identification, such as (Söderström and Stoica, 1988; Ljung, 1998), choose to present prediction error methods in an entirely linear framework, mentioning the nonlinear case only as a possible extension, usually by analogy.

This trend was carried on even to identification textbooks that are focused in nonlinear models, such as (Nørgaard et al., 2000; Nelles, 2013). The nomenclature for prediction error methods also reflects this. Three *linear* models within prediction error framework are: ARX (*autoregressive with exogenous input*); ARMAX (*autoregressive moving average with exogenous input*); and, OE (*output error*) models. Models for systems that are not necessarily linear are called NARX (*nonlinear ARX*), NARMAX (*nonlinear ARMAX*) and NOE (*nonlinear OE*). Notice that the names do not follow the more logical structure of using a name for the more general case, adding qualifiers on the name as more restrictive structures are chosen, but the opposite, they start with a name for the linear case and add a “N” for the more general case. In this dissertation, we will try to present prediction error methods in the more general, nonlinear, framework.

3.2.1 Setup

Consider the dataset $\mathcal{Z}^N = \{(\mathbf{u}[k], \mathbf{y}[k]), k = 1, 2, \dots, N\}$ containing N measured inputs and outputs of a dynamical system. Prediction error methods assume a prediction model that gives a predicted output $\hat{\mathbf{y}}[k]$, for each time instant $k = 1, 2, \dots, N$. A cost function is defined as the distance between the prediction and the measured value:²

$$V = \frac{1}{N} \sum_{k=1}^N \|\mathbf{y}[k] - \hat{\mathbf{y}}[k]\|^2. \quad (3.8)$$

For the prediction model it is usual to employ a parameterized model depending on a parameter vector $\boldsymbol{\theta}$ and, although such a dependence is not made explicit in the notation, $\hat{\mathbf{y}}[k]$ depends upon $\boldsymbol{\theta}$. An estimate $\hat{\boldsymbol{\theta}}$ of the parameter vector may be obtained by minimizing V .

Next, we give examples of prediction error models. These examples assume the measured input-output data was generated by a dynamical, stochastic, discrete-time system. Lets:

$$\begin{aligned} \underline{\mathbf{u}}[k] &= (\mathbf{u}[k], \dots, \mathbf{u}[k - n_u]), \\ \underline{\mathbf{y}}[k - 1] &= (\mathbf{y}[k - 1], \dots, \mathbf{y}[k - n_y]), \end{aligned}$$

where n_y, n_u are the maximum input and output lags. The examples differ in how they take the noise into the model. But, in a noiseless situation, the data generation model for the three examples correspond to a difference equation $\mathbf{y}[k] = \mathbf{f}^*(\underline{\mathbf{y}}[k - 1], \underline{\mathbf{u}}[k])$. We define the parametrized function $\mathbf{f}_{\boldsymbol{\theta}}$ and, if our noise

² Ljung (1978) gives a slightly more general framework, and defines the cost function as:

$$V = h \left(\frac{1}{N} \sum_{k=1}^N l(\mathbf{y}[k] - \hat{\mathbf{y}}[k]) \right). \quad (3.7)$$

For any function l that maps a vector into a positive definite matrix (i.e. $l(\boldsymbol{\epsilon}) = \boldsymbol{\epsilon}\boldsymbol{\epsilon}^T$), and, for a function h that maps a positive definite matrix into a scalar. For any positive definite matrix Q and any matrix Λ , it must respect $h(Q + \Lambda) \geq h(Q) + h(\Lambda)$. Two possible choices of $h(\cdot)$ satisfying this condition are $h(Q) = \det(Q)$ and $h(Q) = \text{tr}(\Sigma Q)$ for any a symmetric positive definite weighting matrix Σ .

assumption and model structure is correct, then for the models we describe next the minimization of the cost function V would yield the estimate $\hat{\boldsymbol{\theta}}$ such that, as $N \rightarrow \infty$, then $\mathbf{f}_{\hat{\boldsymbol{\theta}}} \rightarrow \mathbf{f}^*$ (or to function with the same performance in predicting the training dataset). A more precise description of the asymptotic properties of nonlinear prediction error methods is described in Section 3.2.8.

Choosing the right model structure (one for which there exists $\boldsymbol{\theta}^*$ such that $\mathbf{f}_{\boldsymbol{\theta}^*} = \mathbf{f}^*$) might be impossible in a practical application, nevertheless the assumption is not so restrictive as it might appear, since there exist families of universal approximator functions (e.g. neural networks and polynomials) for which the distance $\|\mathbf{f}_{\boldsymbol{\theta}} - \mathbf{f}^*\|$ might be made arbitrarily small within a compact set.

3.2.2 Nonlinear ARX models

The nonlinear ARX (*autoregressive with exogenous input*) model considers the output at instant k is corrupted by a white *process* noise. That is, it assumes the data was generated by the stochastic discrete-time system:

$$\mathbf{y}[k] = \mathbf{f}^*(\mathbf{y}[k-1], \mathbf{u}[k]) + \mathbf{v}[k], \quad (3.9)$$

where $\mathbf{v}[k]$ is a zero-mean white noise. This assumption yields (cf. Appendix 3.2.8) the prediction model:

$$\hat{\mathbf{y}}[k] = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{y}[k-1], \mathbf{u}[k]). \quad (3.10)$$

And the minimization of the cost function in Eq. (3.8) yields an estimator with the desired asymptotic properties.

3.2.3 Nonlinear output error models

Output error models consider the output at instant k is corrupted by a white *measurement* noise. That is, it assume the data was generated by:

$$\begin{aligned} \bar{\mathbf{y}}[k] &= \mathbf{f}^*(\bar{\mathbf{y}}[k-1], \dots, \bar{\mathbf{y}}[k-n_y], \mathbf{u}[k]); \\ \mathbf{y}[k] &= \bar{\mathbf{y}}[k] + \mathbf{v}[k], \end{aligned}$$

where, again, $\mathbf{v}[k]$ is a zero-mean white noise and $\bar{\mathbf{y}}[k]$ represents the noiseless output. This assumption yields the prediction model:

$$\begin{aligned} \tilde{\mathbf{y}}[k] &= \mathbf{f}_{\boldsymbol{\theta}}(\tilde{\mathbf{y}}[k-1], \dots, \tilde{\mathbf{y}}[k-n_y], \mathbf{u}[k]); \\ \hat{\mathbf{y}}[k] &= \tilde{\mathbf{y}}[k]. \end{aligned} \quad (3.11)$$

Here $\tilde{\mathbf{y}}[k]$ represents an estimation of the noiseless output $\bar{\mathbf{y}}[k]$, which should approach the true value as $\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}^*$.

3.2.4 Nonlinear ARMAX models

The ARMAX (*autoregressive moving average with exogenous input*) models consider the output at instant k is corrupted by an additive zero-mean *process* noise. In this case, the propagation equation allows the noise terms $\mathbf{v}[k]$ to be propagated by the dynamics:

$$\mathbf{y}[k] = \mathbf{f}^*(\mathbf{y}[k-1], \mathbf{u}[k], \mathbf{v}[k-1], \dots, \mathbf{v}[k-n_v]) + \mathbf{v}[k].$$

This assumption allows the model to account for some forms of colored process noise and yields the prediction model:

$$\begin{aligned} \tilde{\mathbf{v}}[k] &= \mathbf{y}[k] - \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{y}[k-1], \mathbf{u}[k], \tilde{\mathbf{v}}[k-1], \dots, \tilde{\mathbf{v}}[k-n_v]); \\ \hat{\mathbf{y}}[k] &= \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{y}[k-1], \mathbf{u}[k], \tilde{\mathbf{v}}[k-1], \dots, \tilde{\mathbf{v}}[k-n_v]). \end{aligned} \quad (3.12)$$

Here $\tilde{\mathbf{v}}[k]$ represents an estimation of the noise corrupting the system and should approach the true noise if the estimated parameter vector approach the true parameter value $\boldsymbol{\theta}^*$.

3.2.5 General nonlinear state-space framework

From now on, we will focus on a more general state-space representation that encompasses all the three examples we just mentioned (for an appropriate choice of the functions \mathbf{h} and \mathbf{g}). For this representation, the predicted output is given by:

$$\mathbf{x}[k] = \mathbf{h}(\mathbf{x}[k-1], \mathbf{z}[k]; \boldsymbol{\theta}); \quad (3.13a)$$

$$\hat{\mathbf{y}}[k] = \mathbf{g}(\mathbf{x}[k], \mathbf{z}[k]; \boldsymbol{\theta}), \quad (3.13b)$$

where $\mathbf{x}[k]$ denotes the internal state vector of this model at instant k . For the ARX model we would have an empty transition state $\mathbf{x} = \emptyset$; for the output error model we have $\mathbf{x}[k] = (\bar{\mathbf{y}}[k-1], \dots, \bar{\mathbf{y}}[k-n_v])$; and, for ARMAX model we would have $\mathbf{x}[k] = (\tilde{\mathbf{v}}[k-1], \dots, \tilde{\mathbf{v}}[k-n_y])$.

Here we define $\mathbf{z}[k] = (\mathbf{y}[k-1], \mathbf{u}[k])$. Using both inputs $\mathbf{u}[k]$ and autorregressive terms $\mathbf{y}[k-1]$ is what allows this state-space representation to encompass ARX, ARMAX and output error models, and also other representations such as the polynomial greybox models proposed in (Noël and Schoukens, 2018).

3.2.6 Initial conditions

In order to guarantee the desirable asymptotic properties, the prediction model needs to be simulated, starting with appropriate initial conditions \mathbf{x}_0 . Since the true initial condition, \mathbf{x}_0^* is unknown. There are two possible approaches when estimating the parameters.

The first approach is to fix \mathbf{x}_0 , for some $\mathbf{x}_0 \approx \mathbf{x}_0^*$, and minimize the cost function (3.8). This approach is based on the idea that, for an asymptotically stable system, the influence of the initial conditions on the output decreases with time for many cases of interest (Boyd and Chua, 1985a) and, hence, even if $\mathbf{x}_0 \neq \mathbf{x}_0^*$ we can still obtain a good estimate of the parameters. In this case the first samples may be discarded, to make sure the transient errors are not too large. For the ARMAX model, an appropriate choice of initial values would be $\tilde{\mathbf{v}}[k] = \mathbf{0}$, $k = 1, \dots, n_v$ and, for the output error model, $\tilde{\mathbf{y}}[k] = \mathbf{y}[k]$, $k = 1, \dots, n_y$.

The second approach consists of including \mathbf{x}_0 in the optimization problem, so it converges to \mathbf{x}_0^* and improves the quality of the parameter estimates. The optimization problem to be solved in this case is to minimize V with both $\boldsymbol{\theta}$ and \mathbf{x}_0 as optimization variables:

$$\min_{\boldsymbol{\theta}, \mathbf{x}_0} V. \quad (3.14)$$

3.2.7 The data generation process

Consider $\mathbf{y}[k]$ and $\mathbf{u}[k]$ to be one realization of the random variables $\mathbf{Y}[k]$ and $\mathbf{U}[k]$. And denote:

$$\begin{aligned} \underline{\mathbf{U}}[k] &= [\mathbf{U}[k], \dots, \mathbf{U}[k-n_u]], \\ \underline{\mathbf{Y}}[k-1] &= [\mathbf{Y}[k-1], \dots, \mathbf{Y}[k-n_y]], \end{aligned}$$

Hence, rewriting the data generation difference equations for nonlinear ARX, output error and ARMAX (see Sections 3.2.2, 3.2.3 and 3.2.4) with the new notation yields, respectively:

$$\bullet \quad \mathbf{Y}[k] = \mathbf{f}^*(\mathbf{Y}[k-1], \mathbf{U}[k]) + \mathbf{V}[k]; \quad (3.15)$$

$$\bullet \quad \begin{cases} \bar{\mathbf{Y}}[k] = \mathbf{f}^*(\bar{\mathbf{Y}}[k-1], \dots, \bar{\mathbf{Y}}[k-n_y], \mathbf{U}[k]); \\ \mathbf{Y}[k] = \bar{\mathbf{Y}}[k] + \mathbf{V}[k]; \end{cases} \quad (3.16)$$

$$\bullet \quad \mathbf{Y}[k] = \mathbf{f}^*(\mathbf{Y}[k-1], \mathbf{U}[k], \mathbf{V}[k-1], \dots, \mathbf{V}[k-n_v]) + \mathbf{V}[k]. \quad (3.17)$$

Here $\mathbf{V}[k]$ is a random variable representing the white noise that is injected in the system. Notice that for the three examples there is a deterministic additive relation between $\mathbf{Y}[k]$ and $\mathbf{V}[k]$. Hence, if $\mathbf{Y}[k]$ and $\mathbf{U}[k]$ are determined, so is $\mathbf{V}[k]$, or, conversely, if $\mathbf{V}[k]$ and $\mathbf{U}[k]$ are determined, so is $\mathbf{Y}[k]$.

3.2.8 Asymptotic properties

3.2.8.1 Optimal output prediction

Lets define the *optimal output prediction* at time k as the following conditional expectation:

$$\hat{\mathbf{y}}_*[k] = E \left\{ \mathbf{Y}[k] \mid \mathbf{U}[k] = \mathbf{u}[k], \mathbf{Y}[k-1] = \mathbf{y}[k-1] \right\}, \quad (3.18)$$

which is, in the least square sense, the best prediction for the output given its previous values.³ For the nonlinear ARX, output error and ARMAX models the *optimal output prediction* are given, respectively, by:

$$\bullet \quad \begin{cases} \hat{\mathbf{y}}_*[k] = \mathbf{f}^*(\mathbf{y}[k-1], \mathbf{u}[k]); \\ \begin{cases} \tilde{\mathbf{v}}[k] = \mathbf{y}[k] - \mathbf{f}^*(\mathbf{y}[k-1], \mathbf{u}[k], \tilde{\mathbf{v}}[k-1], \dots, \tilde{\mathbf{v}}[k-n_v]); \\ \hat{\mathbf{y}}_*[k] = \mathbf{f}^*(\mathbf{y}[k-1], \mathbf{u}[k], \tilde{\mathbf{v}}[k-1], \dots, \tilde{\mathbf{v}}[k-n_v]); \end{cases} \\ \begin{cases} \bar{\mathbf{y}}[k] = \mathbf{f}^*(\bar{\mathbf{y}}[k-1], \dots, \bar{\mathbf{y}}[k-n_y], \mathbf{u}[k]); \\ \hat{\mathbf{y}}_*[k] = \bar{\mathbf{y}}[k]. \end{cases} \end{cases}$$

which follows from a direct application of the definition (4.3) to the stochastic difference equation that are assumed for the data generation in each case.

3.2.8.2 Optimal cost function

Ideally the model predicted output $\hat{\mathbf{y}}[k]$ should be as close as possible to the optimal one $\hat{\mathbf{y}}_*[k]$. The distance is given by the *ideal* cost function:

$$V_* = \frac{1}{N} \sum_{k=1}^N \|\hat{\mathbf{y}}_*[k] - \hat{\mathbf{y}}[k]\|^2. \quad (3.19)$$

Now, for nonlinear ARX, ARMAX or output error models, given a parametrized function \mathbf{f}_θ with the correct model structure (that is: there exist θ^* such that $\mathbf{f}_{\theta^*} = \mathbf{f}^*$), it follows from Eqs. (3.10), (3.11) and (3.12) that θ^* yields $V_* = 0$, and hence the optimal parameter θ^* is the a minimizer of V_* .

3.2.8.3 Uniform convergence of V

The optimal cost function V_* is not available for optimization. Nevertheless, under some mild regularity conditions, it has been proved that $V \rightarrow V_*$ with probability 1 as $N \rightarrow \infty$ and that *this convergence is uniform* (Ljung, 1978).

³

This prediction provides the smallest squared conditional expected error between the predicted and observed values:

$$\hat{\mathbf{y}}_*[k] = \arg_{\hat{\mathbf{Y}}} \min E \left\{ \|\mathbf{Y}[k] - \hat{\mathbf{Y}}\|^2 \mid \mathbf{U}[k], \mathbf{Y}[k-1] \right\}.$$

If V_* has a single minimum θ^* and V is convex in a convex set containing θ^* the minimizer of V converges to the minimizer of V_* (Newey and McFadden, 1994, Theorem 2.1). Alternative conditions for this to hold are given in (Ljung, 1978). For our three examples, this would imply $\mathbf{f}_\theta \rightarrow \mathbf{f}^*$ (convergence in probability for $N \rightarrow \infty$). If the conditions are not satisfied, the uniform convergence, at least, guarantee that the minimizer has an equivalent performance (as $N \rightarrow \infty$). Additionally, in (Ljung and Caines, 1980) it is shown that if the solution is unique the estimator has an asymptotic normal distribution.

3.2.9 Prediction error and maximum likelihood

Prediction error estimation methods might be derived from maximum likelihood estimation under some conditions. We assume that the output $\mathbf{y}[k]$ is normally distributed with conditional mean $\mathbf{f}(\mathbf{x}[k], \theta)$ and constant covariance matrix $\sigma^2 \mathbf{I}$. That is, $p(\mathbf{y}[k] | \mathbf{x}[k]) \sim \mathcal{N}(\mathbf{f}(\mathbf{x}[k], \theta), \sigma^2 \mathbf{I})$, where $\mathbf{x}[k] = (\mathbf{y}[1], \dots, \mathbf{y}[k-1], \mathbf{u}[1], \dots, \mathbf{u}[k])$ contains past values (See 1.1.1 for a similar construction). Using exactly the same procedure as in Section 1.1.2, we have that the maximum likelihood estimation of θ is given by the minimizer of:

$$V(\theta) = \frac{1}{N} \sum_{k=1}^N \|\mathbf{y}[k] - \mathbf{f}(\mathbf{x}[k], \theta)\|^2, \quad (3.20)$$

and the equivalence with Equation (3.8) follows from defining $\hat{\mathbf{y}}[k] = \mathbf{f}(\mathbf{x}[k], \theta)$. Also, the maximum likelihood estimator for the variance is given by: $\hat{\sigma}^2 = \frac{1}{nN} \sum_{i=1}^N \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}[k], \theta)\|^2$.

3.2.10 Some terminology

The sequence of random variables associated with the input values $\{\mathbf{U}[k]\}$, or the rule used to define these values, are referred to as the *experimental condition*.

Let us define the set \mathcal{R} as the set of all parameters values θ which yield $\hat{\mathbf{y}}[k] = \hat{\mathbf{y}}_*[k]$ for $k = 1, \dots, N$. If the set \mathcal{R} is not empty then we say there is *system identifiability*. That is, we can recover a system that has the same behavior under the same experimental conditions.

Notice that all elements of \mathcal{R} are minimizers of V_* , but not necessarily all minimizers of V_* belong to \mathcal{R} . For this to happen, one must choose an experimental condition that yields a sufficiently representative picture of how the system behaves.

Assume the model structure matches the one from the data generation process. If \mathcal{R} is not-empty and also consists of a single element, we say there is *parameter identifiability*, since it is possible to recover the parameter set that generated the data.

3.2.11 Computing the derivatives

3.2.11.1 Sensitivity equations

Let the Jacobian matrices of $\mathbf{h}(\mathbf{x}, \mathbf{z}; \theta)$ with respect to \mathbf{x} and to θ evaluated at the point $(\mathbf{x}[k], \mathbf{z}[k]; \theta)$ be denoted, respectively, as A_k and B_k . Similarly, the Jacobian matrices of $\mathbf{g}(\mathbf{x}, \mathbf{z}; \theta)$ are denoted as C_k and F_k . Also, we denote the Jacobian matrices of $\hat{\mathbf{y}}[k]$ with respect to θ and to \mathbf{x}_0 as $J_\theta[k]$ and $J_{\mathbf{x}_0}[k]$. And the Jacobian matrices of $\mathbf{x}[k]$ are denoted as $D_\theta[k]$ and $D_{\mathbf{x}_0}[k]$.

A direct application of the chain rule to (3.13) gives a recursive formula for computing the derivatives of the predicted output in relation to the parameters in the interval $1 \leq k \leq N$:

$$\begin{aligned} D_\theta[k] &= A_k D_\theta[k-1] + B_k \text{ for } D_\theta[0] = \mathbf{0}, \\ J_\theta[k] &= C_k D_\theta[k] + F_k. \end{aligned} \quad (3.21)$$

A similar recursive formula may be used for computing the derivatives of the predicted output in relation to the initial conditions:

$$\begin{aligned} D_{\mathbf{x}_0}[k] &= A_k D_{\mathbf{x}_0}[k-1] \text{ for } D_{\mathbf{x}_0}[0] = \mathbf{I}; \\ J_{\mathbf{x}_0}[k] &= C_k D_{\mathbf{x}_0}[k]. \end{aligned} \quad (3.22)$$

Finally, we define $D[k] = [D_{\boldsymbol{\theta}}[k], D_{\mathbf{x}_0}[k]]$ and $J[k] = [J_{\boldsymbol{\theta}}[k], J_{\mathbf{x}_0}[k]]$.

These sensitivity equations will be used both for numerical implementations and in theoretical derivations. It is important to highlight that, from an implementation point of view, the underlying procedure used here to propagate the derivatives is entirely equivalent to the forward-mode automatic differentiation, described in Section 1.4.1.

3.2.11.2 Derivatives of the cost function

For the cost function V defined as in (3.8), its gradient ∇V is given by:

$$\nabla V = \frac{2}{N} \sum_{k=1}^N J[k]^T (\hat{\mathbf{y}}[k] - \mathbf{y}[k]). \quad (3.23)$$

Its Hessian $\nabla^2 V$ is given by:

$$\nabla^2 V = \frac{2}{N} \sum_{k=1}^N (J[k]^T J[k] + \mathbf{S}[k]), \quad (3.24)$$

where $\mathbf{S}[k] = \sum_{j=1}^{N_y} \hat{y}_j[k] \nabla^2 \hat{y}_j[k]$. Ignoring $\mathbf{S}[k]$ is a common approximation used in nonlinear least-squares implementations, that will be employed in Chapters 4 and 5.

3.3 Test design and data collection

A dataset containing information about the system behavior in most situations of interest is necessary to estimate a good model. Hence, several decisions must be made to collect an appropriate dataset. How the dynamic system is excited has a significant influence on the range of different dynamic behaviors present in the collected dataset. Classical system identification textbooks offer some practical guidance in what concerns designing tests for system identification (Ljung, 1998; Aguirre, 2004).

For linear system identification, it is necessary to excite a wide range of system frequencies. Signals such as the pseudorandom binary sequence (PRBS) or filtered white Gaussian noise are usually appropriate choices for that purpose. For nonlinear models, there is a need to drive the system over a wider range of amplitudes, and signals that are appropriate for linear system identification may no longer be good choices. For instance, Leontaritis and Billings (1987) show that PRBS may be unsuitable for nonlinear system identification. In other situations, only “routine operation” data is available, and automatic procedures may be devised to find data windows containing relevant dynamical information, as in (Ribeiro and Aguirre, 2015).

Furthermore, the user should remove drifts, trends and seasonal variations, pre-filter the data and deal with missing points and outliers in order to obtain good results (Ljung, 1998, Chapter 14).

Part II

The trade-offs of recurrence

4 Parallel training considered harmful?

As discussed in the Introduction, the comparison between recurrent and feedforward structures takes many shapes and forms in the literature. [Narendra and Parthasarathy \(1990\)](#) have shown that neural network models for dynamic systems can be trained either in *parallel* or in *series-parallel* configurations. The parallel configuration incurs in a recurrent structure and the series-parallel, in a feedforward structure. Influenced by early arguments given in this landmark paper, several works justify the choice of series-parallel rather than parallel configuration claiming it has a lower computational cost, better stability properties during training and provides more accurate results.

Other published results, on the other hand, defend parallel training as being more robust and capable of yielding more accurate long-term predictions. The main contribution of this chapter is to present a study comparing both methods under the same unified framework and revisit some of the arguments presented in this landmark paper. We focus on three aspects: i) robustness of the estimation in the presence of noise; ii) computational cost; and, iii) convergence. A unifying mathematical framework and simulation studies show situations where each training method provides better validation results, being parallel training better in what is believed to be more realistic scenarios. An example using measured data seems to reinforce such claim. We also show, with a novel complexity analysis and numerical examples, that both methods have similar computational cost, being series series-parallel training, however, more amenable to parallelization. Some informal discussion about stability and convergence properties is presented and explored in the examples.

This material presented in this chapter is related with the following publication:

“Parallel Training Considered Harmful?” : Comparing Series-Parallel and Parallel Feedforward Network Training, Antônio H. Ribeiro, Luis A. Aguirre. *Neurocomputing*, 2018. v. 316 (17) pp. 222-231. doi: 10.1016/j.neucom.2018.07.071

The rest of the chapter is organized as follows: Section 4.1 introduces the difference between parallel and series-parallel training modes and contextualize the contribution of this chapter; Section 4.2 presents both training modes as *prediction error methods* ([Ljung, 1978, 1998](#)) and Section 4.3 formulates neural network training in both configurations as a nonlinear least-squares problem. It is a secondary contribution of this chapter to present both training methods under the same framework and the comparison in subsequent sections is built on top of this formulation. Section 4.4 presents a complexity analysis comparing the methods and Section 4.5 discusses signal unboundedness and the possibility of convergence to “bad” local solutions. In Section 4.6, numerical examples with measured and simulated data investigate the effect of the noise in the estimation and the running time of both methods. Final comments and future work ideas are presented in Section 4.7.

4.1 Introduction

Neural networks are widely used and studied for modeling nonlinear dynamic systems ([Narendra and Parthasarathy, 1990](#); [Zhang et al., 2006](#); [Singh et al., 2013](#); [Saad et al., 1994](#); [Beale et al., 2017](#); [Saggar et al., 2007](#); [Petrović et al., 2013](#); [Tijani et al., 2014](#); [Khan et al., 2015](#); [Diaconescu, 2008](#); [Warwick and Craddock, 1996](#); [Kamiński et al., 1996](#); [Rahman et al., 2000](#); [Su et al., 1992](#); [Su and McAvoy, 1993](#); [Aguirre et al., 2010](#); [Patan and Korbicz, 2012](#); [Wang et al., 2014, 2016](#)). In the seminal paper by

Narendra and Parthasarathy (Narendra and Parthasarathy, 1990) series-parallel and parallel configurations are introduced as possible neural network architectures for predicting the output of dynamic systems. Figure 4.1 illustrates the difference. *Parallel configuration* feeds the output back to the input of the network and, hence, uses its own previous values to try to predict the next output of the system being modeled. *Series-parallel configuration*, on the other hand, uses the true measured output rather than feeding back the estimated one.

Hence, when training the neural network in series-parallel configuration measured values from past instants are used to make *one-step-ahead* predictions. On the other hand, when training the neural network in parallel configuration the predicted output is computed by running a *free-run simulation* of the system through the entire window length. In both cases the error between predicted and measured values is minimized in order to estimate the neural network parameters.

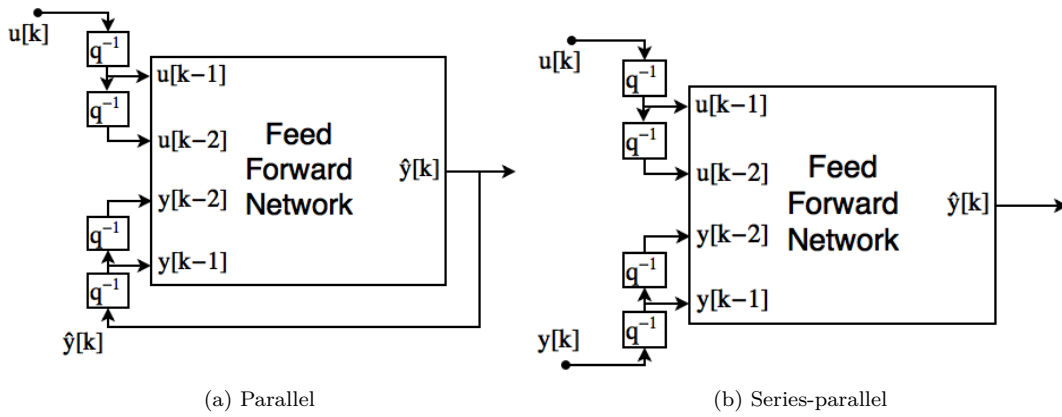


Figure 4.1 – **(Parallel and series-parallel)** Display neural network training modes. In both cases, considers the dynamic system model $\mathbf{y}[k] = \mathbf{F}(\mathbf{y}[k-1], \mathbf{y}[k-2], \mathbf{u}[k-1], \mathbf{u}[k-2])$. The block q^{-1} holds and delay the output by one sample period.

In (Narendra and Parthasarathy, 1990), training in series-parallel configuration is said to be preferred than parallel configuration for three main reasons, latter mentioned in several other works: i) all signals generated in the identification procedure for series-parallel configuration are bounded, while this is not guaranteed for the parallel configuration (Zhang et al., 2006; Singh et al., 2013); ii) for the parallel configuration a modified version of backpropagation is needed, resulting in a greater computational cost, while the standard backpropagation can be used for series-parallel configuration (Saad et al., 1994; Beale et al., 2017; Saggar et al., 2007); and, iii) assuming that the error tends asymptotically to small values the simulated output is not so different from the actual one and, therefore, the results obtained from two configurations would not be significantly different (Warwick and Craddock, 1996; Kamiński et al., 1996; Rahman et al., 2000). An additional reason that also appears sometimes in the literature is that: iv) the series-parallel training provides better results because of more accurate inputs to the neural network during training (Beale et al., 2017; Saggar et al., 2007; Petrović et al., 2013; Tijani et al., 2014; Khan et al., 2015; Diaconescu, 2008).

Results presented in other papers, however, show the strengths of parallel training: according to (Su et al., 1992; Su and McAvoy, 1993) neural network trained in parallel yield more accurate long-term predictions than the ones trained in series-parallel; (Aguirre et al., 2010) shows for diverse types of models, including neural networks, that the parallel training can be more robust than series-parallel training in the presence of some kinds of noise; and, in (Patan and Korbicz, 2012) neural network parallel models presents better validation results than series-parallel models for modeling a boiler unit. Furthermore, the free-run simulation error minimization (used in parallel training) seems to be successful when dealing

with other types of models: some state-of-the-art structure selection techniques for polynomial models are based on it (Piroddi and Spinelli, 2003; Farina and Piroddi, 2008, 2010, 2011, 2012); and, in (Zhang et al., 2014) it provided the best results when estimating the parameters of a battery.

The main contribution of this chapter is to compare these two training methods. We focus on three aspects: i) robustness of the estimation in the presence of noise; ii) computational cost; and, iii) convergence. Our findings suggest that parallel training may provide the most accurate models under some common circumstances. Furthermore, its computational cost is not significantly different than series-parallel training. We believe this to be relevant because it contradicts some of the frequently cited reasons for using series-parallel rather than parallel training.

4.2 Unifying framework

Consider the following data set: $\mathcal{Z} = \{(\mathbf{u}[k], \mathbf{y}[k]), k = 1, 2, \dots, N\}$, containing a sequence of sampled inputs-output pairs. Here $\mathbf{u}[k] \in \mathbb{R}^{N_u}$ and $\mathbf{y}[k] \in \mathbb{R}^{N_y}$ are vectors containing all the inputs and outputs of interest at instant k . The output $\mathbf{y}[k]$ is correlated with its own past values and with past input values. This work is trying to find a difference equation model $\mathbf{y}[k] = \hat{\mathbf{f}}(\mathbf{y}_{[k]}, \mathbf{u}_{[k]}; \hat{\boldsymbol{\theta}})$, for $\hat{\mathbf{f}}$ being a neural network.

In this section we present parallel and series-parallel training in the *prediction error methods* framework (Chapter 3). This analysis provides some insight on the type of situation each model is supposed to perform better.

4.2.1 Parallel vs series-parallel training

Let us define one-step-ahead prediction and free-run simulation:

Definition 4.1 (One-step-ahead prediction). *For a given function $\hat{\mathbf{f}}$, parameter vector $\hat{\boldsymbol{\theta}}$ and dataset \mathcal{Z} , the one-step-ahead prediction is defined as:*

$$\hat{\mathbf{y}}_1[k] = \hat{\mathbf{f}}(\mathbf{y}[k-1], \dots, \mathbf{y}[k-n_y], \mathbf{u}[k-\tau_d], \dots, \mathbf{u}[k-n_u]; \hat{\boldsymbol{\theta}}). \quad (4.1)$$

□

Definition 4.2 (Free-run simulation). *For a given function $\hat{\mathbf{f}}$, parameter vector $\hat{\boldsymbol{\theta}}$, dataset \mathcal{Z} , and a set of initial conditions $\{\mathbf{y}_0[k]\}_{k=1}^{n_y}$, the free-run simulation is defined using the recursive formula:*

$$\hat{\mathbf{y}}_s[k] = \begin{cases} \mathbf{y}_0[k], & 1 \leq k < n_y; \\ \hat{\mathbf{f}}(\hat{\mathbf{y}}_s[k-1], \dots, \hat{\mathbf{y}}_s[k-n_y], \mathbf{u}[k-\tau_d], \dots, \mathbf{u}[k-n_u]; \hat{\boldsymbol{\theta}}), & \\ n \geq n_y. \end{cases} \quad (4.2)$$

The vector of initial conditions is defined as $\mathbf{y}_0 = [\mathbf{y}_0[1]^T, \dots, \mathbf{y}_0[n_y-1]^T]^T$.

□

Let $\mathbf{e} = [\mathbf{e}[1]^T, \dots, \mathbf{e}[N]^T]^T$ be the prediction error vectors. The parameters are estimated minimizing $\frac{1}{2} \|\mathbf{e}\|^2$. In series-parallel it is used the one-step-ahead error, $\mathbf{e}_1[k] = \hat{\mathbf{y}}_1[k] - \mathbf{y}[k]$, while in parallel training the free-run simulation error is used, $\mathbf{e}_s[k] = \hat{\mathbf{y}}_s[k] - \mathbf{y}[k]$.

This can be put into the framework presented in Chapter 3 easily. Just consider the objective function defined as in Equation (3.8) for the prediction $\hat{\mathbf{y}}_{\boldsymbol{\theta}}$ given either by the one-step-ahead prediction $\hat{\mathbf{y}}_1[k]$ or by the free-run simulation $\hat{\mathbf{y}}_s[k]$. The minimization of $\frac{1}{2} \|\mathbf{e}\|^2$ is then equivalent to the minimization of V . In Chapter 3 we used the nomenclature *nonlinear ARX* to refer to the models obtained using series-parallel training; and *nonlinear output error* to refer to the models obtained using and parallel training.

4.2.2 Optimal predictor

If the measured values of \mathbf{y} and \mathbf{u} are known at all instants previous to k , the optimal prediction of $\mathbf{y}[k]$ is, as defined in Section 3.2.8.1, given by the following conditional expectation:

$$\hat{\mathbf{y}}_*[k] = E \left\{ \mathbf{y}[k] \mid \underline{\mathbf{y}}_{[k]}, \underline{\mathbf{u}}_{[k]} \right\}. \quad (4.3)$$

For the data being generated as in (3.19), consider two situations:

Situation 4.1 (White equation error). *The sequence of equation errors $\{\mathbf{v}[k]\}$ is a white noise process and the output error and error-in-variables are zero ($\mathbf{w}[k] = 0$ and $\mathbf{s}[k] = 0$).*

Situation 4.2 (White output error). *The sequence of output errors $\{\mathbf{w}[k]\}$ is a white noise process and the equation error and error-in-variables are zero ($\mathbf{v}[k] = 0$ and $\mathbf{s}[k] = 0$).*

For function, parameter vector and initial conditions matching the true ones than: if Situation 4.1 holds then the one-step-ahead prediction is equal to the optimal prediction ($\hat{\mathbf{y}}_1[k] = \hat{\mathbf{y}}_*[k]$); On the other hand, if Situation 4.2 holds then the free-run simulation is the optimal prediction ($\hat{\mathbf{y}}_s[k] = \hat{\mathbf{y}}_*[k]$).

Hence both training methods minimize an error that approaches the optimal predictor error $fie_*[k] = \hat{\mathbf{y}}_*[k] - \mathbf{y}[k]$ as the parameter approaches the true one. Furthermore, in both cases the prediction model containing the optimal prediction for one of the situations. Series-parallel training does it for Situation 4.1 and parallel training for Situation 4.2. Hence, each training method is more appropriate for one type of situation.

4.3 Nonlinear least-squares network training

Unlike other machine learning applications (e.g. natural language processing and computer vision) where there are enormous datasets available to train neural network models, the datasets available for *system identification* are usually of moderate size. The available data is usually obtained through tests with limited duration because of practical and economical reasons. And, even when there is a long record of input-output data, it either does not contain meaningful dynamic behavior (Ribeiro and Aguirre, 2015) or the system cannot be considered time-invariant over the entire record, resulting in the necessity of selecting smaller portions of this longer dataset for training.

Due to the unavailability of large datasets, the use of neural networks in system identification is usually restricted to neural networks with few hundred weights. The Levenberg-Marquardt method does provide a fast convergence rate (Nocedal and Wright, 2006) and has been described as very efficient for batch training of moderate size problems (Hagan and Menhaj, 1994), where the memory used by this algorithm is not prohibitive. Hence, it will be the method of choice for training neural networks in this chapter.

Besides that, recurrent neural networks often present vanishing gradients that may prevent the progress of the optimization algorithm. The use of second order information (as in the Levenberg-Marquardt algorithm) help to mitigate this problem (Bengio et al., 1994).

This section presents the parallel and series-parallel training as nonlinear least-squares problems. Sections 4.3.1 and 4.3.2 give some background in the the Levenberg-Marquardt algorithm and in the backpropagation algorithm proposed by (Hagan and Menhaj, 1994). Series-parallel and parallel training are discussed in Sections 4.3.3 and 4.3.4. The backpropagation can be directly applied to series-parallel training, while for parallel training we introduce a new formula for computing the derivatives. This formula can be interpreted either as a variation of the dynamic backpropagation (Narendra and Parthasarathy,

1990) adapted to compute the Jacobian instead of the gradient; or, as a specific case of real-time recurrent learning (Williams and Zipser, 1989) with a special type of recurrent connection.

4.3.1 Nonlinear least-squares

Let $\boldsymbol{\theta} \in \mathbb{R}^{N_\theta}$ be a vector of parameters *containing all neural network weights and bias* and $\mathbf{e}(\boldsymbol{\theta}) \in \mathbb{R}^{N_e}$ an error vector. In order to estimate the parameter vector $\boldsymbol{\theta}$ the sum of square errors $V(\boldsymbol{\theta}) = \frac{1}{2} \|\mathbf{e}(\boldsymbol{\theta})\|^2$ is minimized. Its gradient vector and Hessian matrix may be computed as: (Nocedal and Wright, 2006, p. 246)

$$\frac{\partial V}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \right]^T \mathbf{e}(\boldsymbol{\theta}), \quad (4.4)$$

$$\frac{\partial^2 V}{\partial \boldsymbol{\theta}^2} = \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \right]^T \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \right] + \sum_{i=1}^{N_e} e_i \frac{\partial^2 e_i}{\partial \boldsymbol{\theta}^2}. \quad (4.5)$$

where $\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \in \mathbb{R}^{N_e \times N_\theta}$ is the Jacobian matrix associated with $\mathbf{e}(\boldsymbol{\theta})$. Non-linear least-squares algorithms usually update the solution iteratively ($\boldsymbol{\theta}^{n+1} = \boldsymbol{\theta}^n + \Delta \boldsymbol{\theta}^n$) and exploit the special structure of the gradient and Hessian of $V(\boldsymbol{\theta})$, in order to compute the parameter update $\Delta \boldsymbol{\theta}^n$.

The Levenberg-Marquardt algorithm considers a parameter update (Marquardt, 1963):

$$\Delta \boldsymbol{\theta}^n = - \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n^T \frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n + \lambda^n D^n \right]^{-1} \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n \right]^T \mathbf{e}_n, \quad (4.6)$$

for which λ^n is a non-negative scalar and D^n is a non negative diagonal matrix. Furthermore, \mathbf{e}_n and $\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n$ are the error and its Jacobian matrix evaluated at $\boldsymbol{\theta}^n$

There are different ways of updating λ^n and D^n . The update strategy presented here is similar to (Fletcher et al., 1971). The elements of the diagonal matrix D^n are chosen equal to the elements in the diagonal of $\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n^T \frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n$. And λ^n is increased or decreased according to the agreement between the local model $\left(\phi_n(\Delta \boldsymbol{\theta}^n) = \frac{1}{2} \|\mathbf{e}_n + \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n \right] \Delta \boldsymbol{\theta}^n \|^2 \right)$ and the real objective function $V(\boldsymbol{\theta}_n)$. The degree of agreement is measured using the following ratio:

$$\rho_n = \frac{V(\boldsymbol{\theta}^n) - V(\boldsymbol{\theta}^n + \Delta \boldsymbol{\theta}^n)}{\phi_n(\mathbf{0}) - \phi_n(\Delta \boldsymbol{\theta}^n)}. \quad (4.7)$$

One iteration of the algorithm is summarized next:

ALGORITHM 4.1 (LEVENBERG-MARQUARDT ITERATION). For a given $\boldsymbol{\theta}^n$ and λ^n :

1. Compute $\mathbf{e}(\boldsymbol{\theta}^n)$ and $\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^n)$, if not already computed.
2. $\text{diag}(D^n) \leftarrow \text{diag} \left(\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n^T \frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} \Big|_n \right)$.
3. Solve (4.6) and compute $\Delta \boldsymbol{\theta}^n$.
4. Compute ρ_n as in (4.7).
5. $\lambda^{n+1} \leftarrow 4\lambda^n$ if $\rho_n > \frac{3}{4}$; $\lambda^{n+1} \leftarrow \frac{1}{2}\lambda^n$ if $\rho_n < \frac{1}{4}$; otherwise, $\lambda^{n+1} \leftarrow \lambda^n$.
6. $\boldsymbol{\theta}^{n+1} \leftarrow \boldsymbol{\theta}^n + \Delta \boldsymbol{\theta}^n$ if $\rho_n > 10^{-3}$; otherwise, $\boldsymbol{\theta}^{n+1} \leftarrow \boldsymbol{\theta}^n$.
7. $n = n + 1$.

□

4.3.2 Modified backpropagation

Consider a multi-layer feedforward network, such as the three-layer network in Figure 4.2. This network can be seen as a function that relates the input $\mathbf{x} \in \mathbb{R}^{N_x}$ to the output $\mathbf{z} \in \mathbb{R}^{N_z}$. The parameter vector $\boldsymbol{\theta}$ contains all weights $w_{i,j}^{(n)}$ and bias $\gamma_i^{(n)}$ of the network. This subsection presents a modified version of backpropagation (Hagan and Menhaj, 1994) for computing the neural network output \mathbf{z} and its Jacobian matrix for a given input \mathbf{x} . The notation used is the one displayed in Figure 4.2.

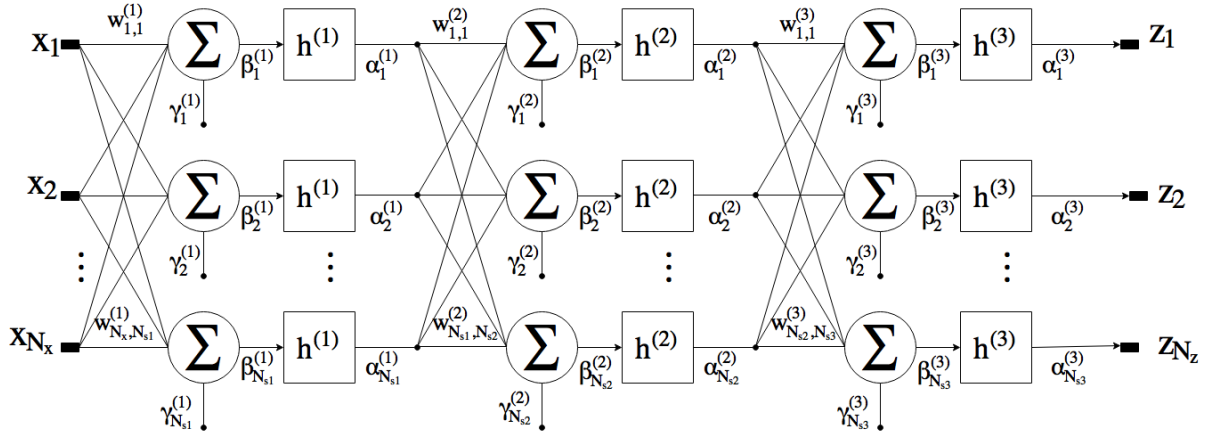


Figure 4.2 – (Fully connected neural network) Display three-layer feedforward network.

1. **Forward Stage:** For an \mathcal{L} layer network the output nodes can be computed using the following recursive matrix relation:

$$\boldsymbol{\alpha}^{(n)} = \begin{cases} \mathbf{x} & n = 0; \\ \mathbf{h}^{(n)}(W^{(n)}\boldsymbol{\alpha}^{(n-1)} + \boldsymbol{\gamma}^{(n)}) & n = 1, \dots, \mathcal{L}, \end{cases} \quad (4.8)$$

where, for the n -th layer, $W^{(n)}$ is a matrix containing the weights $w_{i,j}^{(n)}$, $\boldsymbol{\gamma}^{(n)}$ is a vector containing the bias $\gamma_i^{(n)}$ and $\mathbf{h}^{(n)}$ applies the nonlinear function $h^{(n)}$ element-wise. The output \mathbf{z} is given by:

$$\mathbf{z} = \boldsymbol{\alpha}^{(\mathcal{L})}. \quad (4.9)$$

2. **Backward Stage:** The follow recurrence relation can be used to compute $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(n)}}$ for every n :

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(n)}} = \begin{cases} \dot{H}^{(\mathcal{L})}(\boldsymbol{\beta}^{(\mathcal{L})}) & n = \mathcal{L}; \\ \frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(n+1)}} \cdot W^{(n+1)} \cdot \dot{H}^{(n)}(\boldsymbol{\beta}^{(n)}) & n = \mathcal{L} - 1, \dots, 1, \end{cases} \quad (4.10)$$

where $\dot{H}^{(n)}$ is given by the following diagonal matrix:

$$\dot{H}^{(n)}(\boldsymbol{\beta}^{(n)}) = \text{diag}(\dot{h}^{(n)}(\beta_1^{(n)}), \dots, \dot{h}^{(n)}(\beta_{N_{sk}}^{(n)})).$$

The recursive expression (4.10) follows from applying the chain rule $\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(n)}} = \frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(n+1)}} \frac{\partial \boldsymbol{\beta}^{(n+1)}}{\partial \boldsymbol{\alpha}^{(n)}} \frac{\partial \boldsymbol{\alpha}^{(n)}}{\partial \boldsymbol{\beta}^{(n)}}\right)$, and considering $\frac{\partial \boldsymbol{\beta}^{(n+1)}}{\partial \boldsymbol{\alpha}^{(n)}} = W^{(n+1)}$ and $\frac{\partial \boldsymbol{\alpha}^{(n)}}{\partial \boldsymbol{\beta}^{(n)}} = \dot{H}^{(n)}$.

3. **Computing Derivatives:** The derivatives of \mathbf{z} in relation to the weights $w_{i,j}^{(n)}$ and the bias $\gamma_i^{(n)}$ can be used to form the Jacobian matrix $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\theta}}$ and can be computed using the following expressions:

$$\frac{\partial \mathbf{z}}{\partial w_{i,j}^{(n)}} = \frac{\partial \mathbf{z}}{\partial \beta_i^{(n)}} \frac{\partial \beta_i^{(n)}}{\partial w_{i,j}^{(n)}} = \frac{\partial \mathbf{z}}{\partial \beta_i^{(n)}} \alpha_j^{(n-1)}; \quad (4.11)$$

$$\frac{\partial \mathbf{z}}{\partial \gamma_i^{(n)}} = \frac{\partial \mathbf{z}}{\partial \beta_i^{(n)}} \frac{\partial \beta_i^{(n)}}{\partial \gamma_i^{(n)}} = \frac{\partial \mathbf{z}}{\partial \beta_i^{(n)}}. \quad (4.12)$$

Furthermore, the derivatives of \mathbf{z} in relation to the inputs \mathbf{x} are:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(1)}} \frac{\partial \boldsymbol{\beta}^{(1)}}{\partial \boldsymbol{\alpha}^{(0)}} = \frac{\partial \mathbf{z}}{\partial \boldsymbol{\beta}^{(1)}} W^{(1)}. \quad (4.13)$$

4.3.3 Series-parallel training

In the series-parallel configuration the parameters are estimated by minimizing $\frac{1}{2} \|\mathbf{e}_1\|^2$, what can be done using the algorithm described in Section 4.3.1. The required Jacobian matrix $\frac{\partial \mathbf{e}_1}{\partial \boldsymbol{\theta}}$ can be computed according to the following well known result.

Lemma 4.1. *The Jacobian matrix of \mathbf{e}_1 in relation to $\boldsymbol{\theta}$ is $\frac{\partial \mathbf{e}_1}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathbf{e}_1[1]}{\partial \boldsymbol{\theta}}^T, \dots, \frac{\partial \mathbf{e}_1[N]}{\partial \boldsymbol{\theta}}^T \right]^T$, where $\frac{\partial \mathbf{e}_1[k]}{\partial \boldsymbol{\theta}} = \frac{\partial \hat{\mathbf{y}}_1[k]}{\partial \boldsymbol{\theta}} = \frac{\partial \hat{\mathbf{f}}}{\partial \boldsymbol{\theta}}(\mathbf{y}_{[k]}, \mathbf{u}_{[k]}; \hat{\boldsymbol{\theta}})$ that can be computed using the backpropagation described at Section 4.3.2.*

Proof. Results from differentiating (4.1). □

4.3.4 Parallel training

In the parallel configuration the parameters are estimated by minimizing $\frac{1}{2} \|\mathbf{e}_s\|^2$. There are two different ways to take into account the initial conditions \mathbf{y}_0 , they are: (i) to fix the initial conditions \mathbf{y}_0 and estimate the model parameters $\boldsymbol{\theta}$; and, (ii) to define an extended parameter vector $\boldsymbol{\Phi} = [\boldsymbol{\theta}^T, \mathbf{y}_0^T]^T$ and estimate \mathbf{y}_0 simultaneously with $\boldsymbol{\theta}$.

When using formulation (i), a suitable choice is to set the initial conditions equal to the measured outputs ($\mathbf{y}_0[k] = \mathbf{y}[k]$, $k = 1, \dots, n_y - 1$). When using formulation (ii) the measured outputs may be used as an initial guess to be refined by the optimization algorithm.

The optimal choice for the initial condition would be $\mathbf{y}_0[k] = \mathbf{y}^*[k]$ for $n = 1, \dots, n_y - 1$. Formulation (i) uses the non-optimal choice $\mathbf{y}_0[k] = \mathbf{y}[k] \neq \mathbf{y}^*[k]$. Formulation (ii) goes one step further and include the initial conditions $\mathbf{y}_0[k]$ in the optimization problem, so it converges to $\mathbf{y}^*[k]$ and hence improves the parameter estimation.

The Jacobian matrices $\frac{\partial \mathbf{e}_s}{\partial \boldsymbol{\theta}}$ and $\frac{\partial \mathbf{e}_s}{\partial \mathbf{y}_0}$ can be computed according to the following lemma.

Lemma 4.2. *The Jacobian matrices of \mathbf{e}_1 in relation to $\boldsymbol{\theta}$ and \mathbf{y}_0 are $\frac{\partial \mathbf{e}_s}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathbf{e}_s[1]}{\partial \boldsymbol{\theta}}^T, \dots, \frac{\partial \mathbf{e}_s[N]}{\partial \boldsymbol{\theta}}^T \right]^T$ and $\frac{\partial \mathbf{e}_s}{\partial \mathbf{y}_0} = \left[\frac{\partial \mathbf{e}_s[1]}{\partial \mathbf{y}_0}^T, \dots, \frac{\partial \mathbf{e}_s[N]}{\partial \mathbf{y}_0}^T \right]^T$ where $\frac{\partial \mathbf{e}_s[k]}{\partial \boldsymbol{\theta}} = \frac{\partial \hat{\mathbf{y}}_s[k]}{\partial \boldsymbol{\theta}}$ and $\frac{\partial \mathbf{e}_s[k]}{\partial \mathbf{y}_0} = \frac{\partial \hat{\mathbf{y}}_s[k]}{\partial \mathbf{y}_0}$ can be computed according to the following recursive formulas:*

$$\frac{\partial \hat{\mathbf{y}}_s[k]}{\partial \boldsymbol{\theta}} = \begin{cases} \mathbf{0}, & 1 \leq k \leq n_y - 1; \\ \frac{\partial \hat{\mathbf{f}}}{\partial \boldsymbol{\theta}}(\hat{\mathbf{y}}_{[k]}, \mathbf{u}_{[k]}; \hat{\boldsymbol{\theta}}) + \sum_{i=1}^{n_y} \frac{\partial \hat{\mathbf{f}}}{\partial \mathbf{y}[k-i]}(\hat{\mathbf{y}}_{[k]}, \mathbf{u}_{[k]}; \hat{\boldsymbol{\theta}}) \frac{\partial \hat{\mathbf{y}}_s[k-i]}{\partial \boldsymbol{\theta}}, & n \geq n_y, \end{cases} \quad (4.14)$$

$$\frac{\partial \hat{\mathbf{y}}_s[k]}{\partial \mathbf{y}_0} = \begin{cases} D^{(n)}, & 1 \leq k \leq n_y - 1; \\ \sum_{i=1}^{n_y} \frac{\partial \hat{\mathbf{f}}}{\partial \mathbf{y}[k-i]}(\hat{\mathbf{y}}_{[k]}, \mathbf{u}_{[k]}; \hat{\boldsymbol{\theta}}) \frac{\partial \hat{\mathbf{y}}_s[k-i]}{\partial \mathbf{y}_0}, & n \geq n_y, \end{cases} \quad (4.15)$$

where $D^{(n)} \in \mathbb{R}^{N_y \times n_y N_y}$ is defined as:

$$\{D^{(n)}\}_{i,j} = \begin{cases} 1, & \text{if } j = (n-1) \cdot N_y + i, \\ 0, & \text{otherwise.} \end{cases} \quad (4.16)$$

Proof. Results from differentiating (4.2) and applying the chain rule. □

4.4 Complexity analysis

In this section we present a novel complexity analysis comparing series-parallel training (SP), parallel training with fixed initial conditions ($P\theta$) and parallel training with extended parameter vector ($P\Phi$). We show that the training methods have similar computational cost for the nonlinear least-squares formulation. The number of floating point operations (*flops*) is estimated based on (Golub and Van Loan, 2012, Table 1.1.2). Low order terms, as usual, are neglected in the analysis.

4.4.1 Neural network output and its partial derivatives

The backpropagation algorithm described in Section 4.3.2 can be used for training both fully or partially connected networks. What would have to change is the internal representation of the weight matrices $W^{(n)}$: for a partially connected network the matrices would be stored using a sparse representation, e.g. compressed sparse column (CSC) representation.

The total number of *flops* required to evaluate the output and to compute its partial derivatives for the feedforward network is summarized in Table 4.1 considering a fully connected network. The total number of network weights is denoted as N_w and the total number of bias terms as N_γ , such that $N_w + N_\gamma = N_\Theta$. For this fully connected network:

$$\begin{aligned} N_w &= N_x \cdot N_{s1} + N_{s1} \cdot N_{s2} + \dots + N_{s(\mathcal{L}-1)} \cdot N_{s\mathcal{L}}, \\ N_\gamma &= N_{s1} + N_{s2} + \dots + N_{s\mathcal{L}}. \end{aligned} \quad (4.17)$$

Table 4.1 – **(Backpropagation computational cost)** Modified backpropagation number of *flops* for a fully connected network.

Computing Neural Network Output	
i) Compute $\hat{\mathbf{f}}(\mathbf{x}; \theta)$ — Eq. (4.8)-(4.9)	$2N_w$
Computing Partial Derivatives	
ii) Backward Stage — Eq. (4.10)	$(2N_z + 1)(N_w - N_x N_{s1})$
iii) Compute $\frac{\partial \hat{\mathbf{f}}}{\partial \theta}$ — Eq. (4.11)-(4.12)	$N_w \cdot N_z$
iv) Compute $\frac{\partial \hat{\mathbf{f}}}{\partial \mathbf{x}}$ — Eq. (4.13)	$2N_x \cdot N_{s1} \cdot N_z$

Since the more relevant terms of the complexity analysis in Table 4.1 are being expressed in terms of N_w and N_z the results for a fully connected network are similar to the ones that would be obtained for a partially connected network using a sparse representation.

4.4.2 Number of flops for series-parallel and parallel training

The number of flops of each iteration of the Levenberg-Marquardt algorithm is summarized in Table 4.2. Entries (i) to (iv) in Table 4.2 follow directly from Table 4.1, considering $N_z = N_y$ and multiplying the costs by N because of the number of different inputs being evaluated. Furthermore, in entries (v) and (vi), it is considered that the evaluation of (4.14) and (4.15) is done, not by recomputing the entire summation each time, but by storing each computed values and computing only one new matrix-matrix product per evaluation.

The cost of solving (4.6) is about $2 \cdot N \cdot N_\Theta^2 + \frac{1}{3} \cdot N_\Theta^3$ where the cost $2N \cdot N_\Theta^2$ is due to the multiplication of the Jacobian matrix by its transpose and $\frac{1}{3}N_\Theta^3$ is due to the needed Cholesky factorization. For an extended parameter vector N_Θ is replaced by N_Φ in the analysis.

Table 4.2 – **(Computational cost)** Levenberg-Marquardt number of *flops* per iteration for series-parallel training (SP), parallel training with fixed initial conditions ($\mathbf{P}\boldsymbol{\theta}$) and parallel training with extended parameter vector ($\mathbf{P}\boldsymbol{\Phi}$). A mark \times signals which calculation is required in each method.

		SP	$\mathbf{P}\boldsymbol{\theta}$	$\mathbf{P}\boldsymbol{\Phi}$
Computing Error				
i) Compute $\hat{\mathbf{f}}(\mathbf{x}; \boldsymbol{\theta})$	$2N \cdot N_w$	\times	\times	\times
Computing Partial Derivatives				
ii) Backward Stage	$N(2N_y + 1)(N_w - N_x N_{s1})$	\times	\times	\times
iii) Compute $\frac{\partial \mathbf{f}}{\partial \boldsymbol{\theta}}$	$N \cdot N_w \cdot N_y$	\times	\times	\times
iv) Compute $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$	$2N \cdot N_x \cdot N_{s1} \cdot N_y$		\times	\times
v) Equation (4.14)	$2N \cdot N_{\Theta} \cdot (N_y^2 + N_y)$		\times	\times
vi) Equation (4.15)	$2N \cdot n_y \cdot N_y \cdot (N_y^2 + N_y)$			\times
Solving Equation (4.6)				
vii) Solve (4.6) — $\boldsymbol{\theta}$	$2N \cdot N_{\Theta}^2 + \frac{1}{3}N_{\Theta}^3$	\times	\times	
viii) Solve (4.6) — $\boldsymbol{\Phi}$	$2N \cdot N_{\Phi}^2 + \frac{1}{3}N_{\Phi}^3$			\times

4.4.3 Comparing methods

Assuming the number of nodes in the last hidden layer is greater than the number of outputs ($N_y < N_{s(\mathcal{L}-1)}$), the inequalities follow directly from this chapter definitions:

$$\begin{aligned} n_y \cdot N_y &\leq N_x < N_x \cdot N_{s1} \leq N_w < N_{\Theta}; \\ N_y < N_y^2 < N_y \cdot N_{s(\mathcal{L}-1)} &\leq N_w < N_{\Theta}. \end{aligned} \quad (4.18)$$

From Table 4.2 and from the above inequalities it follows that the cost of each Levenberg-Marquardt iteration is dominated by the cost of solving Equation (4.6). Furthermore, $N_{\Phi} = N_{\Theta} + n_y \cdot N_y < 2N_{\Theta}$, therefore, $\mathbf{P}\boldsymbol{\Phi}$ training method has the same asymptotic computational cost of SP and $\mathbf{S}\boldsymbol{\theta}$ methods: $\mathcal{O}(N \cdot N_{\Theta}^2 + N_{\Theta}^3)$.

From Table 4.2 it is also possible to analyze the cost of each of the major steps needed in each full iteration of the algorithm:

- **Computing Error:** The cost of computing the error is the same for all of the training methods.
- **Computing Partial Derivative:** The computation of partial derivatives has a cost of $\mathcal{O}(N \cdot N_w \cdot N_y)$ for the SP training method and a cost $\mathcal{O}(N \cdot N_{\Theta} \cdot N_y^2)$ for both $\mathbf{P}\boldsymbol{\Phi}$ and $\mathbf{P}\boldsymbol{\theta}$. For many cases of interest in system identification, the number of model outputs N_y is small. Furthermore, $N_{\Theta} = N_w + N_{\gamma} \approx N_w$ (see Eq. (4.17)). That is why the cost of computing the partial derivatives for parallel training is comparable to the cost for series-parallel training.
- **Solving Equation (4.6):** It already has been established that the cost of this step $\mathcal{O}(N \cdot N_{\Theta}^2 + N_{\Theta}^3)$ dominates the computational cost for all the training methods. Furthermore $n_y \cdot N_y$ is usually much smaller than N_{Θ} such that $N_{\Phi} \approx N_{\Theta}$ and the number of flops of this stage is basically the same for all the training methods.

4.4.4 Memory complexity

Considering that N is significantly larger than n_y , it follows that, for the three training methods, the storage capacity is dominated by the storage of the Jacobian matrix $\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}}$ or of the matrix resulting from

the product $\left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}}\right]^T \left[\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}}\right]$. Therefore the size of the memory used by the algorithm is about $\mathcal{O}(\max(N \cdot N_y \cdot N_{\Theta}, N_{\Theta}^2))$.

For very large datasets and a large number of parameters, this storage requirements may be prohibitive and others methods should be used (e.g. stochastic gradient descent or variations). Nevertheless, for datasets of moderate size and networks with few hundred parameters, as it is usually the case for system identification, the use of nonlinear least-squares is a viable option.

4.5 Practical aspects

4.5.1 Convergence towards a local minima

The optimization problem that results from both series-parallel and parallel training of neural networks are non-convex and may have multiple local minima. The solution of the Levenberg-Marquardt algorithm discussed in this chapter, as well as most algorithms of interest for training neural networks (e.g. stochastic gradient descent, L-BFGS¹, conjugate gradient), converges towards a local minima². However, there is no guarantee for neither series-parallel nor parallel training that solution found is the global optimum. The convergence to “bad” local solutions may happen for both training methods, however, as illustrated in the numerical examples, it seems to happen more often for parallel training.

4.5.2 Signal unboundedness during training

Signals obtained in intermediary steps of parallel identification may become unbounded. The one-step-ahead predictor, used in series-parallel training, is always bounded since the prediction depends only on measured values – it has a FIR (Finite Impulse Response) structure – while, for the parallel training, the free-run simulation could be unbounded for some choice of parameters because of its dependence on its own past simulation values.

This is a potential problem because during an intermediary stage of parallel training a choice of $\boldsymbol{\theta}^k$ that results in unbounded values of $\hat{\mathbf{y}}_s[k]$ may need to be evaluated, resulting in overflows. Hence, optimization algorithms that may work well minimizing one-step-ahead prediction errors \mathbf{e}_1 , may fail when minimizing simulation errors \mathbf{e}_s .

For instance, steepest descent algorithms with a fixed step size may, for a poor choice of step size, fall into a region in the parameter space for which the signals are unbounded and the computation of the gradient will suffer from overflow and prevent the algorithm from continuing (since it does not have a direction to follow). This may also happen in more practical line search algorithms (e.g. the one described in (Nocedal and Wright, 2006, Sec. 3.5)).

The Levenberg-Marquardt algorithm, on the other hand, is robust against this kind of problem because every step $\boldsymbol{\theta}^n + \Delta \boldsymbol{\theta}^n$ that causes overflow in the objective function computation yields a negative ρ_n ³, hence the step is rejected by the algorithm and λ_n is increased. The increase in λ_n causes the length of $\Delta \boldsymbol{\theta}^n$ to decrease⁴. Therefore, the step length is decreased until a point is found close enough to the current iteration such that overflow does not occur. Hence, the Levenberg-Marquardt algorithm does not fail or stall due to overflows. Similar reasoning could be used for any trust-region method or for backtracking line-search.

¹ The *Limited-memory Broyden–Fletcher–Goldfarb–Shanno* (L-BFGS) algorithm.

² It is proved in (Moré, 1978) that the Levenberg-Marquardt (not exactly the one discussed here) converges towards a local minima or a stationary point under simple assumptions.

³ Programming languages as Matlab, C, C++ and Julia returns the floating point value encoded for infinity when an overflow occur. In this case formula (4.7) yields a negative ρ_n .

⁴ This inverse relation between λ_n and $\|\Delta \boldsymbol{\theta}^n\|$ is explained in (Nocedal and Wright, 2006).

Regardless of the optimization algorithm, signal unboundedness is not a problem for feedforward networks with bounded activation functions (e.g. Logistic or Hyperbolic Tangent) in the hidden layers, because its output is always bounded. Hence parallel training of these particular neural networks is not affected by the previously mentioned difficulties.

4.5.3 Time series prediction

It is important to highlight that parallel training is inappropriate to train neural networks for predicting time series in general. That is, parallel configuration is generally inadequate for the case when there are no inputs ($N_u = 0$).

For any asymptotically stable system in parallel configuration, the absence of inputs would make the free-run simulation converge towards an equilibrium, and even models that should be capable of providing good predictions for a few steps-ahead in the time series, may present poor performance for the entire training window. Hence, minimizing $\|e_s\|^2$ will not provide good results.

It still makes sense to use series-parallel training for time series models. That is because, feeding real outputs from the systems into the neural network keeps it from converging towards zero (for asymptotically stable systems) and makes the estimation robust against unknown disturbances affecting the time series. An interesting approach that mixes parallel and series-parallel training for time series prediction is given in (Menezes and Barreto, 2008).

4.6 Implementation and test results

The implementation is in Julia and runs on a computer with a processor Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz. For all examples in this chapter, the activation function used in hidden layers is the hyperbolic tangent, the weights $w_{i,j}^{(n)}$ initial values are drawn from a normal distribution with standard deviation $\sigma = (N_{s(n)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros (LeCun et al., 1998). Furthermore, in all parallel training examples we include the initial conditions as parameters of the optimization process (PΦ training).

The free run simulation mean square error ($\text{MSE} = \sum_n (y[k] - \hat{y}_s[k])^2$) is used as a goodness of fit measure to compare the models in the validation window.

The first example compares the training method using data from a real process and the second one investigates different noise configurations on computer generated data. The code and data used in the numerical examples are available in the GitHub repository: <https://github.com/antonior92/ParallelTrainingNN.jl>.

4.6.1 Example 1: data from a pilot plant

In this example, input-output signals were collected from *LabVolt Level Process Station* (model 3503-MO (LabVolt, 2015)). This plant consists of a tank that is filled with water from a reservoir. The water is pumped at fixed speed, while the flow is regulated using a pneumatically operated control valve driven by a voltage u . The water column height y is indirectly measured using a pressure sensor at the bottom of the tank. Figure 4.3 shows the free-run simulation in the validation window of models obtained for this process using parallel and series-parallel training.

Since the parameters initial guess are randomly initialized, different realizations will yield different results. Figure 4.4 shows the validation errors for both training methods for randomly drawn initial guesses. While parallel training consistently provides models with better validation results than series-parallel training, it also has some outliers with very bad validation results. We interpret these outliers in the

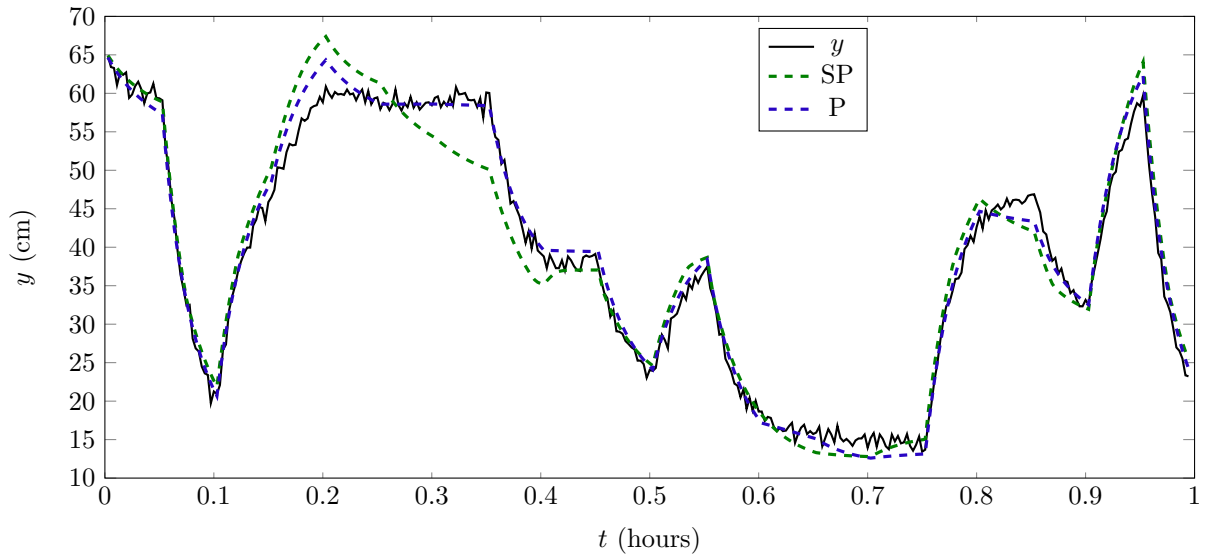


Figure 4.3 – **(Pilot plant validation)** Displays free-run simulation in the validation window for models obtained using series-parallel (SP) and parallel (P) training. The mean square errors are $\text{MSE}_{\text{SP}} = 1144.6$; $\text{MSE}_{\text{P}} = 296.2$. The models have $n_y = n_u = 1$ and 10 nodes in the hidden layer and were trained on a two hour long dataset sampled at $T_s = 10s$. The same initial parameter guess was used for both training methods. The training was 100 epochs long, which took 3.3 and 3.9 seconds, respectively, for series-parallel and parallel training.

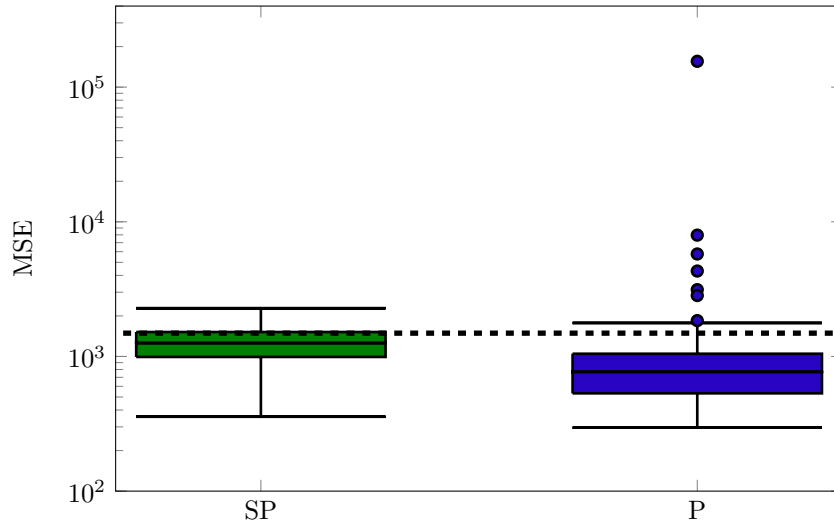


Figure 4.4 – **(Series-parallel vs parallel training)** Boxplots show the distribution of the free-run simulation MSE over the validation window for models trained using series-parallel (SP) and parallel (P) methods under the circumstances specified in Figure 4.3. There are 100 realizations of the training in each boxplot, in each realization the weights $w_{i,j}^{(n)}$ are drawn from a normal distribution with standard deviation $\sigma = (N_{s(n-1)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros (LeCun et al., 1998). For comparison purposes, the dashed horizontal line gives the performance of an ARX linear model ($n_y = 1$ and $n_u = 1$) trained and tested under the same conditions.

boxplot as consequence of the parallel training getting trapped in “bad” local minima, as mentioned in Section 4.5.

The training of the neural network was performed using normalized data. However, if *unscaled data* were used instead, parallel training would yield models with $\text{MSE} > 10000$ over the validation window while series-parallel training can still yield solutions with a reasonable fit to the validation data. We understand this as another indicator of parallel training greater sensitivity to the initial parameter guess: for unscaled data, the initial guess is far away from meaningful solutions of the problem, and, while series-parallel training converges to acceptable results, parallel training gets trapped in “bad” local solutions.

4.6.2 Example 2: investigating the noise effect

The non-linear system: (Chen et al., 1990b)

$$\begin{aligned} y^*[k] &= (0.8 - 0.5e^{-y^*[k-1]^2})y^*[k-1] - (0.3 + 0.9e^{-y^*[k-1]^2})y^*[k-2] + u[k-1] + \\ &\quad 0.2u[k-2] + 0.1u[k-1]u[k-2] + v[k] \\ y[k] &= y^*[k] + w[k], \end{aligned} \quad (4.19)$$

was simulated and the generated dataset was used to build neural network models. Figure 4.5 shows the validation results for models obtained for a training set generated with white Gaussian equation and output errors v and w . In this section, we repeat this same experiment for diverse random processes applied to v and w in order to investigate how different noise configurations affect parallel and series-parallel training.

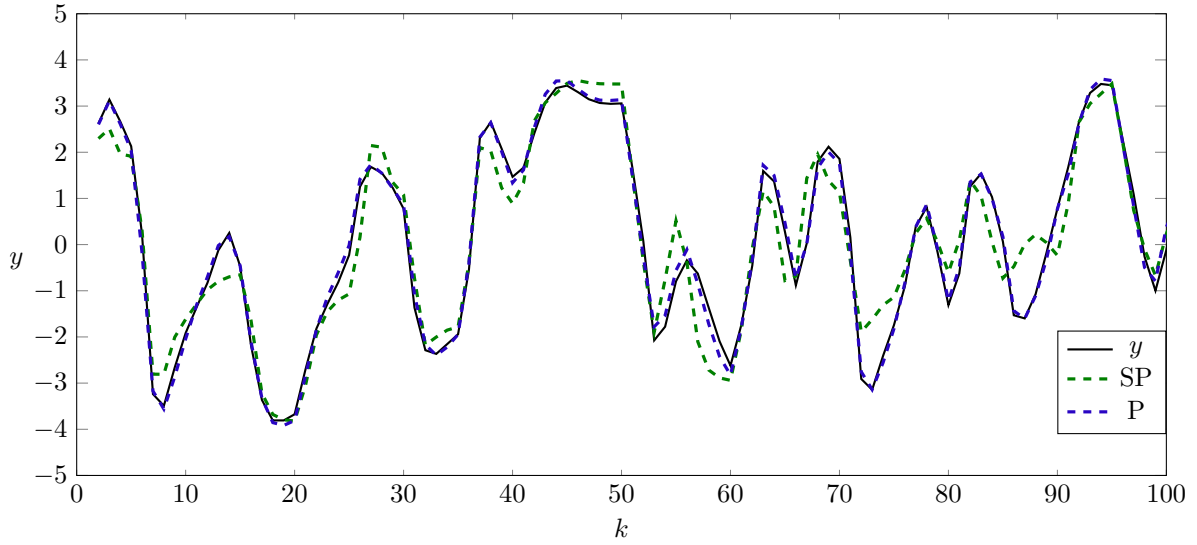


Figure 4.5 – **(Toy problem validation)** Displays the first 100 samples of the free-run simulation in the validation window for models trained using series-parallel (SP) and parallel (P) methods. The mean square errors are $\text{MSE}_{\text{SP}} = 0.39$; $\text{MSE}_{\text{P}} = 0.06$. The models have $n_y = n_u = 2$ and a single hidden layer with 10 nodes. The training set has $N = 1000$ samples and was generated with (7.6) for v and w white Gaussian noise with standard deviations $\sigma_v = 0.1$ and $\sigma_w = 0.5$. The validation window is generated without the noise effect. For both, the input u is randomly generated with standard Gaussian distribution, each randomly generated value held for 5 samples. The training was 100 epochs long, which took 5.0 and 6.1 seconds for, respectively, series-parallel and parallel training.

Let v be white Gaussian noise with standard deviation σ_v and let w be zero. Figure 4.6 (a) shows the free-run simulation error on the validation window using parallel or series-parallel training for

increasing values of σ_v . Figure 4.6 (b) shows the complementary experiment, for which v is zero and w is white Gaussian noise with increasing larger values of σ_w being tried out.

In Section 4.2, series-parallel training was derived considering only the presence of white equation error and, in this situation, the numerical results illustrate the model using this training method presents the best validation results (Figure 4.6 (a)). On the other hand, parallel training was derived considering only the presence of white output error and is significantly better in this alternative situation (Figure 4.6 (b)).

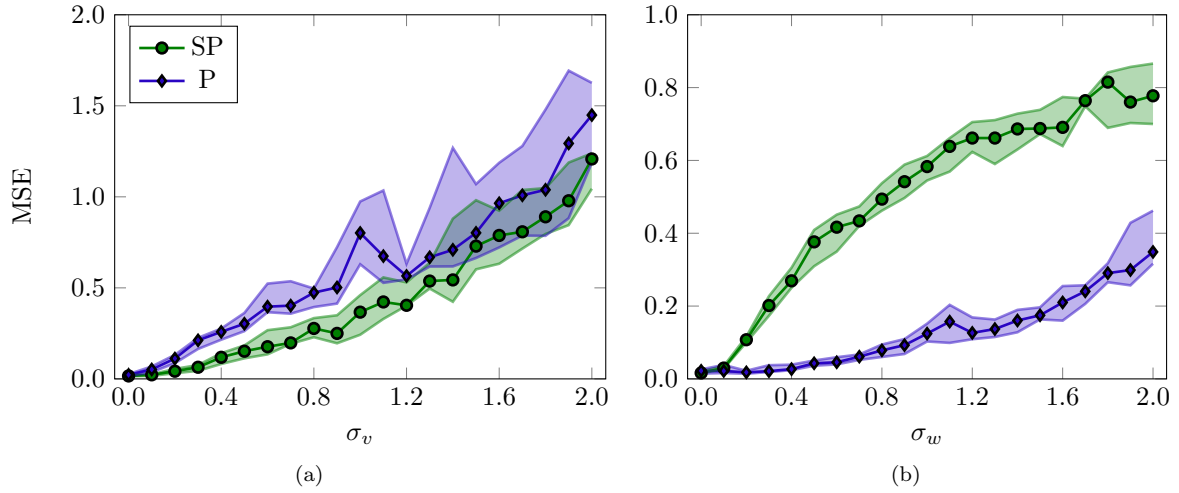


Figure 4.6 – **(White noise during training)** Free-run simulation MSE on the validation window vs noise levels for series-parallel and parallel training. The main line indicates the median and the shaded region indicates interquartile range. These statistics were computed from 12 realizations. In (a) v is a Gaussian white process and $w = 0$; and, in (b) w is a Gaussian white process and $v = 0$.

Consider $w = 0$ and v a white Gaussian noise filtered by a low pass filter with cutoff frequency ω_c . Figure 4.7 shows the free-run simulation error in the validation window for both parallel and series-parallel training for a sequence of values of ω_c and different noise intensities. The result indicates parallel training provides the best results unless the equation error has a very large bandwidth.

More extensive tests are summarized in Table 4.3, which shows the validation errors for a training set with colored Gaussian errors in different frequency bands. Again, except for white or large bandwidth equation error, parallel training seems to provide the smallest validation errors.

Equation error can be interpreted as the effect of unaccounted inputs and unmodeled dynamics, hence situations where this error is not auto-correlated are very unlikely. Therefore, the only situations we found series-parallel training to perform better (when the equation error power spectral density occupy almost the whole frequency spectrum) seem unlikely to appear in practice. This may justify parallel training to produce better models for real application problems as the pilot plant in Example 1, the battery modeling described in (Zhang et al., 2014), or the boiler unit in (Patan and Korbicz, 2012).

4.6.3 Timings

In Section 4.4 we find out the computational complexity of $\mathcal{O}(N \cdot N_\Theta^2 + N_\Theta^3)$. The first term $\mathcal{O}(N \cdot N_\Theta^2)$ seems to dominate and in Figure 4.8 we show that the running time grows linearly with the number of training samples N and quadratically with the number of parameters N_Θ .

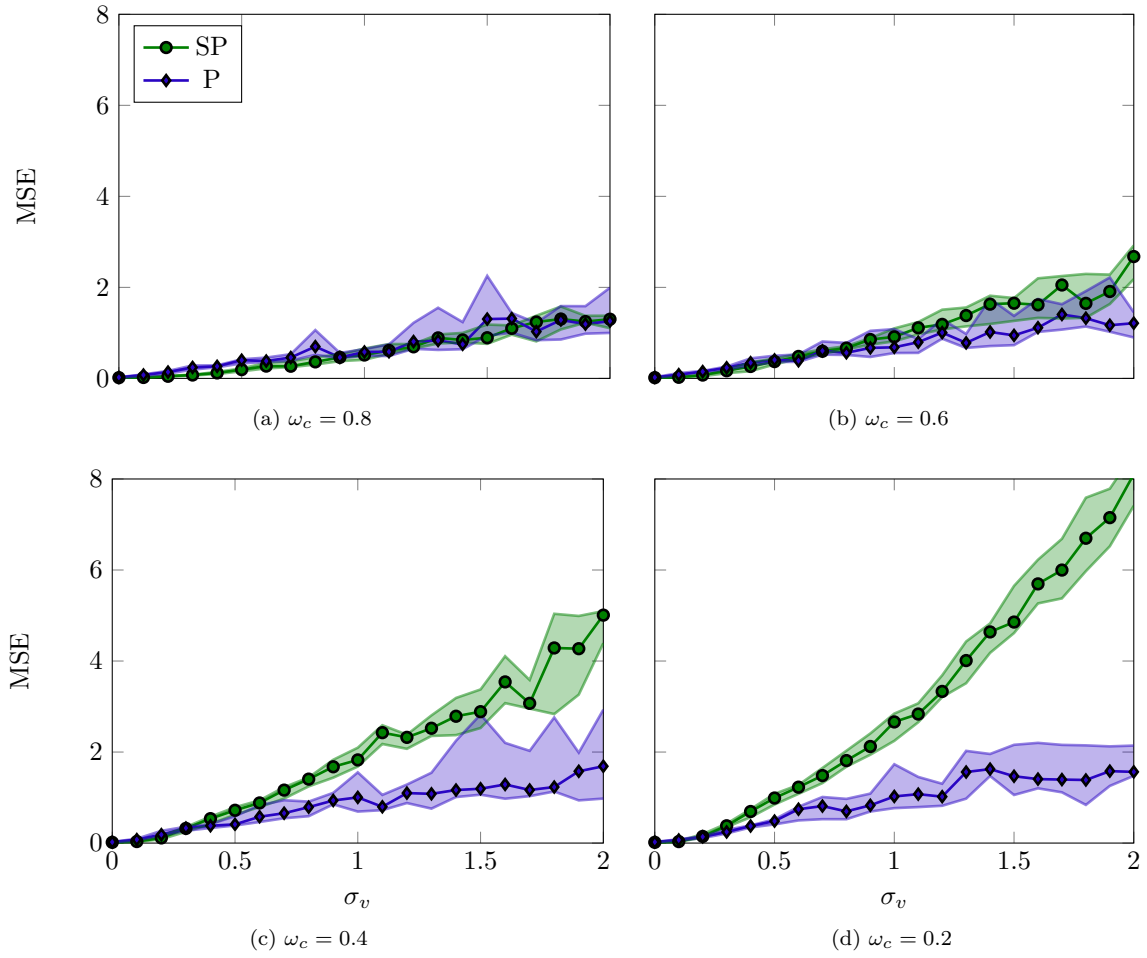


Figure 4.7 – **(Colored noise during training)** Free-run simulation MSE over the validation window vs standard deviation of colored equation error. The output error $w = 0$ and the equation error v is a colored Gaussian noise obtained by applying a 4th-order lowpass Butterworth filter with cutoff frequency ω_c to white Gaussian noise in both the forward and reverse directions. The figure shows the result for different values of ω_c , where ω_c is the normalized frequency (with $\omega_c = 1$ the Nyquist frequency). The main line indicates the median and the shaded region indicates interquartile range. These statistics were computed from 12 realizations.

The running time growing with the same rate for both training methods implies that the ratio between series-parallel and parallel training running time is bounded by a constant. For the examples we presented in this chapter the parallel training takes about 20% more than series-parallel training. Hence the difference of running times for sequential execution does not justify the use of one method over the other. Parallel training is, however, much less amenable to parallelization because of the dependencies introduced by the recursive relations used for computing its error and Jacobian matrix. We propose a possible solution to this problem in the final remarks.

4.7 Conclusion

In this chapter we have studied different aspects of parallel training under a nonlinear least squares formulation. Due to the results presented in the numerical examples we have reasons to believe parallel training can provide models with smaller generalization error than series-parallel training under more realistic noise assumptions. Furthermore, for sequential execution both the complexity analysis and the numerical examples suggest the computational cost is not significantly different for both methods.

Table 4.3 – **(Noise in frequency bands during training)** Free-run simulation MSE on the validation window for parallel and series-parallel training. Both the mean and the standard deviation are displayed (30% trimmed estimation computed from 12 realizations). In situation **(a)**, the training data was generated with zero output error ($w = 0$) and v is a Gaussian random process. In **(b)**, the training data was generated with $v = 0$ and w is a Gaussian random process. The Gaussian random process has standard deviation $\sigma = 1.0$ and power spectral density confined to the given frequency band. In both situations, the rows where the frequency ranges from 0.0 to 1.0 (the whole spectrum) corresponds to white noise, in the remaining rows we apply a 4th-order lowpass (or highpass) Butterworth filter to white Gaussian noise (in both the forward and reverse directions) in order to obtain the signal in the desired frequency band. The cell of the training method with the best validation results between the two models is colored. Its colored **red** when the difference in the MSE is larger than the sum of standard deviations and **yellow** when it is not.

ω range	(a) $v \neq 0, w = 0$		(b) $w \neq 0, v = 0$	
	SP	P	SP	P
0.0 \rightarrow 1.0	0.36 \pm 0.09	0.78 \pm 0.18	0.58 \pm 0.03	0.13 \pm 0.03
0.0 \rightarrow 0.8	0.53 \pm 0.13	0.58 \pm 0.10	0.44 \pm 0.02	0.12 \pm 0.03
0.0 \rightarrow 0.6	0.94 \pm 0.13	0.75 \pm 0.23	0.30 \pm 0.05	0.15 \pm 0.05
0.0 \rightarrow 0.4	1.86 \pm 0.20	1.07 \pm 0.38	0.46 \pm 0.05	0.15 \pm 0.02
0.0 \rightarrow 0.2	2.60 \pm 0.26	1.16 \pm 0.44	0.71 \pm 0.05	0.18 \pm 0.04
0.0 \rightarrow 1.0	0.36 \pm 0.09	0.78 \pm 0.18	0.58 \pm 0.03	0.13 \pm 0.03
0.2 \rightarrow 1.0	0.52 \pm 0.07	0.57 \pm 0.12	0.59 \pm 0.03	0.09 \pm 0.02
0.4 \rightarrow 1.0	0.57 \pm 0.08	0.22 \pm 0.05	0.63 \pm 0.05	0.05 \pm 0.01
0.6 \rightarrow 1.0	0.54 \pm 0.07	0.21 \pm 0.01	0.68 \pm 0.03	0.03 \pm 0.02
0.8 \rightarrow 1.0	0.58 \pm 0.05	0.17 \pm 0.04	0.78 \pm 0.09	0.03 \pm 0.01

Some other reasons mentioned in the literature to avoid parallel training, as the possibility of signal unboundedness (Narendra and Parthasarathy, 1990; Zhang et al., 2006; Singh et al., 2013), are also easy to circumvent (see Section 4.5).

We have based our analysis on batch training. However, instead of using all the available samples for training at once we could have fed samples one-by-one or chunk-by-chunk to the training algorithm (online training). The choice of parallel or series-parallel training, however, is orthogonal to the choice between online and batch training. And most of the ideas presented in this chapter, including: 1) the unified framework; 2) the discussion about bad local minima; and 3) the study of how colored noise affects the parameter estimation; are all applicable to the case of online training.

Several published works take for granted that series-parallel training always provide better results with lower computational cost (Narendra and Parthasarathy, 1990; Saad et al., 1994; Beale et al., 2017; Saggar et al., 2007; Petrović et al., 2013; Tijani et al., 2014; Khan et al., 2015; Diaconescu, 2008; Warwick and Craddock, 1996; Kamiński et al., 1996; Rahman et al., 2000). The results presented in this chapter show *this is not always the case* and that parallel training does provide advantages that justify its use for training neural networks in several situations (see the numerical examples).

Series-parallel training, however, has two advantages over parallel training: i) it seems less likely to be trapped in “bad” local solutions; ii) it is more amenable to parallelization. For the examples presented in this chapter the possibility of being trapped in “bad” local solutions is only a small inconvenience, requiring the data to be carefully normalized and, in some rare situations, the model to be retrained. We believe this to be the case in many situations. An exception are chaotic systems for which small variations in the parameters may cause great variations in the free-run simulation trajectory, causing the parallel training objective function to be very intricate and full of undesirable local minima.

In the following chapter, a technique called multiple shooting is introduced in the prediction error

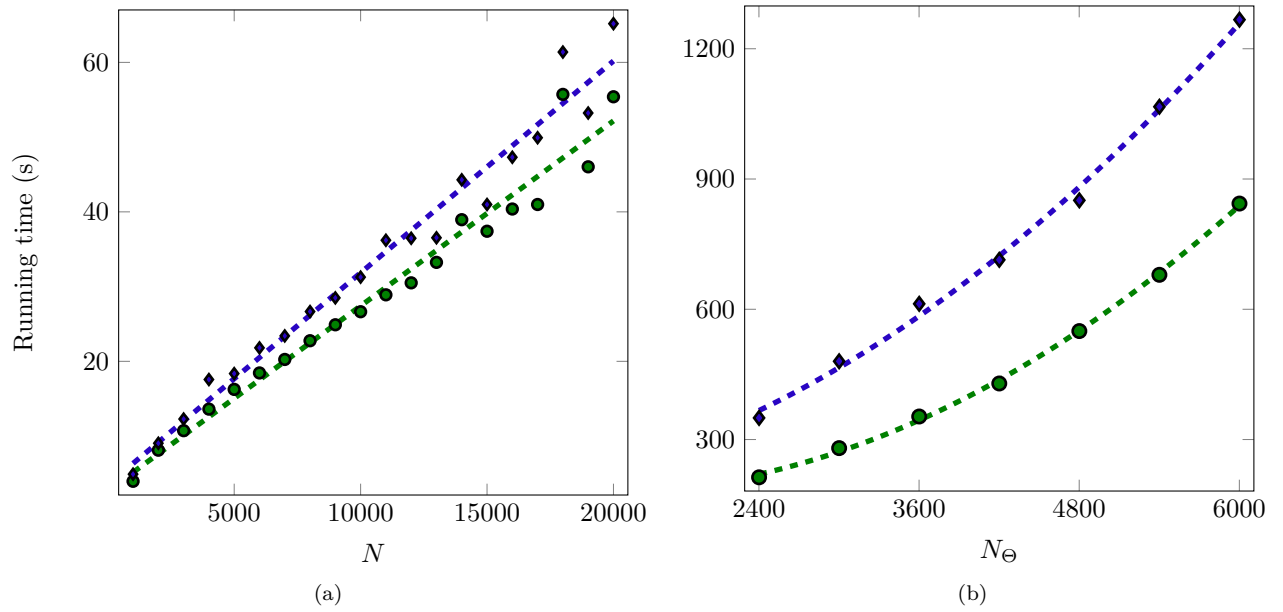


Figure 4.8 – **(Computational cost)** Running time (in seconds) of training (100 epochs). The average running time of 5 realizations is displayed for series-parallel training (●) and for parallel training (◆). The neural network has a single hidden layer and a total of N_Θ parameters to be estimated. The training set has N samples and was generated as described in Figure 7.1. In (a), we fix $N_\Theta = 61$ and plot the timings as a function of the number of training samples N , a line is adjusted to illustrate the running time grows linearly with the training size for both training methods. In (b), we fix $N = 10000$ and plot the timings as a function of the number of parameters N_Θ , a second order polynomial is adjusted to illustrate the running time quadratic growth.

methods framework as a way of reducing the possibility of recurrent structures getting trapped in “bad” local minima. Multiple shooting can also make the algorithm much more amenable to parallelization and seems to be a promising way to solve the shortcomings of parallel training we have just described.

5 On the smoothness of nonlinear system identification

The principle behind prediction error methods, detailed in Chapter 3, is to estimate the parameters of dynamic models by minimizing the error between predicted and measured values. ARX models estimate the output using a feedforward structure, for which the predicted output depends only on measured values. Output error and ARMAX models, on the other hand, have a recurrent structure, where the output prediction depends on its own past values. This distinction is illustrated in Fig. 5.1.

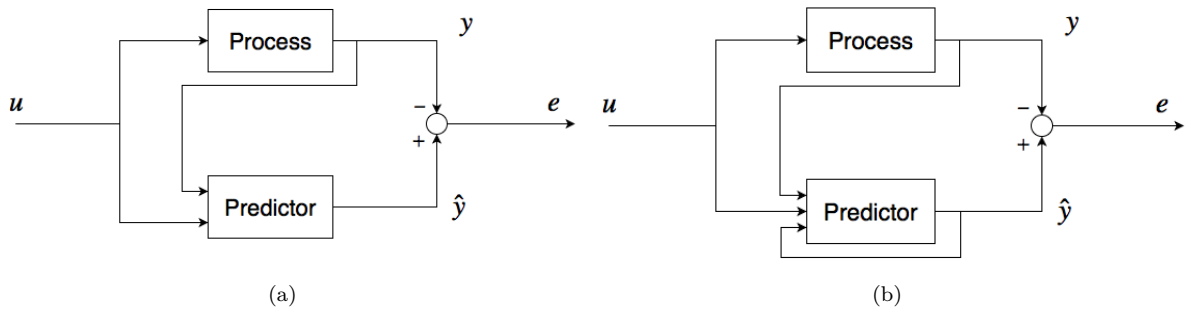


Figure 5.1 – **(Feedforward vs Recurrent)** Display feedforward and recurrent structures used by prediction error method. Prediction error methods estimate the process parameters by minimizing the error e between the measured output y and the model predicted value \hat{y} . The input-output pairs (u, y) are measured values from some dynamic process. In (a) the prediction \hat{y} depends only on measured values: ARX model; In (b) the predictor has a recurrent structure: ARMAX and OE models.

Models with a recurrent structure might yield more accurate models than their feedforward counterpart: ARMAX models are more flexible than ARX models because they include an error model that is estimated together with the *process* model, which allows them to yield consistent results even in the presence of colored equation errors (Nelles, 2013); and, OE models are often described as having smaller generalization error (Patan and Korbicz, 2012; Zhang et al., 2014) and better capability of long-term prediction (Su et al., 1992; Su and McAvoy, 1993) than ARX models.

It is common knowledge among practitioners that the optimization problem resulting from a recurrent model structure is harder to solve (Piroddi and Spinelli, 2003). For linearly parametrized models and convex loss functions, minimization of one-step-ahead prediction error leads to a convex optimization problem; for recurrent model structures, the ensuing optimization is, in general, non-convex, complicating the search for global optima. Even during local optimization, recurrent model structures can lead to cost functions with poor smoothness properties, including many ‘jagged’ local minima, cf. Figure 5.3 for an illustration. Understanding of the relationship between model structure and smoothness properties of the cost function is, however, imprecise and provides little insight into ways to circumvent the problem. Furthermore, the few studies that investigate the objective function properties in this context are focused on linear systems (Eckhard et al., 2017).

The purpose of this chapter is twofold. First, we aim to provide insight into the properties of the objective function arising in prediction error estimation problems in a general nonlinear setup. Specifically, we show how the smoothness of the objective function depends on two factors: the simulation length and the decay rate of the recurrent part of the prediction model. Second, we illustrate how this theoretical insight might be leveraged for the design and analysis of new methods.

We study the use of *multiple shooting* for prediction error methods. This technique reformulate the optimization problem that arises from minimizing the error between a prediction model and observed values. Rather than running the prediction model in the entire dataset, multiple shooting formulation splits the dataset into smaller ones and runs the prediction model in each subdivision. The equivalence with the original problem is obtained by including equality constraints in the optimization problem. This method yield a smoother objective function since it avoids long simulations, not giving the opportunity to trajectories to diverge too much.

The multiple shooting formulation has reportedly provided improvements in the parameter estimation of ordinary differential equations (Bock, 1983; Baake et al., 1992; Timmer et al., 2000; Horbelt et al., 2001; Peifer and Timmer, 2007; Sarode et al., 2015), in the solution of optimal control (Bock and Plitt, 1984; Carraro et al., 2014; Geisert and Mansard, 2016) and two-point boundary value problems (Osborne, 1969). In a system identification setting, multiple shooting has been used for estimating polynomial nonlinear space-state models (Van Mulders et al., 2010) and output error models (Ribeiro and Aguirre, 2017) in settings where conventional methods fail to provide good solutions. And, here, we extend the class of system identification problems for which multiple shooting can be applied to the entire class of prediction error methods. In addition, theoretical arguments are put forward to help understand why and when the proposed method is useful.

An early version of the results from this chapter was presented at the 2017 IFAC World Congress and appear in the proceedings:

Shooting Methods for Parameter Estimation of Output Error Models, *Antonio H. Ribeiro, L.A. Aguirre*. Proceedings of the IFAC World Congress, 2017. IFAC-PapersOnLine v. 50 (1), pp. 13998-14003. doi: 10.1016/j.ifacol.2017.08.2421

A more mature version, much closer to what we will be presented in this Chapter, is now provisionally accepted for publication in the journal *Automatica*:

On the Smoothness of Nonlinear System Identification, *Antônio H. Ribeiro, Koen Tiels, Jack Umenberger, Thomas B. Schön, Luis A. Aguirre*. Provisionally accepted for publication in *Automatica*, 2020.

This more mature version was also presented as a poster at the *European Research Network on System Identification, ERNSI* (2019).

Section 5.1 gives some notes about the notation and setup used in this chapter. Section 5.2 provides results about the objective function's Lipschitzness and β -smoothness for problems with a recurrent prediction model. Section 5.3 presents a multiple shooting formulation for prediction error methods and analyzes its objective function properties. Section 5.4 describes a multiple shooting implementation and Section 5.5 gives numerical examples. Final comments are provided in Section 5.6.

5.1 Notation and setup

We use the exactly same notation and setup from Chapter 3. Hence, we consider a training dataset $\mathcal{Z}^N = \{(\mathbf{u}[k], \mathbf{y}[k]), k = 1, 2, \dots, N\}$ containing N measured inputs and outputs of a dynamical system. We define the cost function based on the distance between the predicted and measured values:

$$V = \frac{1}{N} \sum_{k=1}^N \|\mathbf{y}[k] - \hat{\mathbf{y}}[k]\|^2. \quad (5.1)$$

And we use the same general nonlinear space state representation:

$$\mathbf{x}[k] = \mathbf{h}(\mathbf{x}[k-1], \mathbf{z}[k]; \boldsymbol{\theta}); \quad (5.2a)$$

$$\hat{\mathbf{y}}[k] = \mathbf{g}(\mathbf{x}[k], \mathbf{z}[k]; \boldsymbol{\theta}), \quad (5.2b)$$

where $\mathbf{x}[k]$ denotes the internal state vector of this model at instant k . As mentioned in Section 3.2.5 this representation is general enough to encompass linear and nonlinear ARX, output error and ARMAX models.

5.2 Smoothness of prediction error methods

The theorem below relates the Lipschitz constant of V , and its gradient (i.e. β -smoothness), to the simulation length N . The Lipschitz constant of the cost function and the β -smoothness both play a crucial role in optimization (Nesterov, 1998). Lower values imply that local (Taylor) expansions of the cost function are more predictive of the cost function, and that optimization algorithms can still converge while taking larger steps. It also gives an upper bound on how distinct in performance two close local minima may be.

The first part of the theorem below can indeed be seen as a formalization of the exploding gradient problem, often studied in the context of neural networks (Pascanu et al., 2013). The second part gives information about the explosion of second order derivatives and curvature and, to the best of our knowledge, is novel. As a result of our analysis we have that not only *walls* (resulting from large first order derivatives) might be formed in the non-contractive regions of the parameter space, but also regions with exploding curvature with multiple close local minima (cf. Figure 5.3(a))

Theorem 5.1. *Let $\mathbf{h}(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta})$ and $\mathbf{g}(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta})$ in (5.2) be Lipschitz in $(\mathbf{x}, \boldsymbol{\theta})$ with constants L_h and L_g on a compact and convex set $\Omega = (\Omega_{\mathbf{x}}, \Omega_{\mathbf{z}}, \Omega_{\boldsymbol{\theta}})$. With $\{\mathbf{z}[k]\}_{k=1}^N \subseteq \Omega_{\mathbf{z}}$ and $(\Omega_{\mathbf{x}}, \Omega_{\boldsymbol{\theta}}) \subseteq \mathbb{R}^{N_x} \times \mathbb{R}^{N_{\boldsymbol{\theta}}}$. If there exist at least one choice of $(\mathbf{x}_0, \boldsymbol{\theta})$ for which there is an invariant set contained in Ω , then, for trajectories and parameters confined within Ω :*

1. *The cost function V defined in (5.1) is Lipschitz with constant:*¹

$$L_V = \begin{cases} \mathcal{O}(L_h^{2N}) & \text{if } L_h > 1, \\ \mathcal{O}(N) & \text{if } L_h = 1, \\ \mathcal{O}(1) & \text{if } L_h < 1. \end{cases} \quad (5.3)$$

2. *If the Jacobian matrices of \mathbf{h} and \mathbf{g} are also Lipschitz with respect to $(\mathbf{x}, \boldsymbol{\theta})$ on Ω , then the gradient of the cost function ∇V is also Lipschitz with constant:*

$$L'_V = \begin{cases} \mathcal{O}(L_h^{3N}) & \text{if } L_h > 1, \\ \mathcal{O}(N^3) & \text{if } L_h = 1, \\ \mathcal{O}(1) & \text{if } L_h < 1. \end{cases} \quad (5.4)$$

Proof. See Appendix B. □

For contractive models², under certain regular conditions, we have $L_h < 1$ and, accordingly to the above theorem, both the Lipschitz constant and the β -smoothness of the cost function can be bounded by

¹ Where \mathcal{O} denotes the big O notation. It should be read as: $L(N) = \mathcal{O}(g(N))$ if and only if there exists positive integers M and N_0 such that $|L(N)| \leq Mg(N)$ for all $N \geq N_0$.

² We say a dynamical system $\mathbf{x}[k+1] = \mathbf{h}(\mathbf{x}[k])$ is *contractive* if, for all \mathbf{x} and \mathbf{w} , it satisfies $\|\mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{w})\| < L\|\mathbf{x} - \mathbf{w}\|$, for $L < 1$.

a constant that, asymptotically, does not depend on the simulation length. All contractive systems have a unique fixed point inside the contractive region, and all trajectories converge to such a fixed point (Rudin, 1964, Theorem 9.23). Systems with richer nonlinear dynamic behaviours, such as limit cycles and chaotic attractors, and also unstable systems, are *non-contractive* and will always have $L_h \geq 1$. The Lipschitz constants and β -smoothness for these systems may, according to Theorem 5.1, blow up exponentially (or polynomially for some limit cases) with the maximum simulation length.

In a less formal way, for models that have infinitely long dependencies (i.e. are non-contractive) the distance between trajectories that are close in the parameter space might become progressively larger along the simulation length because errors will accumulate. This might yield very intricate objective functions in some parts of the parameter space making the optimization problem either very dependent on the initial point or very hard to optimize using nonlinear programming methods.

For example, consider the estimation of the linear system $y[k] = \theta y[k-1]$. For $\theta > 1$ the system is unstable and the distance between trajectories grows exponentially with N . For two close points θ_1 and $\theta_2 = \theta_1 + \epsilon$ and the same initial condition, the trajectory $y_1[k] = \theta_1 y_1[k-1]$ will diverge from the trajectory $y_2[k] = \theta_2 y_2[k-1]$ by a factor $\theta_1^N - (\theta_1 + \epsilon)^N = \mathcal{O}(\epsilon \theta_1^{N-1})$.

5.3 Multiple shooting

The theoretical results from the previous section suggest that long simulation lengths might yield regions of the parameter space where the cost function is intricate and hard for the optimization algorithm to navigate in. In this section, we propose the application, in the context of prediction error methods, of a technique called *multiple shooting* for which the maximum simulation length is a design parameter. This enables solving problems that would be impossible or very hard to solve in the setup of Section 5.1. This conventional setup for which a single set of initial conditions is used will be named *single shooting* from now on.

5.3.1 Method formulation

For the *multiple shooting* formulation, rather than simulating the prediction model (3.13) through the entire dataset from a single initial condition vector \mathbf{x}_0 , the data is split into M intervals $\{[m_i + 1, m_{i+1}] \mid i = 1, \dots, M\}$, $0 = m_1 < m_2 < \dots < m_M < m_{M+1} = N$, each one with its own set of initial conditions $\mathbf{x}_0^i \in \mathbb{R}^{N_x}$. The i -th vector of initial conditions \mathbf{x}_0^i is used in the computation of the prediction $\hat{\mathbf{y}}^i[k]$ in the interval $m_i + 1 \leq k \leq m_{i+1}$:

$$\begin{aligned} \mathbf{x}^i[k] &= \mathbf{h}(\mathbf{x}^i[k-1], \mathbf{z}[k]; \boldsymbol{\theta}), \text{ for } \mathbf{x}^i[m_i] = \mathbf{x}_0^i, \\ \hat{\mathbf{y}}^i[k] &= \mathbf{g}(\mathbf{x}^i[k], \mathbf{z}[k]; \boldsymbol{\theta}). \end{aligned} \quad (5.5)$$

Since the length of the simulation is limited to the smaller interval $[m_i + 1, m_{i+1}]$, the trajectory is less likely to strongly diverge and this typically helps the optimization procedure by making the objective function *smoother*.

Let, $\Delta m_i = m_{i+1} - m_i$, we define:

$$V_i = \frac{1}{\Delta m_i} \sum_{k=m_i+1}^{m_{i+1}} \|\mathbf{y}[k] - \hat{\mathbf{y}}^i[k]\|^2, \quad i = 1, \dots, M, \quad (5.6)$$

to be the cost function associated with the i -th interval. Where the prediction $\hat{\mathbf{y}}^i[k]$ depends upon $\boldsymbol{\theta}$ and

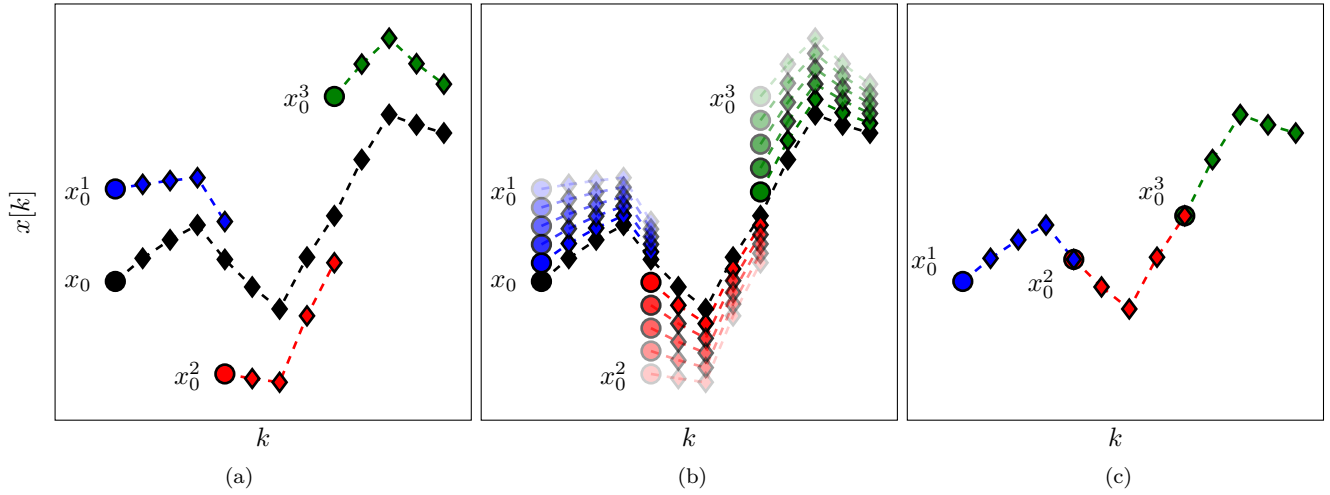


Figure 5.2 – **(Multiple shooting)** Comparison between single shooting and multiple shooting in a unidimensional example ($N^x = 1$). In **black** we present the simulation of the dynamic system through the entire window length using the single initial condition x_0 (represented by \bullet). The simulated values are represented by \blacklozenge . Dividing the window length into three sub-intervals and simulating the system in each of these, for initial conditions x_0^1 , \bullet , x_0^2 , \bullet , and x_0^3 , \bullet , results in the three different simulations represented by \blacklozenge , \blacklozenge and \blacklozenge , respectively. In (a), the end of one simulation does not coincide with the beginning of the next one ($x^{i-1}[m_i] \neq x_0^i$). In (b), we show what happens as $\|x^{i-1}[m_i] - x_0^i\| \rightarrow 0$. And, in (c), we show that, when $x^{i-1}[m_i] = x_0^i$, the concatenation of these short simulations is equivalent to a single one carried out over the entire window length.

\mathbf{x}_0^i , according to (3.13). The *multiple shooting* formulation has as objective function:

$$V^M = \sum_{i=1}^M \frac{\Delta m_i}{N} V_i. \quad (5.7)$$

This objective function includes states $\mathbf{x}_0^1, \dots, \mathbf{x}_0^M$ as free variables in the optimization. Hence, rather than reinforcing the cohesion of the states $\mathbf{x}[k]$ by defining them through a recurrence relation that casts a dependency of $\mathbf{x}[k]$ all the way back to the initial condition \mathbf{x}_0 , as in the *single shooting* formulation, the cohesion between subsequent states is achieved through optimization constraints, resulting in the following problem:

$$\begin{aligned} \min_{\theta, \mathbf{x}_0^1, \dots, \mathbf{x}_0^M} V^M, \\ \text{subject to: } \mathbf{x}^{i-1}[m_i] = \mathbf{x}_0^i, \\ \text{for } i = 2, 3, \dots, M. \end{aligned} \quad (5.8)$$

The next theorem gives the equivalence between (3.14) and (5.7). The theorem and its corollary are a formalization of the intuition provided in Fig. 5.2 and they give further insight into how the constraints in the *multiple shooting* formulation are used to imitate a single simulation throughout the entire dataset.

Theorem 5.2. *If $\mathbf{x}^{i-1}[m_i] = \mathbf{x}_0^i$, for $i = 2, 3, \dots, M$ and $\mathbf{x}_0^1 = \mathbf{x}_0$, then $V = V^M$.*

Proof. Let us call $\mathbf{x}[k]$, $\hat{\mathbf{y}}[k]$ and $\mathbf{x}^i[k]$, $\hat{\mathbf{y}}^i[k]$ the states and predictions, respectively, in the single shooting simulation and in the i -th multiple shooting interval. For a fixed i , if $\mathbf{x}[m_i] = \mathbf{x}_0^i$ then $\mathbf{x}[k] = \mathbf{x}^i[k]$ for all $k \in [m_i + 1, m_{i+1}]$. Hence, inside this same interval $\hat{\mathbf{y}}[k] = \hat{\mathbf{y}}^i[k]$. Applying this for every i it follows from the respective definitions that: $V = \sum_{i=1}^M \frac{\Delta m_i}{N} V_i = V^M$. \square

Corollary 5.1. *The pair $(\theta^*, \mathbf{x}_0^*)$ is a global solution of (3.14) if and only if there exist $(\mathbf{x}_0^2, \dots, \mathbf{x}_0^M)$ such that $(\theta^*, \mathbf{x}_0^*, \mathbf{x}_0^2, \dots, \mathbf{x}_0^M)$ is a global solution of the optimization problem (5.8).*

The cost function that arises in the multiple shooting formulation has some nice properties that will be investigated in the next section. Multiple shooting has some other advantages that will not be the focus of this work: namely, it is more amenable to parallelization, since each cost function V_i and the respective derivatives can be computed independently and, possibly, in parallel. Also, longer simulations usually yield larger numerical error due to finite precision errors that accumulate along the simulation, so that having shorter simulations helps attenuating the problem.

Finally, multiple shooting can be understood as a generalization of the single shooting case. That is because, if $M = 1$ and $\Delta m_1 = N$, multiple and single shooting result in the same optimization problem.

5.3.2 Properties of the objective function

The next theorem follows from basic inequality manipulation and relates the Lipschitzness and β -smoothness of the cost function V^M with that of its components V_i .

Theorem 5.3. *Define V^M as in (5.7), if each component V_i is Lipschitz continuous with constant L_{V_i} then V^M is also Lipschitz with constant equal to or smaller than $L_{V^M} = \max(L_{V_1}, \dots, L_{V_M})$. Additionally, if the gradient of each component ∇V_i is Lipschitz continuous with constant L'_{V_i} then ∇V^M is also Lipschitz with constant equal to or smaller than $L'_{V^M} = \max(L'_{V_1}, \dots, L'_{V_M})$.*

Proof. For $\theta_{\text{ext}} = (\theta, \mathbf{x}_0^1, \dots, \mathbf{x}_0^M)$ and $\phi_{\text{ext}} = (\phi, \mathbf{w}_0^1, \dots, \mathbf{w}_0^M)$ we have that:

$$|V^M(\theta_{\text{ext}}) - V^M(\phi_{\text{ext}})| \leq \sum_{i=1}^M \frac{\Delta m_i}{N} |V_i(\theta, \mathbf{x}_0^i) - V_i(\phi, \mathbf{w}_0^i)| \leq$$

$$\sum_{i=1}^M \frac{\Delta m_i}{N} L_{V_i} \|\theta, \mathbf{x}_0^i\|^T - \|\phi, \mathbf{w}_0^i\|^T\| \leq L_{V^M} \|\theta_{\text{ext}} - \phi_{\text{ext}}\|,$$

where $L_{V^M} = \max(L_{V_1}, \dots, L_{V_M})$. And similarly, $L'_{V^M} = \max(L'_{V_1}, \dots, L'_{V_M})$, which yields the second result. \square

Putting together Theorems 5.1 and 5.3 we have that L_{V^M} and L'_{V^M} depend asymptotically on $\Delta m_{\max} = \max_{1 \leq i \leq M} \Delta m_i$ and not on N . For instance, if $L_h > 1$:

$$L_{V^M} = \mathcal{O}(L_h^{2\Delta m_{\max}}); \quad L'_{V^M} = \mathcal{O}(L_h^{3\Delta m_{\max}}). \quad (5.9)$$

Since Δm_{\max} is a design parameter, we can actually have some control over the Lipschitzness and β -smoothness of the objective function for models where $L_h \geq 1$. Hence multiple shooting might help considerably when estimating parameters of non-contractive models ($L_h \geq 1$).

5.3.3 Comparison with other methods

Multiple-shooting is presented here as a possible way of limiting the maximum simulation length Δm_{\max} . A method that appears in the system identification literature that also has a similar effect is the minimization of the multi-step-ahead prediction error (Farina and Piroddi, 2008; Rossiter and Kouvaritakis, 2001; Farina and Piroddi, 2011; Zhao et al., 2014; Terzi et al., 2018). For those methods, the simulation is truncated by a fixed number K of steps backwards. That is, given k , we define an auxiliary variable $\tilde{\mathbf{x}}_k[i]$ and use the state equations: $\tilde{\mathbf{x}}_k[i] = \mathbf{h}(\tilde{\mathbf{x}}_k[k-1], \mathbf{z}[k]; \theta)$ to simulate the evolution of this

auxiliary state variable for $i = k - K, \dots, k$, starting from a fixed initial condition $\tilde{\mathbf{x}}_k[k - K] = \tilde{\mathbf{x}}_{0,k}$. The prediction is then computed using $\hat{\mathbf{y}}[k] = \mathbf{g}(\tilde{\mathbf{x}}_k[k], \mathbf{z}[k]; \boldsymbol{\theta})$. The parameters are obtained by minimizing a cost function similar to (5.1).

The use of multi-step-ahead prediction is equivalent to the original single shooting formulation only if $K = N$. Hence, it imposes a trade-off between: i) the benefits of using a recursive model (such as better properties for non-white process noise) for larger values of K ; and, ii) the benefits of having a smaller simulation length (such as smoother cost function for non-contractive models, by Theorem 5.1) for smaller values of K . Multiple shooting, on the other hand, has an exact equivalence with the original single shooting problem regardless of the simulation length (see Theorem 5.2 and Corollary 5.1), so both desirable characteristics are obtained at once.

The computational cost for computing V and its derivatives is K times greater for multi-step-ahead than for the multiple shooting method, because of the need to propagate the auxiliary variables $\tilde{\mathbf{x}}_k$. On the other hand, multi-step-ahead prediction yields an unconstrained problem, rather than a constrained one, for which more efficient solvers might be available. Also, it does not include initial state variables in the optimization problem, what results in a lower dimensional problem. While the larger number of parameters in the multiple-shooting does not increase the computational cost so much, because of the underlying sparse structure of the problem, it does require a careful implementation with smart use of those properties.

The limitations of such methods in the fully nonlinear setting presented in this work might be avoided in some special cases. Currently, to the best of our knowledge, multi-step-ahead prediction has been studied primarily in a linear model setting (Farina and Piroddi, 2008; Rossiter and Kouvaritakis, 2001; Farina and Piroddi, 2011; Zhao et al., 2014; Terzi et al., 2018), for which these methods might result in convex optimization problems. They are most popular for system identification in model predictive control problems, for which the multi-step-ahead prediction usually fits well into the moving horizon framework and allows for more efficient implementations (Rossiter and Kouvaritakis, 2001; Zhao et al., 2014; Terzi et al., 2018).

5.4 Implementation

The equality constraint problem (5.7), which arises from the multiple shooting formulation, is solved using an implementation of the sequential quadratic programming solver originally described in (Lalee et al., 1998) and available in the SciPy library (Virtanen et al., 2020). The method is described in Appendix A.³ At each iteration of this algorithm, the following values are required i) The cost function V^M ; ii) its gradient ∇V^M ; iii) the constraints; iv) the Jacobian matrix of the constraints (which can be represented using a sparse representation); and, v) for any given vector \mathbf{p} , the product of the Lagrangian⁴ Hessian and the vector \mathbf{p} (the full Lagrangian Hessian matrix does not need to be computed). The following sequence provides a way of computing all derivatives required by the optimizer.

³ SciPy is a open source scientific library written in Python. The optimization method used might be called as `scipy.optimize.minimize(..., method='trust-constr')` and is available since Scipy 1.1. It was implemented as part of my Google Summer of Code project.

⁴ The Lagrangian is given by: $\mathcal{L}(\phi, \boldsymbol{\lambda}) = V(\phi) + \boldsymbol{\lambda}^T \mathbf{c}(\phi)$.

ALGORITHM 5.1 (DERIVATIVES). For a given parameter θ and set of initial conditions:

1. For $i = 1, \dots, M$, do:
 - a) For $k = m_i + 1, \dots, m_{i+1}$:
 - i. Compute $\mathbf{x}^i[k]$ and $\hat{\mathbf{y}}^i[k]$ with (3.13);
 - ii. Compute A_k, B_k, C_k and F_k (as defined in Section 3.2.11.1);
 - iii. Compute $D^i[k]$ and $J^i[k]$ with the sensitivity equations;
 - b) Compute V_i using Eq. (5.6);
 - c) Compute ∇V_i using a formula equivalent to (B.2);
 - d) Approximate the product of the Hessian with a given vector, $\nabla^2 V_i \mathbf{p}$, using the first terms from a expression equivalent to (3.24).
2. Compute V^M with (5.7);
3. Compute $\nabla V^M = \sum_{i=1}^M \frac{\Delta m_i}{N} \nabla V_i$;
4. Compute the value of the constraint from the values of $\mathbf{x}^i[m_{i+1}]$, $i = 1, \dots, M$;
5. Compute the Jacobian matrix of the constraints from $J^i[m_{i+1}]$, $i = 1, \dots, M$;
6. Compute $\nabla^2 V^M \mathbf{p} = \sum_{i=1}^M \frac{\Delta m_i}{N} \nabla^2 V_i \mathbf{p}$;
7. Compute the product of the Hessian $\lambda^T \mathbf{c}(\phi)$ with a vector \mathbf{p} using 2-point finite differences;
8. Compute the product of the Lagrangian Hessian and a vector $\nabla^2 \mathcal{L}(\phi, \lambda) \mathbf{p}$, summing the Hessians computed in steps 6 and 7.

□

Some approximations were used for computing the second derivatives: 1) for computing the Hessian of the objective function, the standard least-squares approximation for the Hessian is used; and, 2) for computing the Hessian of the constraint we use finite-difference approximation. The use of finite differences here comes inexpensively because we only need to evaluate the Hessian times a vector, and not the full matrix. Hence, it can be done at the cost of an extra Jacobian matrix evaluation.

Notice that step (1) from the above algorithm can be parallelized, with different processes (or threads) performing the computation for different values of i .

5.5 Numerical examples

In this section, we provide numerical examples to illustrate the ideas from the previous sections. The focus here is to study both the cost function of the identification problem and the behaviour of the optimization solver for simple examples through the lens of the theoretical results in Section 5.2 and, also, to illustrate how *multiple-shooting* might help mitigate some of the discussed problems. The code for reproducing the examples is available in the GitHub repository: github.com/antonior92/MultipleShootingPEM.jl.

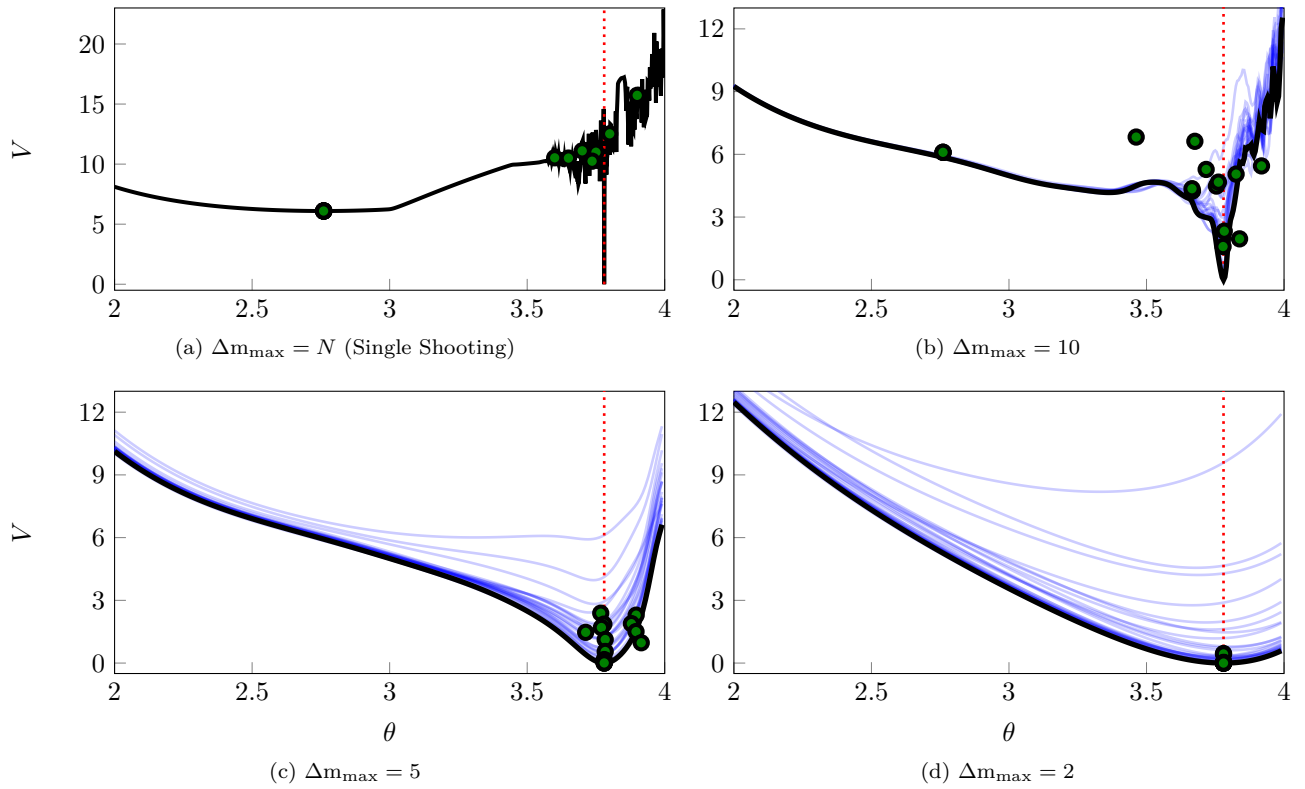


Figure 5.3 – **(Logistic map parameter estimation)** *Cost function* of the optimization problem for \mathbf{x}_0^i fixed in its true values (in black) and for disturbed versions of these initial conditions (in blue). We present the result for four values of Δm_{\max} , and omit disturbed initial condition objective functions for the single shooting case ($\Delta m_{\max} = N$) to make it easier to visualize. The green circles, \bullet , indicate the pair (θ, V) corresponding to a solution found by the solver. There are 15 circles in each figure (some of them overlapping), the circles correspond to solutions for different initial guesses. As initial guesses we picked values of θ uniformly spaced between 3.2 and 3.9, with \mathbf{x}_0^i picked from randomly disturbed versions of the true initial conditions (which are known because we generated the data ourselves). The true value $\theta = 3.78$ is indicated by the dotted red vertical line.

5.5.1 Example 1: output error model for chaotic system

This example illustrates how multiple shooting makes prediction error methods more robust against the choice of initial conditions for the optimization. A dataset with $N = 200$ samples is generated using the logistic map (May, 1976):

$$y[k] = \theta y[k-1](1 - y[k-1]), \quad (5.10)$$

with $\theta = 3.78$. From the generated dataset we try to estimate an *output error* model with the same structure.

Figure 5.3 (a) illustrates the objective function for the *single shooting* case. The model that is being fitted to the data presents a chaotic behavior for $\theta \in [3.57, 4]$, which justifies the very intricate objective function in this region. For chaotic systems, small variations in the parameters may cause large variations in the system trajectory and, hence, abrupt changes in the *free-run simulation error*. This is the reason why the estimation of an output error model has many local solutions in this problem.

The solutions found by the solver, for different initial guesses, are also displayed in Figure 5.3 (a). Notice that the solver fails to find the true solution because it always gets trapped at a local stationary point near its initial guess. Even in the noise-free situation we are considering, the identification procedure

is made very challenging by the chaotic nature of the system, that, for a long enough simulation, yields large trajectory differences even for small parameter variations.

Multiple shooting makes the problem easier by limiting the total simulation length. Figures 5.3 (b), (c) and (d) display the objective function and the solutions found by the solver starting from different initial guesses. Each figure displays the result for a different choice of Δm_{\max} : For (b) the maximum simulation length is $\Delta m_{\max} = 10$; for (c), $\Delta m_{\max} = 5$; and, for (d), $\Delta m_{\max} = 2$.

The identification procedure becomes easier as Δm_{\max} is made smaller. For Figures (b) and (c) the solver converges to the true parameter for some initial guesses but, also, to undesirable local solutions for other initializations. For Figure (d) the solver converges to the true solution regardless of the initial guess.

For the multiple shooting case, besides θ , the initial conditions are also optimization parameters. To help with the visualization of this multidimensional problem, Figures 5.3 (b), (c) and (d) display the main curve corresponding to the objective function for the true initial conditions and faded lines corresponding to the objective function for perturbed initial conditions. Another consequence of the problem having more parameters than displayed in the figure is that the cost function found by the solver does not need to lie on any of the objective function curves displayed in the figure, since it may have a different set of initial conditions \mathbf{x}_0^i .

It is important to highlight that the mechanism used here is not to take the system outside of the chaotic regime, but rather avoid simulating the system for too long. By doing that, we avoid the major problem that arises in the identification of chaotic systems, i.e. the high sensitivity to parameters and initial conditions. This results in a better behaved objective functions (cf. Figure 5.3). The constraints allow the equivalence with the original prediction error problem (according to Theorem 5.2 and Corollary 5.1).

Table 5.1 gives the number of function evaluations and the running time for the four situations displayed in Figure 5.3. The convergence happens within just a few iterations for $\Delta m_{\max} = N$ (single shooting) because any initial point is probably very close to some optimal *local* solution. As we reduce Δm_{\max} the objective function becomes less intricate and this is reflected in the convergence of the solver. For $\Delta m_{\max} = 10$ the solver takes much longer to converge. We believe this happens because the local solution is not so close in the parameter space to the initial guess anymore. As we further decrease Δm_{\max} , however, the convergence becomes faster, because it is dealing with, what we believe to be, a smoother problem that can be more accurately approximated by low order approximations.

Table 5.1 – **(Optimization performance in multiple shooting)** Number of function evaluations and total running time to convergence for different values of Δm_{\max} . We give, the minimum, maximum and median among 15 runs for the situations presented in Figure 5.3. (*)The number of iterations is limited to 1000 and the solver is interrupted when this number is reached.

Δm_{\max}	function evaluations			run time (s)		
	min	median	max	min	median	max
N	1	15	23	0.01	0.2	1.1
10	43	1000*	1000*	0.7	24.7	27.3
5	29	115	645	1.0	2.9	26.9
2	21	50	65	1.7	2.9	3.8

5.5.2 Example 2: neural network for modeling pilot plant

This example uses data from the level process station described in Example 1 from (Ribeiro and Aguirre, 2018). As in the original paper, we use a neural network to model the water column height as a function of the voltage applied to a control valve that modulates the water flow. We compare three different training methods: i) minimizing the one-step-ahead prediction error (NN ARX) ii) minimizing the free-run simulation error using single shooting method (NN OE - SS); and, iii) minimizing the free-run simulation error using multiple shooting method (NN OE - MS).

The neural network (NN) training depends on the weight initialization, hence the performance of the neural network can be regarded as a random variable and is displayed in Figure 5.4, which compares the empirical cumulative distribution of the *mean square error* (MSE) over the validation dataset for the three methods. A linear ARX model ($n_y = 1$ and $n_u = 1$) was trained and tested under the same conditions to serve as the baseline. Methods (i) and (ii) and the linear ARX baseline were described in (Ribeiro and Aguirre, 2018). Method (iii) is introduced here.

The cumulative distribution function gives, for each x -axis value, the probability of the method to yield a validation MSE smaller or equal to this value. It was estimated from 100 realizations of the neural network training procedure. Figure 5.4 shows that for more than 90% of the realizations, estimating the parameters by minimizing the free-run simulation (NN OE) offers significant advantages over the minimization of the one-step-ahead error (NN ARX). When using a standard single shooting formulation, however, it also makes the parameter estimation procedure more sensitive to the initial conditions, with the algorithm yielding some really bad results for some initial choices (Ribeiro and Aguirre, 2018). This results in a long-tailed distribution for the MSE (Fig. 5.4). More precisely, in 10 out of 100 realizations the NN OE - SS model yields a performance that is inferior to the linear ARX baseline, some of the realizations worse than the linear baseline by a factor of 100. The performance of the NN OE - SS and NN OE - MS is very similar for 90% of the realizations, the tail of the distribution, however, is very different, with the multiple shooting procedure rarely producing very bad results. In order to highlight the differences, results where NN OE - SS and NN OE - MS are worse than the baseline are presented, respectively, as blue and red circles in Fig. 5.4.

This example illustrates how the use of multiple shooting alleviates the problem of high sensitivity to initial conditions, making it possible to estimate output error models with extra robustness against variations of the initial conditions and lower probability of getting trapped at local minima with very bad performance.

This example also shows the limitations of the multiple shooting formulation. The training time for NN OE - MS model is 282 seconds, for NN OE - SS model is 3.9 seconds, and for NN ARX model is 3.3 seconds. This means that the single shooting parameter estimation could be repeated, roughly, 70 times for each multiple shooting run. Hence solving the single shooting problem several times and choosing the best result would also avoid very bad solutions and could, still, be computationally less expensive than solving the multiple shooting problem. The longer training time is due to two factors: i) per iteration the multiple shooting approach takes, roughly, 3.5 times more than the single shooting approach; and, ii) it takes, approximately, 20 times more iterations to converge. Both are consequences of the fact that a higher dimensional *constrained* optimization problem is being solved.

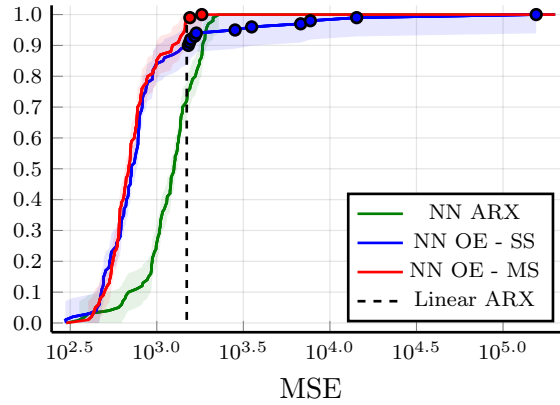


Figure 5.4 – **(Neural network multiple shooting estimation)** Empirical cumulative distribution of the free-run simulation MSE over the validation dataset. The results obtained in (Ribeiro and Aguirre, 2018) for an ARX neural network (NN ARX) and an (single shooting) output error neural network (NN OE - SS) are displayed together with the result obtained estimating the parameters using multiple shooting (NN OE - MS). A Linear ARX model is considered as a baseline and is displayed by the dashed line. The multiple shooting estimation uses $\Delta m_{\max} = 3$ and the training is restricted to 2000 iterations of the optimization algorithm or until either the gradient or the step size drops below 10^{-12} . The other models were estimated exactly as in (Ribeiro and Aguirre, 2018). All the neural network models have 10 nodes in the hidden layer, $n_y = n_u = 1$ and were trained with the same training dataset. Each curve is the result of 100 realizations and, for each realization, the neural network initial weights $w_{i,j}^{(n)}$ are drawn from a normal distribution with zero-mean and standard deviation $\sigma = (N_{s(n-1)})^{-0.5}$ and the bias terms $\gamma_i^{(n)}$ are initialized with zeros (LeCun et al., 1998). Realizations of NN OE - SS and NN OE - MS that perform worse than the baseline are indicated respectively as blue, \bullet , and red circles, \bullet . Confidence intervals (95%) are displayed as shaded regions around the estimated cumulative distribution, these have been computed using the Dvoretzky-Kiefer-Wolfowitz inequality (Dvoretzky et al., 1956).

5.5.3 Example 3: pendulum and inverted pendulum

Consider the following discrete-time nonlinear system:

$$\begin{cases} x_1[k+1] = x_1[k] + \delta x_2[k]; \\ x_2[k+1] = -\delta \frac{g}{l} \sin x_1[k] + (1 - \delta \frac{k_a}{m})x_2[k] + \delta \frac{1}{m}u[k]; \\ y[k] = x_1[k], \end{cases} \quad (5.11)$$

which corresponds to a pendulum model, discretized using the Euler approximation $\dot{x}(t) \approx \frac{x((k+1)\delta) - x(k\delta)}{\delta}$. Where g is the gravity acceleration, m is the mass connected to the extremity of the pendulum, l is the length of the (massless) rod connecting the mass to the pivot point, and k_a is the linear friction constant. It has two states: the angle of the mass (x_1) and the angular velocity (x_2). The input $u[k]$ is the force applied to the mass.

This system has multiple equilibrium points, namely, $(x_1, x_2) = (\pm\pi i, 0)$ for $i = 0, 1, 2, 3, \dots$. The equilibrium points at $(x_1, x_2) = (\pm 2\pi i, 0)$ are stable and the equilibrium points at $(x_1, x_2) = (\pi \pm 2\pi i, 0)$ are unstable. For this system, with $g = 9.8$, $L = 0.3$, $m = 3$, $k_a = 2$ and $\delta = 0.01$, we define three different datasets: (a) A dataset for which small inputs are applied to the system, that stays under the influence of the stable point $(x_1, x_2) = (0, 0)$ and $y[k]$ stays, approximately, inside the range $[-\frac{\pi}{2}, +\frac{\pi}{2}]$; (b) A dataset for which the system is maintained close to the unstable point $(x_1, x_2) = (\pi, 0)$ by a linear controller; and, (c) A dataset for which the input is large enough to drive the pendulum to full rotations around its center. The output corresponding to those three situations are displayed in Figure 5.5.

Fixing $m = 3$ and $\delta = 0.01$ parameters $\frac{g}{l}$ and k_a of an output error model with the structure

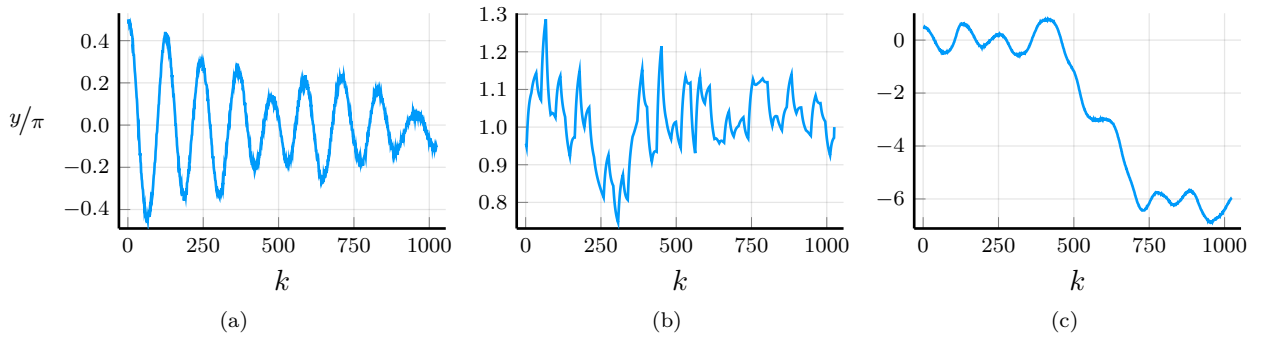


Figure 5.5 – **(Pendulum and inverted pendulum position data)** Display the *output signal* $y[k]$ for the three different datasets used for estimating the parameters. The dataset size is $N = 1024$ samples. (a) The input applied in this case is a zero-mean Gaussian random input with standard deviation $\sigma_u = 10$, each random value being hold for 20 samples. The input in this case is unable to drive the system away from the influence of the stable fixed point $(0, 0)$. (b) The input $u[k]$ in this case is obtained by the control law: $u[k] = 40\delta e[k-1] - 78.8\delta e[k-2] + 38.808\delta e[k-3] + 1.02u[k-1] - 0.02u[k-2]$, where the error is the difference between the reference and the output: $e[k] = r[k] - y[k]$. The reference is $r[k] = \pi + \Delta r[k]$ where $\Delta r[k]$ is a zero-mean Gaussian random input with standard deviation $\sigma_r = 0.2$, each random value being held for 20 samples; and, (c) Same thing as (a) but with the larger standard deviation $\sigma_u = 50$, which is able to drive the pendulum to complete full rotations. For (a) and (c) zero-mean Gaussian white noise with standard deviation $\sigma_r = 0.03$ was added to the output.

presented in (5.11) were estimated from the data. A visualization of the cost function is presented in Figure 5.6 together with numerical solutions found by means of the single shooting and multiple shooting formulation starting from different initial conditions.

For dataset (a), the single shooting formulation is able to recover the true parameters from data for most of the initial conditions. Some exceptions occur when initialized far away from the correct initial conditions. For datasets (b) and (c), for which the system needs, respectively, to operate close to the unstable dynamics or to account for the existence of multiple fixed points, the cost function is highly intricate, very non-convex and full of local minima. In this case, the optimization algorithm, even when initialized close to the local solution, fails to converge to reasonable solutions. This result is consistent with Theorem 5.1 and how the smoothness of the objective function degenerates (exponentially) on sets of the parameter space for which the prediction model is non-contractive, such as the trajectories close to the unstable fixed point of the system (5.11). The use of multiple shooting yields an objective function that looks similar to a paraboloid in the region of interest for the three cases, which suggests that local approximations might be valid over a large region. The solutions converge to the true parameter regardless of the initialization point in this formulation.

In Figure 5.6(a), it is shown that even though the model we are trying to estimate is contractive, for some initial guesses the optimization algorithm gets stuck in undesirable regions of the parameter space. Figure 5.6(d) shows how multiple shooting helps the optimization algorithm escaping from these regions. Nevertheless, we can also notice from this example that the optimization problem is harder when we are trying to estimate a parameter in a region of the parameter space where the model is non-contractive. Indeed, in this case the optimization algorithm will necessarily have to navigate in a region where the cost function might be highly intricate.

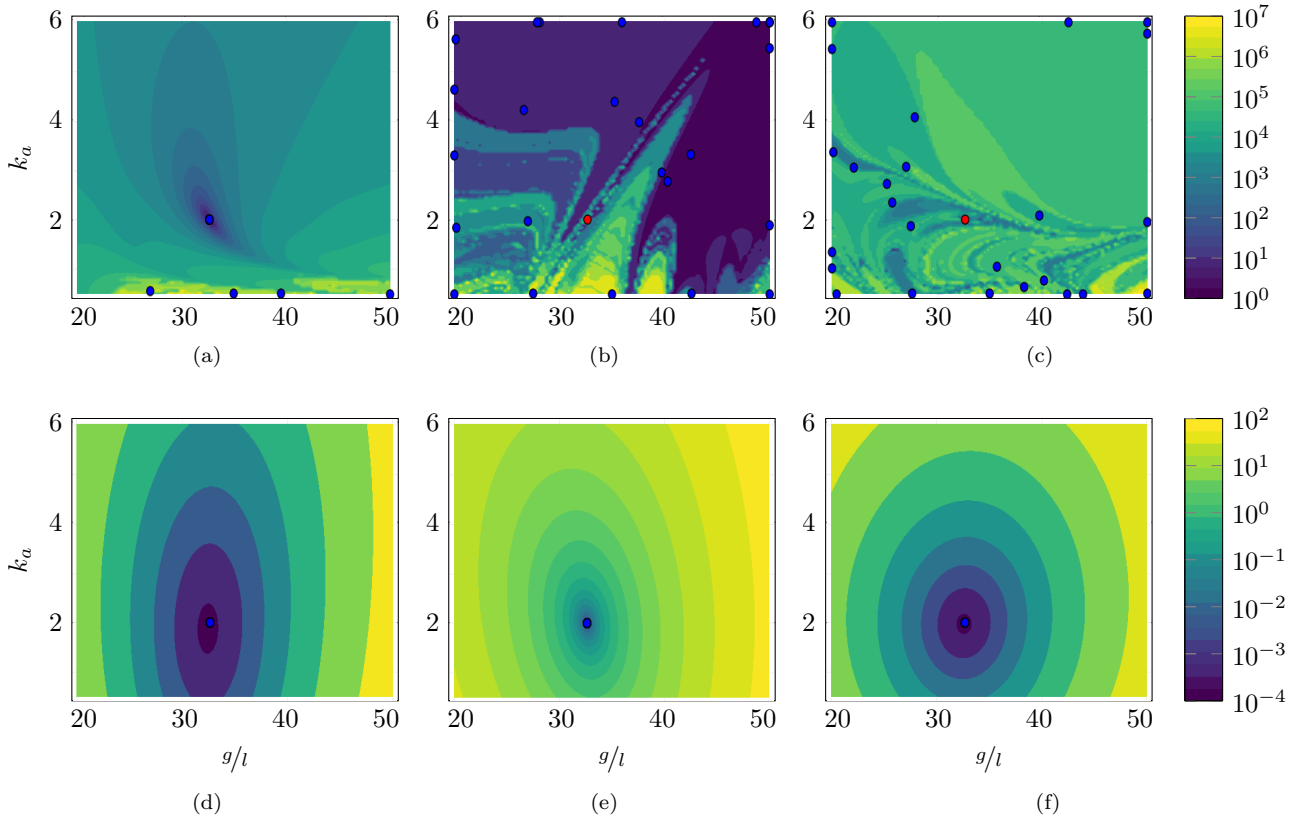


Figure 5.6 – **(Pendulum and inverted pendulum parameter estimation)** Contour plot of the cost function. Figures (a), (b) and (c) correspond to the cost function V from single shooting simulation for the datasets (a), (b) and (c) generated as described in Fig. 5.5 caption; and, in (d), (e) and (f) the cost function for the same problems is displayed for the multiple shooting formulation with $\Delta m_{\max} = 16$. The true parameter is indicated by a red circle, \bullet , solutions found by the solver are indicated by blue circles, \bullet . There are 25 blue circles in each figure (some of them overlapping), each circle corresponds to the solution for a different initial guess. Some solutions are outside of the displayed region and the corresponding blue dots are displayed at the edge of the plot. Initial guesses were picked values of θ uniformly spaced on a grid of points uniformly spaced in the rectangle $[20, 50] \times [0.5, 6]$. It is important to highlight that the plots show a two dimensional projection of a cost function that is defined on an extended parameter space that includes the initial conditions \mathbf{x}_0^i , $i = 1, \dots, M$ as parameters, which were fixed to the true values when generating the contour plots.

5.5.4 Practical considerations

The main benefits, studied in this work, of using smaller Δm_{\max} is to have a smoother cost function and make the algorithm less sensitive to the initialization of the optimization algorithm. In Example 3, this is accomplished for $\Delta m_{\max} = 16$ (cf. Fig. 5.6). For this example, further reductions on the maximum simulation length don't improve these two aforementioned aspects. For other examples $\Delta m_{\max} = 16$ might not be the most appropriate choice. For instance, in Example 1, as the simulation length suffer consecutive reductions, i.e. $\Delta m_{\max} = \{N, 10, 5, 2\}$, it yields a smoother cost functions, it helps the optimization algorithm to avoid local minimums (see Fig. 5.3) and it also improve the algorithm convergence (cf. Table 5.1). Reducing the maximum simulation length, however, also has the undesirable effect of increasing the dimensionality of the problem (both in number of variables and in number of constraints). Hence, Δm_{\max} is a design parameter that encompass the trade-off between smoothness and dimension of the optimization problem. Reducing the simulation length might also increase the number of independent operations that might be run in parallel, although this effect is not the focus of this work.

The choice of initial state conditions \mathbf{x}_0^i follows very naturally for nonlinear ARX, ARMAX or output error models for which the states are, maybe with some noise contamination, directly measured. Hence, the choice of initial guess for the states in Example 1 and 2 follow the guidelines described in Section 3.2.6. Example 3 is one example that has unmeasured state variables. However, the unmeasured state variable x_2 has the interpretation of the derivative of x_1 , so we have used a finite difference approximation of the derivative of x_1 , which was available to us. While the high-pass behaviour of the derivative amplifies the noise, this choice was still better than a completely arbitrary one.

5.6 Conclusion

The relevance of this work lies in the very general setting for which the proposed methods and results hold. The major technical contribution is to show that for dynamic prediction models that are non-contractive (i.e. do not converge asymptotically to a single stable point) in the region of interest, the upper bound for the Lipschitz constant and the β -smoothness blows up exponentially with the simulation length, and this can make the optimization problem very hard to solve. This was illustrated with numerical examples with systems that are not contractive due to the presence of chaotic regions and non-stable equilibrium points. Because of these regimes, the objective function becomes very intricate in some regions of the parameter space and the optimization algorithm fails to find a good solution. Even for problems that are contractive in the region of interest multiple shooting might help preventing the solver from getting stuck in undesirable regions of the parameter space (cf. Section 5.5.3).

Multiple shooting makes the simulation length a design parameter and hence allows one to actually solve optimization problems that would be unfeasible in a single shooting setting. The price paid compared to single shooting methods is that a nonlinear constrained optimization problem must be solved instead of an unconstrained one. It also makes it harder to generalize to situations other than batch training, such as online training.

One advantage of multiple shooting, that is not thoroughly explored in this work, is the possibility of parallelization that comes from creating independent simulation executions. The recent advances of high quality automatic differentiation libraries with easily available interfaces to graphical processing units (GPUs), i.e. PyTorch (Paszke et al., 2017) and Tensorflow (Abadi et al., 2015), might be an interesting path for future implementations that can fully explore this possibility using all the parallelization capabilities offered by GPUs.

We believe understanding the estimation of parameters for problems with recurrent structures in a fully nonlinear and non-convex setting is both very challenging and highly relevant for system identification and machine learning fields and that this work is a step in the direction of a better understanding of this type of problems.

6 Attractor and smoothness in the analysis of recurrent neural networks training

The exploding and vanishing gradient problem has been the major conceptual principle to drive the development and to motivate changes in the recurrent neural network (RNN) architectures during the last decade. In this chapter, we argue that this principle, while powerful, might need some refinement to explain recent developments. We refine the concept of exploding gradients by reformulating the problem in terms of the cost function's smoothness, which gives insight into higher-order derivatives and the existence of regions with many close local minima. On the other hand, we clarify the distinction between vanishing gradients and the need for the RNN to learn attractors to fully use its expressive power. Through the lens of these refinements, we shed new light onto recent developments in the RNN field, namely stable RNN and unitary (or orthogonal) RNNs.

The content of this chapter is featured in the following publication:

Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness, *Antônio H. Ribeiro, Koen Tiels, Luis A. Aguirre and Thomas B. Schön.*
To appear in the Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS), 2020.

The outline of this chapter is the following: Section 6.1 motivates and introduces the importance of this work. Section 6.2 gives the setup and notation used along the chapter. Section 6.3 study the mechanism used to store information in a recurrent neural network. Section 6.4 revisit some of the results from Chapter 5 in the context of RNNs. Section 6.5 study and visualize attractors during training, for the LSTMs, stable and unitary RNNs. Section 6.6 draw some connections with previous work. Section 6.7 give final comments.

6.1 Introduction

Training recurrent neural networks can be challenging. The problem in training these recurrent models is usually stated in terms of the so-called *exploding and vanishing gradient problem* (Hochreiter and Schmidhuber, 1997; Pascanu et al., 2013; Bengio et al., 1994). This problem is easy to understand and has motivated many techniques, including the use of gating mechanisms (Hochreiter and Schmidhuber, 1997; Cho et al., 2014a), gradient clipping (Pascanu et al., 2013), non-saturating activation functions (Chandar et al., 2019) and the manipulation of the propagation path of gradients (Kanuparthi et al., 2019).

A recently proposed family of methods based on the same principle are the so-called unitary and orthogonal RNNs (Mhammedi et al., 2017; Vorontsov et al., 2017; Lezcano-Casado and Martínez-Rubio, 2019; Helfrich et al., 2018; Arjovsky et al., 2016; Jing et al., 2017; Maduranga et al., 2019; Wisdom et al., 2016; Lezcano-Casado, 2019). In these architectures, the eigenvalues of the hidden-to-hidden weight matrix are fixed to one to simultaneously avoid exploding gradients (that might appear when the eigenvalues are larger than one) and vanishing gradients (that might appear when they are less than one). Similar, but less strict constraints were proposed by Zhang et al. (2018); Kerg et al. (2019). A different line of study goes in the opposite direction and suggests that using RNNs constrained to be stable can provide as good performance as unconstrained RNNs on many tasks. Miller and Hardt (2018) discuss examples for which projecting all eigenvalues to values less than one does not affect the performance. In this

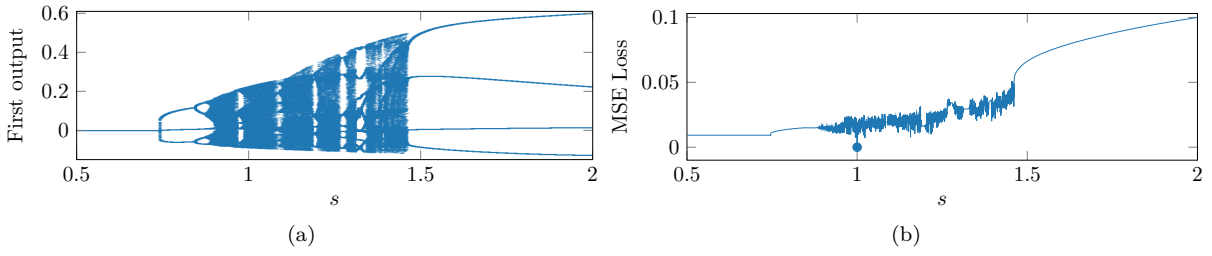


Figure 6.1 – **(Chaotic LSTM)** Display: a) Bifurcation diagram; and b) cost function (mean-square error) for LSTM models with parameter vectors $\theta(s) = s\theta_{\text{true}}$.

case, RNNs can be truncated with arbitrarily small error and hence could be replaced by feedforward structures. Indeed, feedforward structures, such as transformer-based architectures (Vaswani et al., 2017) and convolutional networks (Bai et al., 2018b), have recently matched or outperformed RNNs in many tasks. These feedforward architectures yield impressive results in language and music modeling (Bai et al., 2018b; van den Oord et al., 2016; Dauphin et al., 2017; Radford et al., 2018, 2019), text-to-speech conversion (van den Oord et al., 2016), machine translation (Kalchbrenner et al., 2016; Gehring et al., 2017) and other sequential learning tasks for which RNNs have been the default choice until recently (Bai et al., 2018b).

This work proposes to build and improve on the present understanding of recurrent neural networks. We believe the moment is propitious for such analysis, and it might help to explain this shift of winning paradigm for sequence modeling and the lack of consensus on how to deal with the eigenvalues. On the one hand, we complement the vanishing gradient interpretation with an analysis of the attractors of the RNN for storing long-term information. While the condition for dynamic attractors (other than a single fixed point) to appear is the same for the gradients to vanish, these two notions are distinct. The effect of teachers forcing and the differences between training and inference modes are also highlighted. And the numerical examples explore the training mechanism and the attractors of the stable (Miller and Hardt, 2018) and orthogonal RNNs (Lezcano-Casado and Martínez-Rubio, 2019). On the other hand, we study the *smoothness* of the cost function, which builds on and improves the notion of exploding gradients, since it also takes into account higher-order derivatives. As a result of our analysis we have that not only “walls” might be formed in the cost function (as suggested by Pascanu et al. (2013)), but also regions with very intricate behavior, full of local minima and very hard for the optimization algorithm to navigate on (cf. Figure 6.1).

6.2 Setup and notation

For a given training sequence $\mathcal{Z}^N = \{(\mathbf{u}_t, \mathbf{y}_t), t = 1, 2, \dots, N\}$, be $\hat{\mathbf{y}}_t$ the model predicted output at instant t . A cost function quantifying the performance of this model can be defined as:

$$V = \frac{1}{N} \sum_{t=1}^N l(\mathbf{y}_t, \hat{\mathbf{y}}_t). \quad (6.1)$$

If the prediction model is parametrized and $\hat{\mathbf{y}}_t$ depends on the parameter vector θ , this parameter vector θ can be estimated (i.e. the neural network is trained) by minimizing V or, in the case of multiple independent training sequences, minimizing a weighted average of many V s defined as in (6.1). Here, l is the loss function. Common choices for regression and classification problems are the squared error and cross entropy loss, respectively.

Here we assume the predicted output $\hat{\mathbf{y}}_t$ is the output of an RNNs. RNNs are nonlinear discrete-time

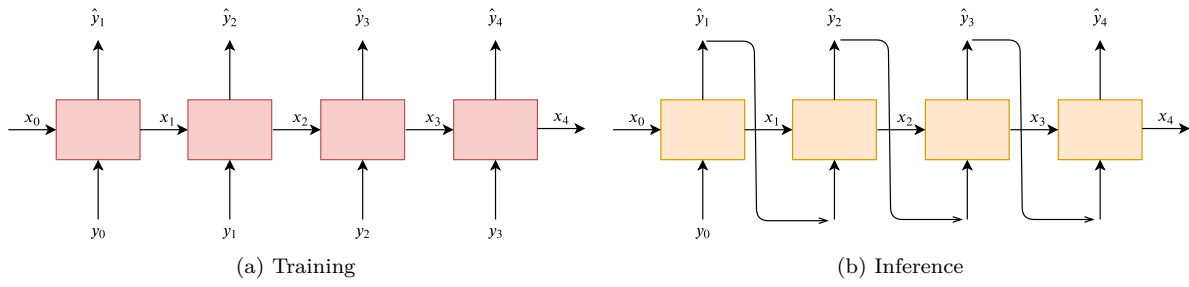


Figure 6.2 – **(Teacher forcing)** The figure display the same recurrent neural network in two different modes. In (a), observed previous values of the output \mathbf{y}_t are used as input to the model. In (b), the neural network own outputs are feed back as inputs, instead.

dynamical systems that can be expressed as:

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{z}_t; \boldsymbol{\theta}); \quad (6.2a)$$

$$\hat{\mathbf{y}}_t = \mathbf{g}(\mathbf{x}_t, \mathbf{z}_t; \boldsymbol{\theta}), \quad (6.2b)$$

with hidden state $\mathbf{x}_t \in \mathbb{R}^{N_x}$, predicted output $\hat{\mathbf{y}}_t \in \mathbb{R}^{N_y}$, and parameters $\boldsymbol{\theta}$. This formula is sufficiently general to capture vanilla RNNs, LSTM (Hochreiter and Schmidhuber, 1997), GRU (Cho et al., 2014a), and stacked layers of these units. Here the input \mathbf{z}_t to the neural network varies depending on the problem and usage. And, it might be different during training and inference. During training, in some problems such as language modeling (Peters et al., 2018), what is used as input to the model are previously observed outputs, i.e. $\mathbf{z}_t = \mathbf{y}_{t-1}$, see Figure 6.2(a). For problems where there is a well-defined input to the problem, this input will be fed to the neural network and $\mathbf{z}_t = \mathbf{u}_t$, see the examples in Section 6.5.1 and 6.5.2. Since we strive for generality here, we consider that $\mathbf{z}_t = (\mathbf{u}_t, \mathbf{y}_{t-1})$. And this technique of using previously observed outputs during training is sometimes called *teacher forcing*. The mode that is used during training, however, does not need to be used during inference. And, after training, instead of the observed values \mathbf{y}_{t-1} , we can use the last predicted value from the neural network. And, redefine $\mathbf{z}_t = (\mathbf{u}_{t-1}, \hat{\mathbf{y}}_{t-1})$. For instance, in language models, this makes it possible to use the model that was trained to predict only the next word to generate an entire phrase, see Figure 6.2(b).

Notice that the representation (6.2) is general enough to account for this inference mode that feeds the output back into the input channel. Let $\mathbf{x}_{t+1}^i = (\mathbf{x}_t, \hat{\mathbf{y}}_t)$ be the new definition of hidden state, and redefine the transition and output functions as bellow:

$$\begin{pmatrix} \mathbf{x}_{t+1} \\ \hat{\mathbf{y}}_t \end{pmatrix} = \begin{pmatrix} \mathbf{f}(\mathbf{x}_{t+1}, (\mathbf{u}_t, \hat{\mathbf{y}}_{t-1}); \boldsymbol{\theta}) \\ \mathbf{g}(\mathbf{x}_t, (\mathbf{u}_t, \hat{\mathbf{y}}_{t-1}); \boldsymbol{\theta}) \end{pmatrix}; \quad (6.3a)$$

$$\hat{\mathbf{y}}_t = (\mathbf{0}, \mathbf{I}) \begin{pmatrix} \mathbf{x}_{t+1} \\ \hat{\mathbf{y}}_t \end{pmatrix}, \quad (6.3b)$$

where \mathbf{I} is the identity matrix. Hence, a different way of understanding teacher forcing is that it introduces the possibility of a new transition and output functions \mathbf{f}^i and \mathbf{g}^i , defined as above, to be used during inference.

6.3 Information in a recurrent network

It is common to associate the difficulty of training recurrent neural networks to store information for long periods with the *vanishing gradients* problem (Hochreiter and Schmidhuber, 1997; Pascanu et al., 2013). That is, as the error gradients are backpropagated through the RNN, they might shrink exponentially to zero, making it harder to learn long dependencies.

This interpretation has motivated successful strategies for training recurrent neural networks, such as the LSTM unit (Hochreiter and Schmidhuber, 1997). While useful, this notion alone does not give the whole picture, and one also needs to consider the presence or absence of dynamic attractors of an RNN to fully evaluate its capability of storing information.

6.3.1 Entropy of the internal states

We start our analysis by studying the amount of *information* stored by an RNN and how this amount of information changes over time. The initial discussion applies both to $(\mathbf{f}, \mathbf{g}, \mathbf{x})$ used during training or to $(\mathbf{f}^i, \mathbf{g}^i, \mathbf{x}^i)$ which is used during inference. We specify the consequences to the different cases latter on.

Assume that at time t the system state \mathbf{x}_t is distributed according to $p_t(\mathbf{x}_t)$. The entropy associated with this probability distribution provides a way of quantifying how much information we would obtain in measuring the state. For the set Ω_x , the entropy H_t at time t can be computed as:

$$H_t = - \int_{\Omega_x} p_t(\mathbf{x}_t) \log p_t(\mathbf{x}_t) d\mathbf{x}_t. \quad (6.4)$$

The next theorem gives a lower bound on the entropy, based on the dimension of \mathbf{x}_t and on the Lipschitz constant of \mathbf{f} .

Theorem 6.1. *Let $\mathbf{f}(\cdot, \mathbf{u}_t; \boldsymbol{\theta})$ in Eq. (3.13) be a one-to-one continuous differentiable map, and let $\mathbf{f}(\mathbf{x}, \mathbf{u}; \boldsymbol{\theta})$ be Lipschitz in $(\mathbf{x}, \boldsymbol{\theta})$ with constant L_f on a compact and convex set $\Omega = (\Omega_x, \Omega_u, \Omega_\theta)$. Then the entropy H_t in Eq. (6.4) with $x_t \in \mathbb{R}^{N_x}$ satisfies:*

$$H_t + N_x \log L_f \leq H_{t+1}. \quad (6.5)$$

Proof. Under the assumption that $\mathbf{f}(\cdot, \mathbf{u}_t; \boldsymbol{\theta})$ is a 1-1 continuous differentiable map (cf. Theorems 3-13 and 3-14 by Spivak (1998)), applying the change of variable $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta})$ we get:

$$\begin{aligned} H_t &= - \int_{\mathbf{f}(\Omega_x, \mathbf{u}_t; \boldsymbol{\theta})} p_{t+1}(\mathbf{x}_{t+1}) \log \left(p_{t+1}(\mathbf{x}_{t+1}) \left| \det \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \right| \right) d\mathbf{x}_{t+1} \\ &= H_{t+1} - \int_{\mathbf{f}(\Omega_x, \mathbf{u}_t; \boldsymbol{\theta})} p_{t+1}(\mathbf{x}_{t+1}) \log \left(\left| \det \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \right| \right) d\mathbf{x}_{t+1}, \end{aligned}$$

where $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t}$ is the Jacobian matrix of $\mathbf{f}(\cdot, \mathbf{u}_t; \boldsymbol{\theta})$. Using Hadamard's inequality:

$$\log \left| \det \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \right| \leq \sum_{i=1}^{N_x} \log \|\mathbf{v}_i\|_2, \quad (6.6)$$

where \mathbf{v}_i is the i -th column of the Jacobian matrix and $\log \|\mathbf{v}_i\|_2 \leq \log \left\| \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \right\|_2 \leq \log L_f$. Equation (6.5)

□

The bounds are illustrated in Figure 6.3, and:

- for a system with $L_f < 1$, the entropy might decay over time towards zero. The larger L_f is, the slower the decay of information retention can be.
- for a system with $L_f = 1$, the entropy can stay constant if the bound in (6.5) is tight. This means that such a system might retain the information;

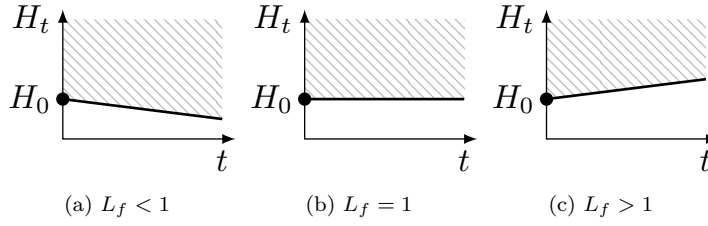


Figure 6.3 – **(Information in RNNs)** Illustrate the lower bound on the entropy H_t obtained in (6.5). Starting from H_0 , the entropy H_t can only take values in the shaded region. The entropy: a) may decay over time (contractive system); b) may remain constant (e.g. for a periodic oscillator); and, c) increases when $L_f > 1$ (e.g. for a chaotic system).

- for a system with $L_f > 1$, the amount of entropy increases with time. This is the case for chaotic systems, for instance.

This notion is complementary to the concept of vanishing gradients. On the one hand, $L_f < 1$ is a sufficient condition for gradients to vanish, making it harder for the model to escape from the corresponding regions of the parameter space during training. On the other hand, these are also the regions where the model may not be able to retain information indefinitely.

6.3.2 Contractive vs non-contractive systems

There is a deep connection between systems for which $L_f < 1$ and the definition of contractive systems:

Definition 6.1 (contractive system). A dynamical system (6.8) is said to be contractive in $\Omega_{\mathbf{x}}$ if it satisfies

$$\|\mathbf{f}(\mathbf{x}, \bar{\mathbf{u}}; \boldsymbol{\theta}) - \mathbf{f}(\mathbf{w}, \bar{\mathbf{u}}; \boldsymbol{\theta})\| < L_{\bar{\mathbf{u}}} \|\mathbf{x} - \mathbf{w}\|, \quad (6.7)$$

for $L_{\bar{\mathbf{u}}} < 1$ and for all \mathbf{x} and \mathbf{w} in $\Omega_{\mathbf{x}}$. □

All contractive systems have a unique fixed point inside the contractive region $\Omega_{\mathbf{x}}$, and all trajectories converge to such a fixed point (Rudin, 1964, Theorem 9.23). In this case the distribution $p_t(\mathbf{x}_t)$ will tend towards a discrete distribution with only one point mass at the fixed point, and thus zero entropy. If the function \mathbf{f} is Lipschitz and the system is non-contractive we have $L_f \geq 1$ ¹. Systems with richer nonlinear dynamic behaviors, such as multiple fixed points, limit cycles and chaotic attractors, and also unstable systems, are all *non-contractive*.

6.3.3 Long-term memory

In what follows, the concept of long-term memory will be closely related to the definition of an attractor of a dynamical system.

Definition 6.2 (attractor of a dynamical system). Let \mathbf{u}_t be equal to a constant $\bar{\mathbf{u}}$ for every t . An **attractor** of a dynamical system:

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \bar{\mathbf{u}}; \boldsymbol{\theta}), \quad (6.8)$$

is a subset $A \subset \mathbb{R}^{N_x}$ for which:

1. if $\mathbf{x}_{t_0} \in A$ then $\mathbf{x}_t \in A$ for all $t \geq t_0$;

¹ If $\bar{\mathbf{u}} \in \Omega_{\bar{\mathbf{u}}}$ it follows that $L_f \geq L_{\bar{\mathbf{u}}}$, hence if the system is not contractive $L_f \geq L_{\bar{\mathbf{u}}} \geq 1$.

2. there exists a neighborhood of A , called the **basin of attraction** $B(A)$, such that for any open neighborhood of A , say N , there is a positive integer $T \geq t_0$ such that if $\mathbf{x}_{t_0} \in B(A)$ then $\mathbf{x}_t \in N$ for all real $t > T$;
3. there is no proper (non-empty) subset of A having the first two properties.

□

Property 1 is related to the concept of long-term memory. If at a given time instant t_0 , $\mathbf{x}_{t_0} \in A$, the system state will still be in A at any point in the future, that is, it will “remember” this set A . Systems that do not respect this property will just leave (i.e. “forget”) set A for some $t > t_0$.

Property 2 is related to the robustness of long-term memory. If a given state \mathbf{x}_t belongs to $B(A)$ then $\mathbf{x}_t \rightarrow A$ as $t \rightarrow \infty$. If we apply a finite non-zero input sequence \mathbf{u} sufficiently small such that the state remains in $B(A)$, then for \mathbf{f} continuous in \mathbf{u} the system will converge to A . In other words, the system will not “forget” A even in the presence of some (sufficiently small) disturbance.

The idea of defining long-term memory as attractors of a dynamical system is not new. Similar approaches have been pursued in (Bengio et al., 1993, 1994). Examples of attractors of after-training RNNs are studied by Sussillo and Barak (2013) and by Maheswaranathan et al. (2019), for toy problems and sentiment analysis, respectively.

In this work, we will build on this idea and show the RNN may go through bifurcations during training for tasks that require long-term memory and learn attractors to solve the problem. We will also show how stable RNNs and orthogonal RNNs use different mechanisms. For doing this analysis, we will use the **bifurcation diagram** for visualizing the attractors of the RNN. These diagrams show the values visited in steady-state behavior (when a constant input is applied to the system) as a function of some bifurcation parameter s . See Figure 6.1(a).

As mentioned before, it is possible to have different state transition and output functions for training and inference, respectively. See Figure 6.2 and Equation (6.3). Hence, the attractors of a recurrent model might be different during training and inference. This is the case for the numerical example in Section 6.5.3, where we include the analysis of the attractor and bifurcation diagram in both scenarios. Before this analysis, however, we will study the influence of the internal dynamics on the smoothness of the cost function.

6.4 Smoothness of the cost function

In this section, we extend the concept of exploding gradient with the analysis of the smoothness of the cost function. We do that by establishing connections between the *smoothness* of the cost function and the internal dynamics of the RNN *during training*. Our analysis is based on the Lipschitz constant of the cost function and of its gradient (sometimes called β -smoothness). Both constants play a crucial role in optimization, see (Nesterov, 1998), and can be understood as qualitative measurements of how *smooth* the cost function is. Lower values imply that the cost function is less intricate and that optimization algorithms can still converge while taking larger steps. It also provides an upper bound on how distinct in performance two close local minima may be.

This smoothness analysis will suggest that higher-order derivatives (which contain information about the curvature) might also explode in some regions of the parameter space. This indicates that not only walls are formed, but also entire regions of the parameter space with intricate behavior and full of undesirable local minima. This goes beyond what many papers suggest about exploding gradients. For

instance, [Pascanu et al. \(2013\)](#) formulate the hypothesis that when gradients explode, they do so in some specific direction, creating *walls* of high curvature. [Doya \(1993\)](#) speculates that bifurcations might cause the jumps in the cost function.

6.4.1 Example: chaotic LSTM

We consider an LSTM model of dimension 2 without the bias terms. In Figure 6.1(a) we show the *bifurcation diagram* and, in Figure 6.1(b), the cost function. These two are given as a function of s : the weights of the RNN are $\theta(s) = s\theta_{\text{true}}$ and θ_{true} is the true data generating weights.

The bifurcation diagram depicts, for each parameter, the steady-state behavior of the system. It was generated using a simulation of 200 samples for which the first 100 samples were discarded to remove the transient. In this bifurcation diagram, we can observe a region with rich nonlinear and chaotic behavior for $0.9 \lesssim s \lesssim 1.45$. This region corresponds to the region where the cost function (Figure 6.1(b)) is intricate and full of local minima. This connection between internal dynamics and smoothness of the cost function is formalized in the next subsection.

6.4.2 Theoretical results

The theorem that will be presented in this section relates the Lipschitz constant of the cost function V and of its gradient (i.e. β -smoothness) to the simulation length N and the Lipschitz constant L_f of the state-transition function \mathbf{f} .

In this work, we phrase the exploding gradient problem in terms of the smoothness and Lipschitz constants. The formulation is quite natural and allows us to include higher-order derivatives in the analysis.

The Lipschitz constant of V and of its gradient provide upper bounds, respectively, on $\|\nabla V\|$ and $\|\nabla^2 V\|$, cf. [Khalil \(2002, Lemma 3.1\)](#). Hence, the first part of the theorem below can indeed be seen as a formalization of the exploding gradient problem. The second part gives information about the explosion of second-order derivatives and curvature and, to the best of our knowledge, is novel. For the case $L_f < 1$, similar results have been presented by [Miller and Hardt \(2018\)](#), but not for the case that interests us the most: $L_f > 1$, for which there might be an explosion in the curvature.

Theorem 6.2 (Lipschitz constants of V and ∇V). *Let $\mathbf{f}(\mathbf{x}, \mathbf{u}; \boldsymbol{\theta})$ and $\mathbf{g}(\mathbf{x}, \mathbf{u}; \boldsymbol{\theta})$ in Eq. (6.2) be Lipschitz in $(\mathbf{x}, \boldsymbol{\theta})$ with constants L_f and L_g on a compact and convex set $\Omega = (\Omega_{\mathbf{x}}, \Omega_{\mathbf{u}}, \Omega_{\boldsymbol{\theta}})$. Assume the loss function is either $l(\hat{\mathbf{y}}, \mathbf{y}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2$ or $l(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y}^T \log(\sigma(\hat{\mathbf{y}})) - (1 - \mathbf{y})^T \log(1 - \sigma(\hat{\mathbf{y}}))$. Let $\{\mathbf{u}_t\}_{k=1}^N \subseteq \Omega_{\mathbf{u}}$ and $(\Omega_{\mathbf{x}}, \Omega_{\boldsymbol{\theta}}) \subseteq \mathbb{R}^{N_{\theta}}$. If there exists at least one choice of $(\mathbf{x}_0, \boldsymbol{\theta})$ for which there is an invariant set contained in Ω , then, for trajectories and parameters confined within Ω :*

1. The cost function V defined in (6.1) is Lipschitz with constant:²

$$L_V = \begin{cases} \mathcal{O}(L_f^{2N}) & \text{if } L_f > 1, \\ \mathcal{O}(N) & \text{if } L_f = 1, \\ \mathcal{O}(1) & \text{if } L_f < 1. \end{cases} \quad (6.9)$$

2. If the Jacobian matrices of \mathbf{f} and \mathbf{g} are also Lipschitz with respect to $(\mathbf{x}, \boldsymbol{\theta})$ on Ω , then the gradient

² Here \mathcal{O} is the big O notation. It should be read as: $L(N) = \mathcal{O}(g(N))$ if and only if there exist positive integers M and N_0 such that $|L(N)| \leq Mg(N)$ for all $N \geq N_0$.

of the cost function ∇V is also Lipschitz with constant:

$$L'_V = \begin{cases} \mathcal{O}(L_f^{3N}) & \text{if } L_f > 1, \\ \mathcal{O}(N^3) & \text{if } L_f = 1, \\ \mathcal{O}(1) & \text{if } L_f < 1. \end{cases} \quad (6.10)$$

Proof. In Appendix B. □

Observation 6.1 (other loss functions): The above theorem was stated for two different loss functions (squared error and average cross-entropy preceded by the sigmoid function). The theorem still holds for any loss function for which the equivalent of Lemma B.2 (see Appendix B) remains true. □

It is important to highlight that the proof of this more general result follows a different approach than a standard proof of the exploding gradient problem (such as in Pascanu et al. (2013)). Rather than backpropagating the derivatives (which is more natural in the context of neural networks), we use forward propagation (similarly to what is done by Williams and Zipser (1989)) which allows us to arrive at this more general result. The result was studied by us in (Ribeiro et al., 2019) in the context of control theory and is generalized here to RNNs.

If the function \mathbf{f} is Lipschitz and the system is non-contractive we have $L_f \geq 1$ ³. According to Theorem 6.2, the Lipschitz constants and β -smoothness for all these systems might blow up exponentially (or polynomially for some limit cases) with the maximum simulation length. This results in very intricate cost functions, see Figure 6.1 (b).

Observation 6.2 (relation between L_f and eigenvalues): Let A be the Jacobian matrix of \mathbf{f} with respect to \mathbf{x} for a fixed input $\bar{\mathbf{u}}$. The constant $L_{\bar{\mathbf{u}}}$ in the set Ω is equal to the largest eigenvalue of A at its maximum inside the set. Furthermore, $L_{\bar{\mathbf{u}}}$ is a lower bound on L_f , hence the necessary condition for exploding gradients to appear given in Pascanu et al. (2013) (i.e. largest eigenvalue bigger than 1) also follows from Theorem 6.2. The necessary condition for the second derivative to explode could also be stated in terms of the largest eigenvalue: it needs to be bigger than 1. □

6.5 Attractors evolution during training

In this section, we study the mechanism RNNs use to learn a task that requires long-term memory. We focus on the LSTM (Hochreiter and Schmidhuber, 1997); on the LSTM constrained to the contractive region of the parameter space, as proposed by Miller and Hardt (2018); and on the orthogonal RNN proposed by Lezcano-Casado and Martínez-Rubio (2019), also known as expRNN.

6.5.1 Sine wave generation

We study the use of RNNs for the **generation of sine waves with varying frequencies**. The input is a constant, and the output is a sine wave with unitary amplitude and the frequency specified by the constant input. This artificial task is described in Sussillo and Barak (2013). We have 100 sequences of length 400, each sequence consisting of a sine wave with the given constant frequency. The 100 sequences are uniformly picked in the interval $[\frac{\pi}{16}, \frac{\pi}{8}]$. We use those sequences for training.

For all models, we use the same configurations: Adam optimization algorithm (Kingma and Ba, 2014) with initial learning rate 10^{-3} and default parameters. For the stable model, we decrease the learning

³ If $\bar{\mathbf{u}} \in \Omega_{\mathbf{u}}$ it follows that $L_f \geq L_{\bar{\mathbf{u}}}$, hence if the system is not contractive $L_f \geq L_{\bar{\mathbf{u}}} \geq 1$.

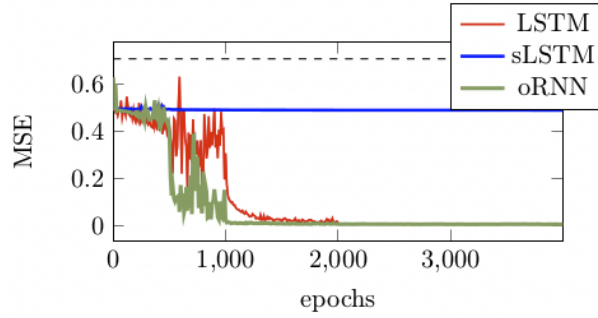


Figure 6.4 – **(Sine wave generator training history)** Mean square error per epoch for the *sine-wave generation task* during training. The baseline performance is indicated by the dashed line. The final performance of the LSTM after 4,000 epochs is $4.8 * 10^{-3}$, the final performance of the stable LSTM (sLSTM) is 0.49 and the final performance of the orthogonal RNN (oRNN) is $4.1 * 10^{-3}$.

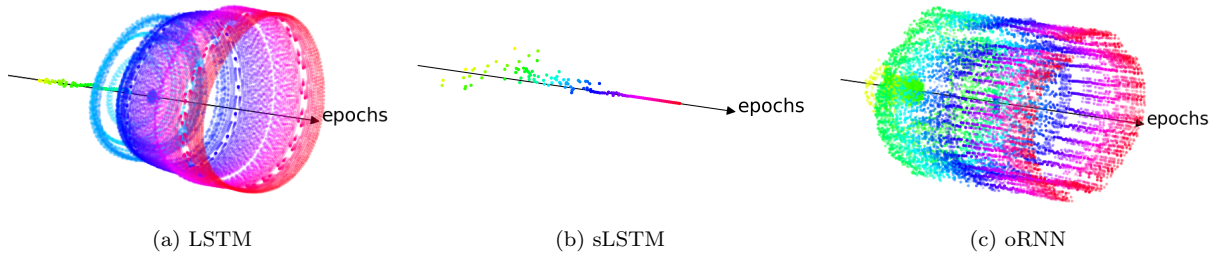


Figure 6.5 – **(The mechanism for learning oscillations)** Bifurcation diagram during training for the *sine-wave generation task* showing the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that varies from 0 to 1500. The bifurcation diagram is for a constant input $\frac{\pi}{8}$, other values will yield similar plots.

rate by 10 at epochs $\{500, 1000, 2000\}$. We use a hidden size of 200 and a single layer for the RNN, which is followed by a linear layer for the output. The gradient is clipped when its norm exceeds 0.25.

The training history for the three models is displayed in Figure 6.4. The stable LSTM fails to perform well in the **sine-generation task**. Figure 6.5 shows the bifurcation diagram for the three models during training. Here we are using a two-dimensional bifurcation diagram containing both the output y_t and its first difference $y_t - y_{t-1}$ to facilitate the visualization of periodic attractors.

For the LSTM, during training, a stable fixed point undergoes a bifurcation at which the fixed point stability switches, and a periodic solution arises. The stable LSTM is constrained to stay in the region of the parameter space for which the system is contractive and, hence has a single stable attractor point (cf. discussion on Section 6.4). Since the system needs to sustain the oscillations, this is a significant limitation that prevents the model from performing the task well.

The bifurcation that the fixed point needs to undergo to oscillate is called Hopf (for the continuous case) or Neimark-Sacker bifurcation (for the discrete case). During this bifurcation, a single stable fixed point changes stability and becomes unstable as a pair of complex eigenvalues (of the Jacobian matrix) go into the unstable region. The orthogonal constraint in the orthogonal RNN prevents the occurrence of these bifurcations since all eigenvalues are fixed to one. Nevertheless, orthogonal RNNs still manage to solve the problem and learn the periodic attractor.

Since most real-world sequence tasks have stochastic elements, the deterministic nature of the

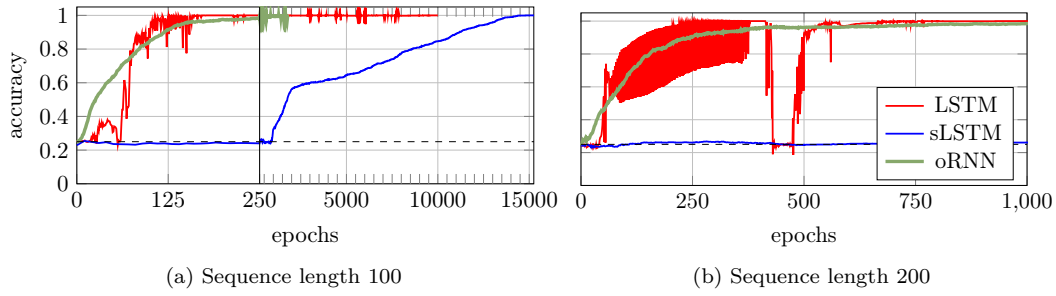


Figure 6.6 – **(Symbol classification training history)** Accuracy on *validation data set* for the recurrent models trained to perform the same *symbol classification task* for two different sequence lengths. The baseline performance (always predicting $\{p, p\}$) is indicated by the dashed line. In (a) two x-axis scales co-exist in the same graph, one scale in $[0, 250)$ and other in $[250, 15250]$, with a relation 1:40 between the two scales.

sine wave generation is a limitation of this task to be representative of real-world problems. Nevertheless, this example illustrates the challenges of learning steady-state behavior using RNNs. Stable models will not be able to store information in the form of attractors, unless other training mechanisms, such as teacher forcing, are used (see Figure 6.2). Orthogonal models, do not have this restriction, but, even so, the orthogonal constraints can also prevent some bifurcations, and require different learning mechanisms. For instance, another type of bifurcation that this model will fail to capture is the supercritical pitch-fork bifurcation, where a single stable fixed point loses stability, and two stable fixed points are created.

6.5.2 Symbol classification

We study the classification according to a few relevant, widely separated, symbols. This artificial task was originally described by Hochreiter and Schmidhuber (1997) and requires the neural network to retain information for long periods. For this problem, the sequence contains categorical values $\{p, q, a, b, c, d\}$. The symbols $\{a, b, c, d\}$ act as distractors and are not relevant to the task. Instead, the relevant symbols are picked from $\{p, q\}$ appearing only twice in the sequence, at positions t_1 and t_2 . The task of the neural network is to classify the sequence according to the order of a few relevant symbols. The four possible classes are $\{(p, p), (p, q), (q, p), (q, q)\}$. Both training and validation sequences have the same length.

For all models, we use the same configurations: Adam optimization algorithm with an initial learning rate 10^{-2} and default parameters. We decrease the learning rate by 10 at epochs $\{500, 1000, 2000\}$, and run the optimization for a total of 4000 epochs. We use a hidden size of 200 and a single layer for the RNN, which is followed by a linear layer with a single output. The gradient is clipped when its norm exceeds 0.25. The batch size is 100.

Figure 6.6 shows the training history for the different models in the **symbol classification task**. In Figure 6.6(a), the sLSTM eventually manages to solve the task, the accuracy, however, increases at a very slow linear rate between epochs $[1000, 10000]$. The linear rate is quite characteristic of first-order optimization methods in a basin of attraction (Nocedal and Wright, 2006), and it is very slow because of the vanishing gradient problem. On the other hand, the standard LSTM converges quickly. The convergence, however, is quite dependent on the initial condition. See Figure 6.7 to see how, for a different random seed and identical setup, the model may fail to converge completely.

For length 200, the LSTM converges, but very unsteadily and with many bumps. The strong dependency with the initial conditions and hard to navigate landscape is in agreement with the considerations about smoothness and multiple close local minima that might affect this model (cf. Section 6.4). The orthogonal RNN manages to avoid this hard to navigate landscape by fixing the eigenvalues equal to

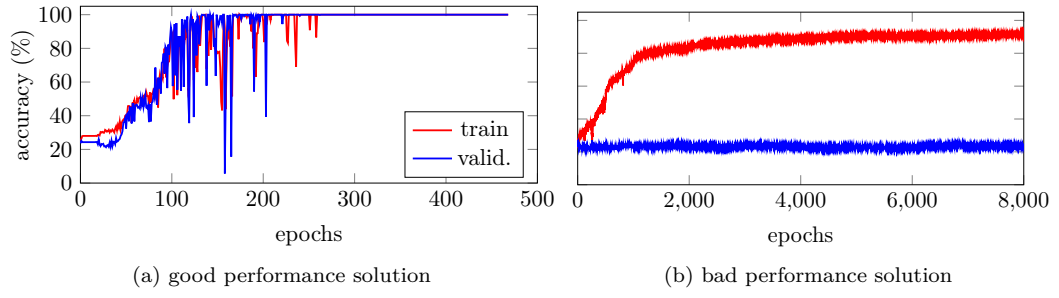


Figure 6.7 – **(The effect of the initial conditions)** Accuracy on training and validation data on the **symbol classification task** for the **LSTM** model in identical scenarios but with different initial random parameter initialization (from different random seeds). In (a), the optimization procedure abruptly finds a solution that has good accuracy on both training and validation; while, in (b), the convergence is slow and steady towards a solution that has good accuracy on the training set (above 90%) but is no better than random guessing on the validation set.

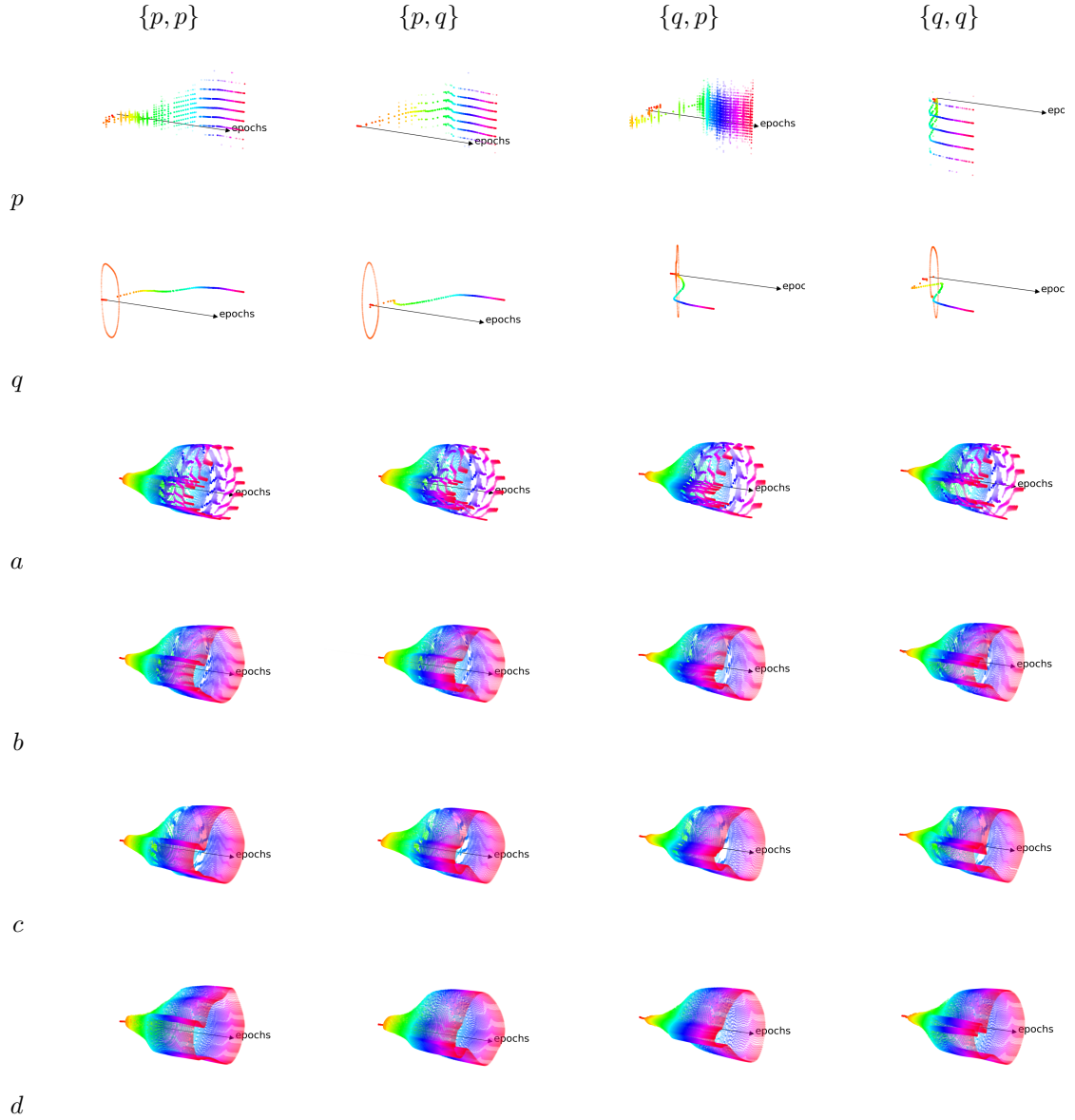


Figure 6.8 – **(The LSTM mechanism for learning to remember symbols)** Bifurcation diagram for the *LSTM* model in *symbol classification task* for sequences of length 100. It shows the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that vary from 0 to 400.

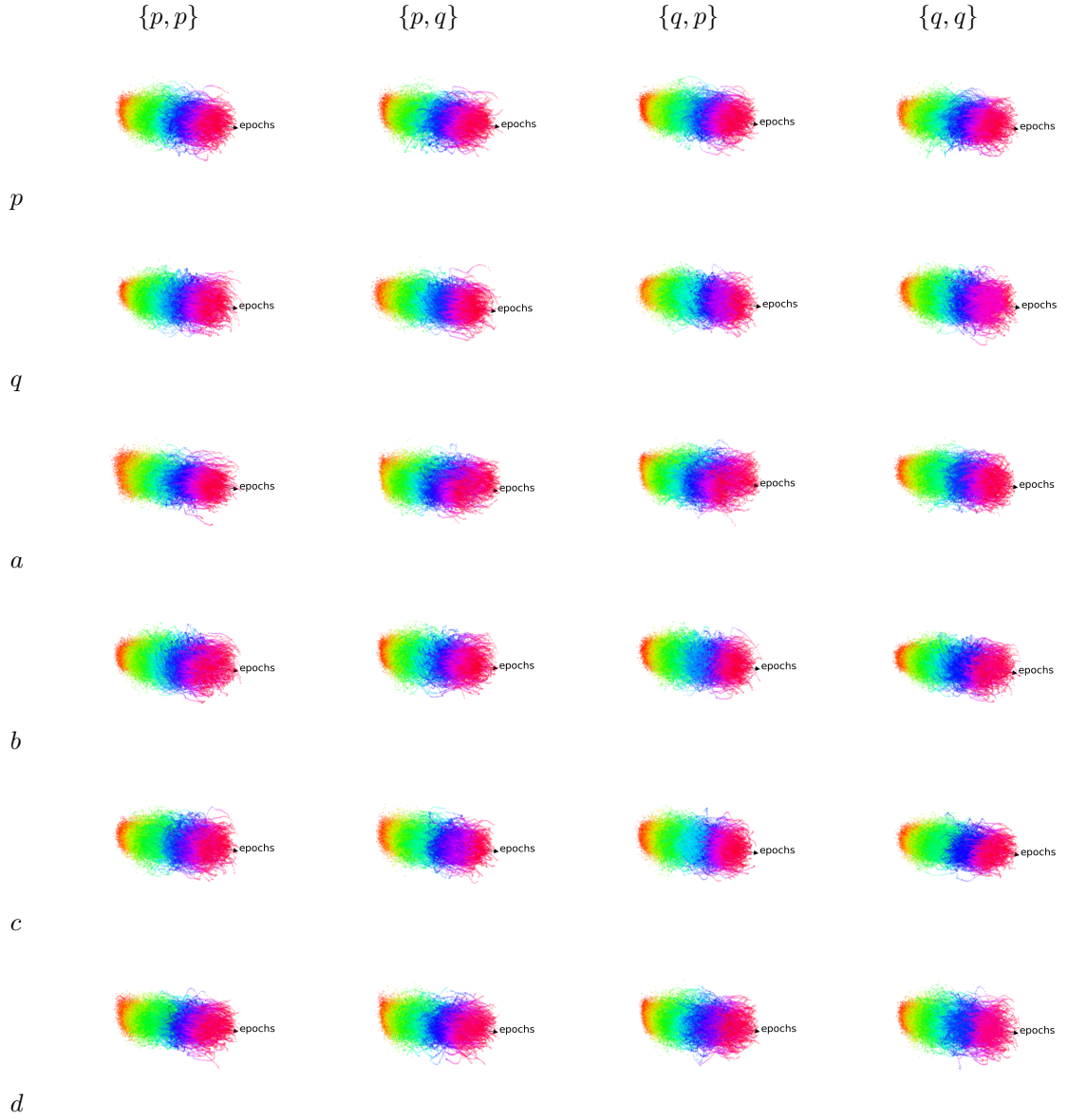


Figure 6.9 – **(The oRNN mechanism for learning to remember symbols under noise)** Bifurcation diagram for the *oRNN* model in *symbol classification task* for sequences of length 100. It shows the steady-state of the output y_t and its first difference $y_t - y_{t-1}$. The arrows point towards the evolution of the number of epochs, that vary from 0 to 400.

Table 6.1 – **(Symbol classification accuracy)** Comparison between the LSTM; stable LSTM (sLSTM); and orthogonal RNN (oRNN); for the **symbol classification task** with different sequence lengths (l). As a baseline, always predicting $\{p, p\}$ would yield an accuracy of 0.25.

l	LSTM	sLSTM	oRNN
50	1.00	1.00	1.000
100	1.00	1.00	1.000
200	1.00	0.27	0.999
300	0.25	0.26	0.995
500	0.27	0.26	0.970

one, and it does have a much smoother convergence than the LSTM. It is also the model that manages to solve the problem for the longest sequences (see Table 6.1). The stable LSTM model, on the other hand, fails to solve the task for the length of 200 or longer.

To analyze the attractors during training, we apply a constant unitary input to one of the six input channels (keeping the others at zero) and measure one of the four possible outputs. Bifurcation diagrams for all 6×4 possible combinations of input/output pairs are displayed, for the LSTM and the orthogonal RNN respectively, in Figures 6.8 and 6.9.

The LSTM goes through bifurcations, for instance: in $q \rightarrow \{q, p\}$, the number of fixed points increases, and, in $a \rightarrow \{q, p\}$, they transition to periodic behavior. We understand the ability to undergo bifurcations is useful, and having multiple fixed points helps the LSTM solving the task for longer sequences than it would be able to only within the contractive region. The comparison between LSTM and stable LSTM in Table 6.1 seems to corroborate such a hypothesis. The periodic behavior in $a \rightarrow \{q, p\}$, on the other hand, is spurious, since a and the other distractor symbols do not influence the outcome. The mechanism the orthogonal RNN uses to solve the task is quite distinct: they create a cloud of fixed points and, without suffering any bifurcation, during training, these fixed points change position in the state space. These fixed points do not disappear during training, and we believe the last layer learns to weigh them differently and use the information to solve the task.

6.5.3 Word-level language model

We train a language model on the openly available dataset Wikitext-2. This dataset contains 600 Wikipedia articles for training (2,088,628 tokens), 60 articles for validation (217,646 tokens), and 60 articles for testing (245,569 tokens) (Merity et al., 2016). The dataset has a vocabulary of 33,278 distinct tokens. The goal is to predict the next token in the article given the previous tokens. The state-of-the-art result for the Wikitext-2 data set when not extending the training set is a perplexity (lower is better) on the test set of 39.14 (Gong et al., 2018).

Since our implementation of orthogonal RNN is restricted to a single layer, we restrict all the models to a single layer. Even so, with our LSTM model implementation, we achieve a perplexity of 99.2 in the test set, which is close to other results reported in the literature (i.e. 99.3 obtained for a standard LSTM model in (Edouard Grave, 2017)). The stable LSTM has a perplexity of 118.8 (which is close to Miller and Hardt (2018), where they achieve a perplexity of 113.2) and the orthogonal LSTM has a perplexity of 185.3.

We trained a model that consists of an embedding layer outputting a size 800 vector to a recurrent unit (LSTM, stable LSTM, or orthogonal RNN) with hidden size 800, followed by a softmax decoder layer that outputs probabilities for each token in the vocabulary. The last layer and the embedding have tied layers. A dropout layer is included before and after the LSTM layer with dropout rate 0.5. The models are trained and evaluated using a cross-entropy loss. The model is trained using ADAM with betas (0.9, 0.999). The learning rate starts at 10^{-3} and is reduced by a factor 10 when the cost function plateaus

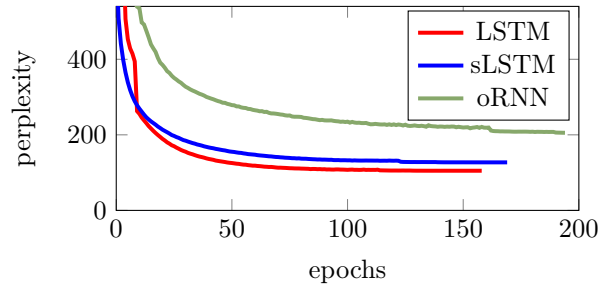


Figure 6.10 – **(Word-level language model training history)** Perplexity on *validation data set* for *word-level language model*.

for more than 7 epochs. And the training stops when the learning rate drops below 10^{-7} . The gradient norm is clipped when it exceeds 0.25. The batch size is 100, and the gradient is not backpropagated for a sequence length larger than 80.

The training history is displayed in Figure 6.10. It is interesting to contrast this training curve with the training curve of the two previous examples: in the two previous examples the LSTM training was unsteady and full of jumps (and also highly sensitive to initial conditions); here, the curve is very smooth.

Figures 6.11, 6.12 and 6.13 show the bifurcation diagram for the LSTM, stable LSTM and the orthogonal RNN, respectively. We do the bifurcation diagram for different random initial states and different constant inputs. This problem has a high-dimensional output, hence we consider different types of projections into one dimension: i) The average of all outputs, i.e. $\bar{x}_t = \frac{1}{n_{\text{outputs}}} \sum_{i=1}^{n_{\text{outputs}}} x_t^i$; and, ii) projections in the direction of some specific tokens in the embedding space. More specifically we consider the tokens “is”, “<unk>”, and “Valkyria”, which are representative of both frequent and infrequent tokens. In Figure 6.11 we also extend the projections into two dimensions by including the first difference as the second dimension. We also consider two different situations: a) We apply a constant input; and, b) We apply a feedback connection similar to that in Figure 6.3(b) using as input, at instant t , the word predicted with the highest probability at the previous time instant, $t - 1$. We show representative samples in the Figures.

When fed with constant input, LSTM presents a wide range of different qualitative behavior: from converging to a single fixed point, as in Figure 6.11(a), to going through bifurcations that result in chaotic or periodic behavior, as in Figure 6.11(b). When fed with constant input, orthogonal RNNs, on the other hand, converge to a single fixed point. This was verified for several random seeds, and we show a representative example in Figure 6.13(a) to (d). In this example, the qualitative behavior of the oRNN model is quite similar to the stable LSTM, which also has single fixed-point (by construction).

Chaotic and periodic attractors in the constant input scenario suggest that training the LSTM goes into region of the parameters space that yield non-contractive models. An experiment presented by Laurent and von Brecht (2016, sec. 2.1) shows that this chaotic behavior, however, has limited influence in the output. This might help explain why the training of the LSTM model progresses so smoothly and not bumpy and unsteady as the scenario presented in Figure 6.1 (for chaotic behavior) would suggest.

We also show the attractors for when the input, at instant t , is the word predicted with the highest probability at the previous time instant, $t - 1$. This scenario corresponds to using the model for sentence generation. LSTM present, again, chaotic attractors, which are quite desirable here, since periodic behavior or a single attractor correspond to repeating a few words or only a single word after sufficient interactions, which would not generate interesting sentences. Both orthogonal RNNs and stable

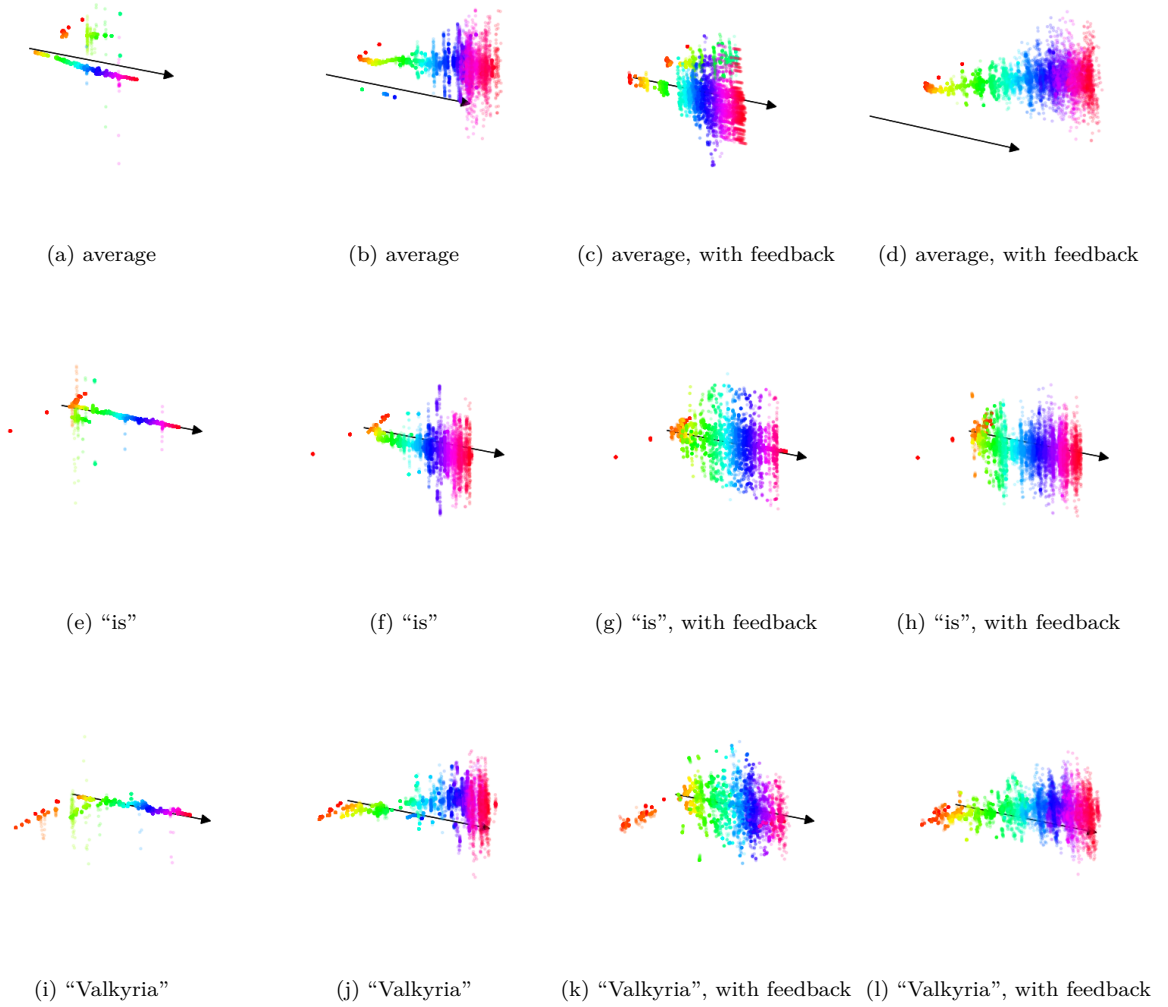


Figure 6.11 – **(The LSTM mechanism for learning a language model)** Bifurcation diagram for the *LSTM world-level language model*. For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ and its first difference $p(\mathbf{x}_t) - p(\mathbf{x}_{t-1})$ after a burnout period of 1500 samples. This burnout period is used to remove the transient reponse and yields a visualization of the system attractors, per epoch. The arrow point towards the evolution of the number of epochs, that varies from 0 to 150. In (a) and (b), we have two different realizations of the bifurcation diagram obtained from constant inputs. In (c) and (d), the diagram is generated using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in (a) and (b), respectively. The subplots (a) to (d) use the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$. The second row, (e) to (h), and third row, (i) to (l), show the exact same experiments but for the projections in the direction of the tokens "is" and "Valkyria", respectively.

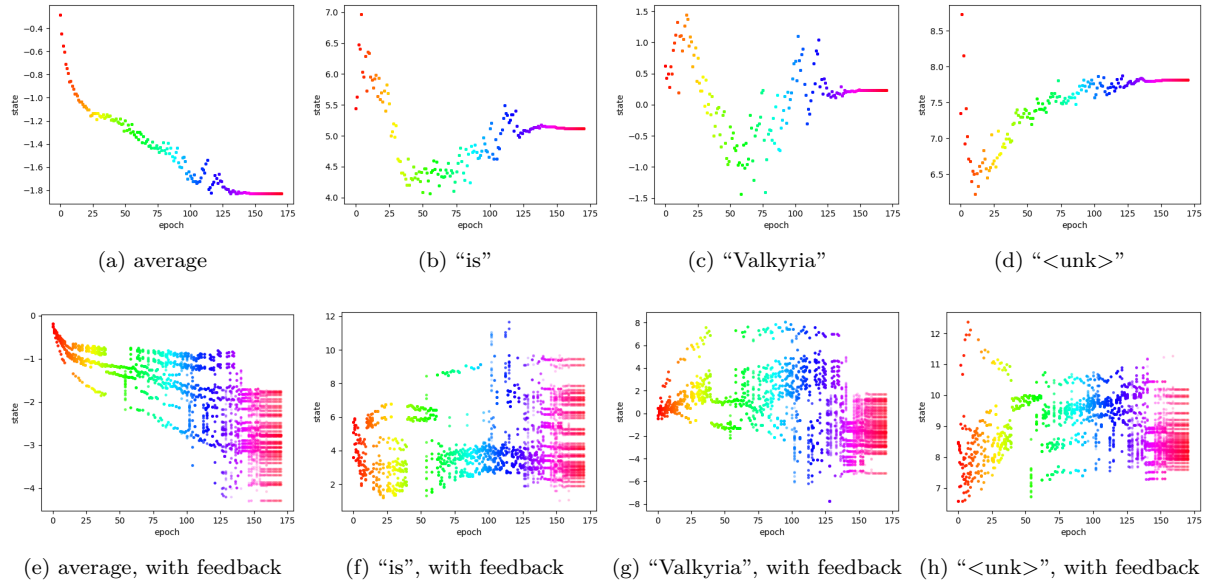


Figure 6.12 – **(The sLSTM mechanism for learning a language model)** Bifurcation diagram for the *stable LSTM world-level language model*. For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ after a burnout period of 1500 samples. This burnout period is used to remove the transient response and yields a visualization of the system attractors, per epoch. In the displays (a) to (d), the diagram is generated for the same constant input. In (e) to (h), the diagram is generated using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in the first row. The projections are: the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$; and, projections into the direction of the tokens “is”, “Valkyria” and “<unk>”.

LSTMs present this type of undesirable behavior. Not to say they are the same, in our experiments, sLSTM usually takes more than 600 tokens to reach this steady-state periodic behavior (after <eos> most of the time) while for the oRNN this sometimes happens before 100 tokens. Also, sLSTM converges to a richer periodic behavior, which oscillates between more points. This example shows how the stable LSTM, during inference, might present richer steady-state behavior due to the feedback connections.

6.6 Related work

Analyzing attractors in an RNN is not new. Both [Sussillo and Barak \(2013\)](#) and [Maheswaranathan et al. \(2019\)](#) give examples where they analyze the attractors of the RNN. While we try to use the attractors for the analysis of the training procedure, these two works are heavily focused on the after-training.

The understanding of exploding and vanishing gradients is usually attributed to ([Hochreiter and Schmidhuber, 1997](#); [Pascanu et al., 2013](#); [Bengio et al., 1994](#)). [Pascanu et al. \(2013\)](#) formulate exploding and vanishing gradients in terms of the eigenvalues of the hidden-to-hidden weight matrix. Neither of these two works, however, explicitly considers the importance of attractors in retaining the information. This is done in ([Bengio et al., 1994](#)), who explicitly consider attractors and present a trade-off between vanishing gradients and retaining information robustly in the presence of noise, using single fixed points in this analysis. We generalize the analysis in terms of any nonlinear attractor (including chaotic and periodic).

[Doya \(1993\)](#) suggests that bifurcations might be a problem in RNN training. [Pascanu et al. \(2013\)](#) presents the hypothesis that bifurcations are the origin of walls in the cost function. In our examples, we

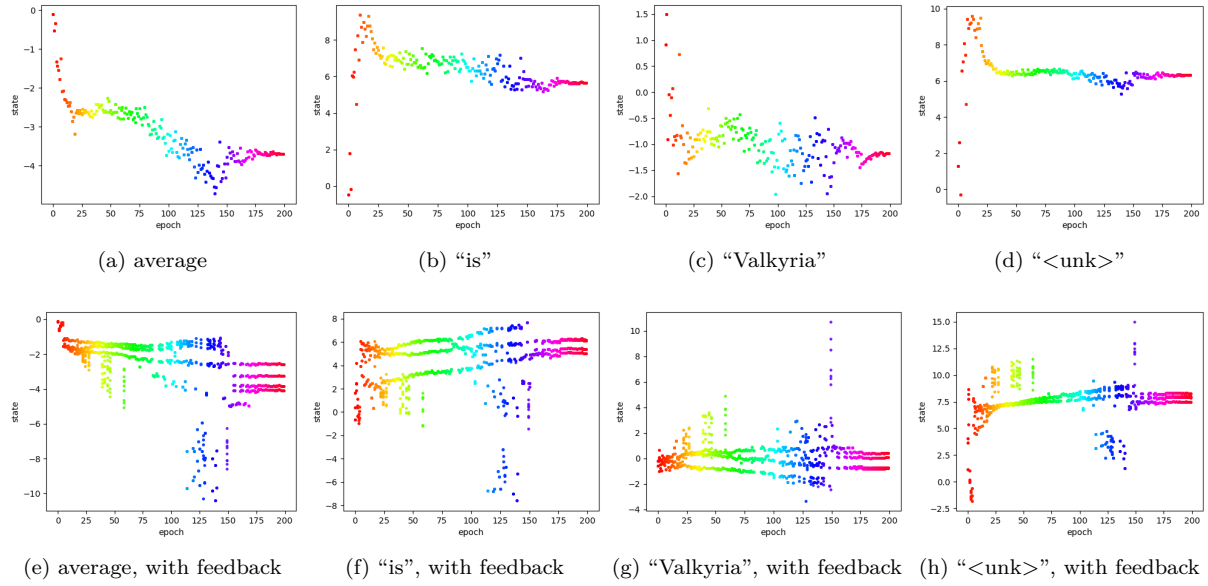


Figure 6.13 – **(The oRNN mechanism for learning a language model)** Bifurcation diagram for the *orthogonal RNN world-level language model*. For each epoch, the plot show values visited by the projections of the internal state $p(\mathbf{x}_t)$ after a burnout period of 1500 samples. This burnout period is used to remove the transient response and yields a visualization of the system attractors, per epoch. In the displays (a) to (d), the diagram is generated for the same constant input. In (e) to (h), the diagram is generated using as input the word predicted with the highest probability at the previous time instant, and using as first input to the sequence the same input as in the first row. The projections are: the average of internal states as projections, i.e. $p(\mathbf{x}_t) = \bar{x}_t$; and, projections into the direction of the tokens “is”, “Valkyria” and “<unk>”.

do not observe this connection between bifurcations and jumps in performance.

The work in (Laurent and von Brecht, 2016) presents a neural network without chaotic behavior. The results presented in Section 6.4, for which the chaotic behavior of LSTMs yield regions in parameter space where the cost function is highly intricate, is a strong suggestion that this is indeed an interesting line of research that deserves to be further explored. Also, (Laurent and von Brecht, 2016) is one of the first papers to propose an architecture based on the properties of the attractors.

The notion of using entropy for a dynamical system (as in Section 6.3) is not new. One classical definition is that of the Kolmogorov-Sinai entropy (Sinai, 1959), which is related to how uncertainty (and information) increases with time in a chaotic attractor. However, this definition cannot be applied for regions of the parameter space that do not preserve volume. The definition presented in Section 6.3 serves our purpose better since it does not introduce such restrictions.

6.7 Discussion

We believe the refinements presented here in the classical concept of exploding and vanishing gradient might be highly relevant to the development of new optimization and regularization methods for RNNs and to understand the limitations and strengths of many techniques already in use.

Constraining the RNN eigenvalues to be smaller than one has the effect of yielding exponentially decreasing transient behavior that gives rise to vanishing gradients. When the training and inference mode are exactly the same, this constraint also has the effect of not allowing attractors to appear. Much of the current RNN literature does not distinguish between the two effects. And, as shown in the experiments,

this distinction can allow for richer analysis and improved understanding of the training mechanism.

It is also important to draw a clear distinction between the RNN behavior during training and inference. As discussed in Section 6.3, attractors are needed to store information in a non-volatile way. When the training and inference mode are exactly the same, learning attractors (other than a single fixed point) require the optimization procedure to explore regions of the parameter space where the model is non-contractive, and this is hard because the objective function might have poor smoothness properties and be ill-suited for traditional optimization procedures (see Section 6.4).

The symbol classification task studied in this work is very similar to other artificial benchmarks used to assess the ability of recurrent models *remembering* information seen many times steps before. Other popular examples include the adding and multiplication problems (Hochreiter and Schmidhuber, 1997) and the copy memory task (Arjovsky et al., 2016). All these problems are quite effective benchmarks for assessing the ability of the model to learn attractors and explore regions of the parameter space where the model is non-contractive. The results in this work seem to reinforce the effectiveness of oRNN model in solving this class of problems.

This class of artificial benchmark is usually formulated in a way that makes it hard or impossible to use teacher forcing. This contrasts with many relevant tasks and real-world problems, such as word language modeling, for which state-of-the-art solutions usually employ such a mechanism. Teacher forcing introduces the appealing possibility of having different modes for training and inference, and even if the training is limited to regions of the parameter space where the dynamic system is contractive, it might still be possible to have attractors (and hence long-term memory) during inference, since the system output is fed back into its input, creating new possibilities for the dynamical behavior. Different training and inference behavior might help explain why the stable model by Miller and Hardt (2018) is successful in many tasks: on the one hand, training the model in the unstable region is challenging, so ruling out this region of the parameter space from the optimization procedure might not be so impactful; on the other hand, the model is not necessarily stable during inference (because the output might be fed back into the input).

Part III

Modeling signals and systems: applications to
engineering and health care

7 Deep convolutional networks in system identification

In Section 2.3 we presented convolutional neural networks for sequences. Our presentation focused on making explicit the connections between signal processing concepts and convolutional neural networks. Here, we focus on presenting convolutional networks from a system identification perspective, relating the architecture with traditional system identification terminology and concepts. Exploring the relationships between the convolutional neural networks, NARX models, Volterra series and block-oriented models. Here we focus on temporal convolutional neural networks (TCNs) which is a terminology introduced by (Bai et al., 2018a) to refer to unidimensional and causal convolutional neural networks. We also present an experimental study where we provide results on two real-world problems, the well-known Silverbox dataset (Wigren and Schoukens, 2013) and dataset originated from ground vibration experiments on an F-16 fighter aircraft (Schoukens and Noël, 2017).

The content of this chapter was presented at the 2019 Conference on Decision and Control (CDC) and should appear in the proceedings of the conference:

Deep Convolutional Networks in System Identification, *Carl Andersson**, *Antonio H. Ribeiro**, *Koen Tiels*, *Niklas Wahlström* and *Thomas B. Schön* (* Equal contribution).
 Proceedings of the 58th IEEE Conference on Decision and Control (CDC), 2019. pp. 3670–3676.
 doi: 10.1109/CDC40024.2019.9030219

The paper was also presented, as a poster, at the *European Research Network on System Identification, ERNSI*, (2019) and at the *Nonlinear System Identification Benchmarks Workshop* (2019), as an oral presentation under the name *Deep Convolutional Networks are useful in System Identification*.

The outline of the chapter is the following: We will describe the new TCN model from a system identification point of view in Section 7.1, focusing in highlighting the connections with NARX models. Additionally, in Section 7.2, we will show that there are indeed interesting connections between the deep TCN structure and the Volterra series and the block-oriented model structures commonly used within system identification. Perhaps most importantly, in Section 7.3 we will provide experimental results on two real-world problems and on a toy problem. Final remarks are provided in 7.4

7.1 Neural networks for temporal modeling

The neural network is a universal function approximator (Hornik et al., 1989b) with a sequential model structure of the form:

$$\hat{y} = g^{(L)}(z^{(L-1)}), \quad (7.1a)$$

$$z^{(l)} = g^{(l)}(z^{(l-1)}), \quad l = 1, \dots, L-1, \quad (7.1b)$$

$$z^{(0)} = x, \quad (7.1c)$$

where x , $z^{(l)}$, \hat{y} denotes the input, the hidden variables and the output, respectively. The transformation within each layer is of the form $g^{(l)}(z) = \sigma(W^{(l)}z + b^{(l)})$ consisting of a linear transformation $W^{(l)}z + b^{(l)}$ followed by a scalar nonlinear mapping, σ , that acts element-wise. In the final (output) layer the nonlinearity is usually omitted, i.e. $g^{(L)}(z) = W^{(L)}z + b^{(L)}$.

The neural network parameters $\{W^{(l)}, b^{(l)}\}_{l=1}^L$ are usually referred to as the weights $W^{(l)}$ and the bias terms $b^{(l)}$ and they are estimated by minimizing the prediction error $\frac{1}{N} \sum_{k=1}^N \|\hat{y}[k] - y[k]\|^2$ for some training dataset $\{x[k], y[k]\}_{k=1}^N$.

To train deep neural networks with many hidden layers (large L) have been proved to be a notoriously hard optimization problem. The challenges includes the risk of getting stuck in bad local minimas, exploding and/or vanishing gradients, and dealing with large-scale datasets. It is only over the past decade that these challenges have been addressed, with improved hardware and algorithms, to the extent that training truly deep neural networks has become feasible. We will very briefly review some of these developments below. Additional information can be found in Appendix A.

7.1.1 Temporal convolutional network

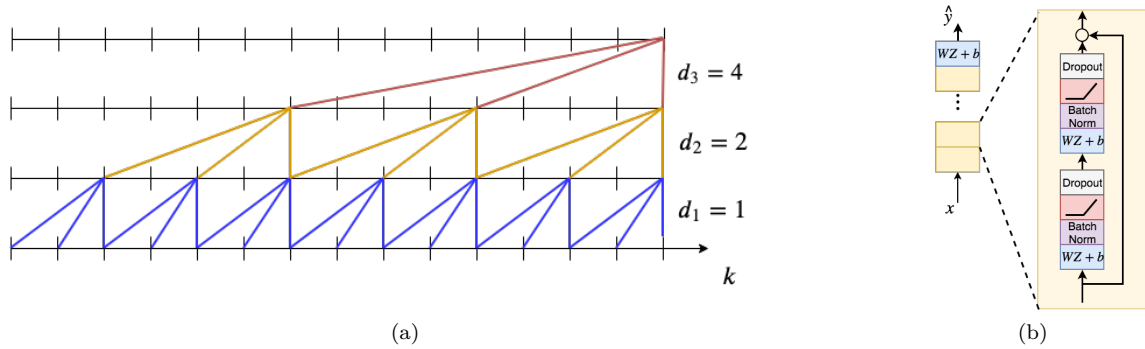


Figure 7.1 – **(Temporal convolution network)** Illustrate the temporal convolution network (TCN) with residual blocks. (a) Temporal convolutional network with dilated causal convolutions with dilation factor $d_1 = 1$, $d_2 = 2$ and $d_3 = 4$ and kernel size $n = 3$. (b) A TCN residual block. Each block consists of two dense layers and an identity (or linear) map on the skip connection. As illustrated in (a) by connection with the same color, neural network weights are shared within the same layer and invariant to time translations. This reflects the hypothesis we are modeling a time invariant system.

As the name suggests, the temporal convolutional network (TCN) is based on convolutions (Bai et al., 2018a). The use of TCNs within a system identification setting can in fact be interpreted as using the nonlinear ARX model as the basic model component:

$$\hat{y}[k+1] = g(x[k], \dots, x[k-(n-1)]), \quad (7.2)$$

with $x[k] = (u[k], y[k])$. We will proceed with this interpretation, where (7.2) would correspond to a one-layer TCN model.

A full TCN can be understood as a sequential construction of several nonlinear ARX models stacked on top of each other:

$$\hat{y}[k+1] = g^{(L)}(Z^{(L-1)}[k]), \quad (7.3a)$$

$$z^{(l)}[k] = g^{(l)}(Z^{(l-1)}[k]), \quad l = 1, \dots, L-1, \quad (7.3b)$$

$$z^{(0)}[k] = x[k], \quad (7.3c)$$

where:

$$Z^{(l-1)}[k] = \left(z^{(l-1)}[k], z^{(l-1)}[k-d_1], \dots, z^{(l-1)}[k-(n-1)d_l] \right).$$

The number of layers L , the size of each intermediate layer $z^{(l)}[k]$, and the model order n , are all design choices determined by the user. This will also determine the number of parameters included in the model.

For each layer, we optionally introduce a *dilation factor* d_l . With $d_l = 1$ we recover the standard nonlinear ARX model in each layer. With $d_l > 1$ the corresponding output of that layer can represent a wider range of past time instances. The effective memory of that layer will be $(n - 1)d_l$. Typically the dilation factor is chosen to increase exponentially with the number of layers, for example $d_l = 2^{(l-1)}$, see Fig. 7.1a. If we assume that we have the same number of parameters in each layer, the memory will increase exponentially, not only with the number of layers, but also with the number of parameters in the model. This is a very attractive but yet uncommon property for system identification models present in the literature.

Each layer in a TCN can also be seen as dilated causal convolution where n would be the kernel size and $\dim(z^{(l)}[k])$ the number of channels in layer l . These convolutions can be efficiently implemented in a vectorized manner where many computations are reused across the different time steps k . Analogously to what is done in convolutional neural networks we use zero-padding for $z^{(l)}[k]$ where $k < 1$. We refer to (Bai et al., 2018a) for a presentation of TCN based on convolutions.

7.1.2 Regularization

Similar to other approaches within system identification, L2- and L1-regularization are commonly used to reduce the flexibility of a model and hence avoid overfitting. A number of other regularization techniques have also appeared more specialized to neural networks. One of them is the dropout (Srivastava et al., 2014) which is a technique where a random subset of the hidden units in each layer is set to zero during training. New random subsets are drawn and set to zero in each optimization step which effectively means that a random subnetwork is trained during each iteration.

Data augmentation is very common in classification problems and it can also be interpreted as a regularization technique. It is used to artificially increase the training dataset by utilizing the fact that the class is invariant under some transformation of the input (e.g. translation for image) or in the presence of some low intensity noise (e.g. salt pepper noise for images).

Finally, early stopping is a pragmatic approach in which, as the name suggests, the optimization algorithm is interrupted before convergence. The stopping point is chosen as the point where a validation error is minimized. Hence, it avoids overfitting and can as such also be interpreted as a regularization technique.

7.1.3 Batch normalization

Before training a neural network, the inputs are commonly normalized by subtracting the mean and dividing by the variance. The purpose of this is to avoid early saturation of the activation function and assuring that values in the proceeding layers are within the same dynamic range. In deep networks it is beneficial to not only normalize the input layer, but also the intermediate hidden layers. This idea is exploited in batch normalization (Ioffe and Szegedy, 2015) which, in addition to this normalization, introduces scaling parameters γ and a shift β to be learned during training. The output of the layer is then:

$$\tilde{z}^{(l)}[k] = \gamma \bar{z}^{(l)}[k] + \beta, \quad (7.4)$$

where $\bar{z}^{(l)}[k]$ is normalized version of layer l output. The parameters γ and β will be trained jointly with all other parameters of the network. Batch normalization has become very popular in deep learning models.

An alternative to batch normalization is weight normalization which is, essentially, a reparametrization of the weight matrix, decoupling the magnitude and direction of the weights (Salimans and Kingma, 2016).

7.1.4 Optimization algorithms

Neural networks are trained using gradient-based optimization methods. At each iteration only a random subset of the training data is used to compute the gradient and update the parameters. This is called mini-batch gradient descent and is a crucial component for efficient training of a neural network when the dataset is large.

Multiple extensions to mini-batch gradient descent have been proposed to make the learning more efficient. Momentum (Qian, 1999) applies a first order low-pass filter to the stochastic gradients to compensate for the noise introduced by the random sub-sampling. RMSprop (Tieleman and Hinton, 2012) uses a low-passed version of the squared gradients to scale the learning rate in the different dimensions. One of the most popular optimization method today is referred to as Adam (Kingma and Ba, 2014) which basically amounts to using RMSprop with momentum.

7.1.5 The residual block

A residual block is a combination of possibly several layers together with a skip connection

$$z^{(l+p)} = \mathcal{F}(z^{(l)}) + z^{(l)}, \quad (7.5)$$

where the skip connection adds the value from the input of the block to its output. The purpose of the residual block is to let the layers learn deviations from the identity map rather than the whole transformation. This property is beneficial, especially in deep networks where we want the information to gradually change from the input to the output as we proceed through the layers. There is also some evidence that this makes it easier to train deeper neural networks (He et al., 2016a).

We employ residual blocks in our models by following the model structure in (Bai et al., 2018a). Each block consist of one skip connection and two linear mappings, each of them followed by batch normalization (Ioffe and Szegedy, 2015), activation function and dropout regularization (Srivastava et al., 2014). See Fig. 7.1b for a visual description and Appendix A for a brief explanation of batch normalization and regularization methods. For both mappings a common dilation factor is used and hence the whole block can be seen as one of the layers $g^{(l)}(z)$ in the TCN model (7.3). Note that the skip connection only passes $z^{(l-1)}[k]$ to the next layer and not the whole $Z^{(l-1)}[k]$ for each time instance k . In cases where $z^{(l-1)}[k]$ and $z^{(l)}[k]$ are of different dimensions, a linear mapping is used between them. The coefficients of this linear mapping are also learned during training.

7.2 Connections to system identification

This section describes equivalences between the basic TCN architecture (i.e. without dilated convolutions and skip connections) and models in the system identification community, namely Volterra series and block-oriented models. The discussion is limited to the nonlinear FIR case (where $x = u$) instead of the more general NARX case ($x = (u, y)$) considered in (7.2), and to single input single output systems.

7.2.1 Connection with Volterra series

A Volterra series Schetzen (2006) can be considered as a Taylor series with memory. It is essentially a polynomial in (delayed) inputs $u[k], u[k-1], \dots$. Alternatively, a Volterra series can be considered as a nonlinear generalization of the impulse response $h_1[\tau]$. The output of a Volterra series is obtained using higher-order convolutions of the input with the Volterra kernels $h_d[\tau_1, \dots, \tau_d]$ for $d = 0, 1, \dots, D$. These kernels are the polynomial coefficients in the Taylor series.

The basic TCN architecture is essentially the same as the time delay neural network (TDNN) in Waibel et al. (1989), except for the zero padding Bai et al. (2018a) and the use of ReLU activations instead of sigmoids. The TDNN has been shown to be equivalent to an infinite-degree ($D \rightarrow \infty$) Volterra series in Wray and Green (1994). This connection is made explicit in Wray and Green (1994) by showing how to compute the Volterra kernels from the estimated network weights W . The key ingredient is to use a Taylor series expansion of the activation functions σ either around zero (the bias term b is then considered part of the activation function) or alternatively around the bias values if the Taylor series around zero does not converge for example.

7.2.2 Connection with block-oriented models

Block-oriented models Giri and Bai (2010); Schoukens and Tiels (2017) combine linear time-invariant (LTI) subsystems (or blocks) and nonlinear static (i.e. memoryless) blocks. For example, a Wiener model consists of the cascade of an LTI block and a nonlinear static block. For a Hammerstein model, the order is reversed: it is a nonlinear block followed by an LTI block. Generalizations of these simple structures are obtained by putting more blocks in series (as in Wills and Ninness (2012) for Hammerstein systems) or in parallel branches (as in Schoukens et al. (2015) for Wiener-Hammerstein systems) and/or to consider multivariate nonlinear blocks.

A multi-layer basic TCN can be considered as cascading parallel Wiener models, one for each hidden layer, that have multivariate nonlinear blocks consisting of the activation functions (including the bias). The linear output layer corresponds to adding FIR filters at the end of each parallel branch. The layers can be squeezed together to form less but larger layers (cf. squeezing together the sandwich model discussed in Palm (1979)). This is so since the dynamics consist of time delays and time delays can be placed before or after a static nonlinear function without changing the resulting output ($q^{-1}\sigma(z[k]) = \sigma(q^{-1}z[k]) = \sigma(z[k-1])$). The TCN model could be squeezed down to a parallel Wiener model.

7.2.3 Conclusion

The basic TCN architecture is equivalent to Volterra series and parallel Wiener models. They are thus all universal approximators for time-invariant systems with fading memory Boyd and Chua (1985b). This equivalence does *not* mean that all these model structures can be trained with equal ease and will perform equally well in all identification tasks. For example, a Volterra series uses polynomial basis functions, whereas TDNNs use sigmoids and TCNs use ReLU activation functions. Depending on the system at hand, one basis function might be better suited than another to avoid bad local minima and/or to obtain both an accurate and sparse representation.

7.3 Numerical results

We now present the performance of the TCN model on three system identification problems. We compare this model with the classical NARX Multilayer Perceptron (MLP) network with two layers and with the Long Short-Term Memory (LSTM) network. When available, results from other papers on the same problem are also presented.

We make a distinction between *training*, *validation* and *test* datasets. The *training* dataset is used for estimating the parameters. The performance in the *validation* data is used as the early stopping criteria for the optimization algorithm and for choosing the best hyper-parameters (i.e. neural network number of layers, number of hidden nodes, optimization parameter, and so on). The *test* data allows us

to assess the model performance on unseen data. Since the major goal of the first example is to compare different hyper-parameter choices we do not use a *test* set.

In all the cases, the neural network parameters are estimated by minimizing the mean square error using the Adam optimizer [Kingma and Ba \(2014\)](#) with default parameters and an initial learning rate of $\text{lr} = 0.001$. The learning rate is reduced whenever the validation loss does not improve for 10 consecutive epochs.

We use the Root Mean Square Error ($\text{RMSE} = \sqrt{\frac{1}{N} \sum_{k=1}^N \|\hat{y}[k] - y[k]\|^2}$) as metric for comparing the different methods in the validation and test data. Throughout the text we will make clear when the predicted output \hat{y} is computed through the free-run simulation of the model and when it is computed through the one-step-ahead prediction.

The code for reproducing the examples is available at <https://github.com/antonior92/sysid-neuralnet>. Additional information about the hyperparameters and training time can be found in Appendix B.

7.3.1 Example 1: nonlinear toy problem

The nonlinear system [Chen et al. \(1990b\)](#):

$$\begin{aligned} y^*[k] &= (0.8 - 0.5e^{-y^*[k-1]^2})y^*[k-1] - \\ &\quad (0.3 + 0.9e^{-y^*[k-1]^2})y^*[k-2] + u[k-1] + \\ &\quad 0.2u[k-2] + 0.1u[k-1]u[k-2] + v[k], \\ y[k] &= y^*[k] + w[k], \end{aligned} \tag{7.6}$$

was simulated and the generated dataset was used to build neural network models. Fig. 7.2 shows the validation results for a model obtained for a training and validation set generated with white Gaussian process noise v and measurement noise w . In this section, we repeat this same experiment for different neural network architectures, with different noise levels and different training set sizes N .

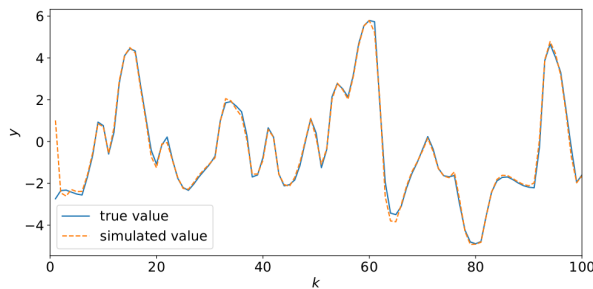


Figure 7.2 – **(Toy problem validation)** Displays 100 samples of the free-run simulation TCN model vs the simulation of the true system. The kernel size for the causal convolutions is 2, the dropout rate is 0, it has 5 convolutional layers and a dilation rate of 1. The training set has 20 batches of 100 samples and was generated with (7.6) for v and w white Gaussian noise with standard deviations $\sigma_v = 0.3$ and $\sigma_w = 0.3$. The validation set has 2 batches of 100 samples. For both, the input u is randomly generated with a standard Gaussian distribution, each randomly generated value held for 5 samples.

The best results for each neural network architecture on the validation set are compared in Table 7.1. It is interesting to see that when few samples ($N = 500$) are available for training, the TCN performs the best among the different architectures. On the other hand, when there is more data ($N = 8000$) the other architectures give the best performance.

Table 7.1 – **(Performance vs dataset size vs noise intensity)** One-step-ahead RMSE on the validation set for the models (MLP, LSTM and TCN) trained on datasets generated with: different noise levels (σ) and lengths (N). The standard deviation of both the process noise v and the measurement noise w is denoted by σ . We report only the best results among all hyper-parameters and architecture choices we have tried out for each entry.

σ	N=500			N=2 000			N=8 000		
	LSTM	MLP	TCN	LSTM	MLP	TCN	LSTM	MLP	TCN
0.0	0.362	0.270	0.254	0.245	0.204	0.196	0.165	0.154	0.159
0.3	0.712	0.645	0.607	0.602	0.586	0.558	0.549	0.561	0.551
0.6	1.183	1.160	1.094	1.105	1.070	1.066	1.038	1.052	1.043

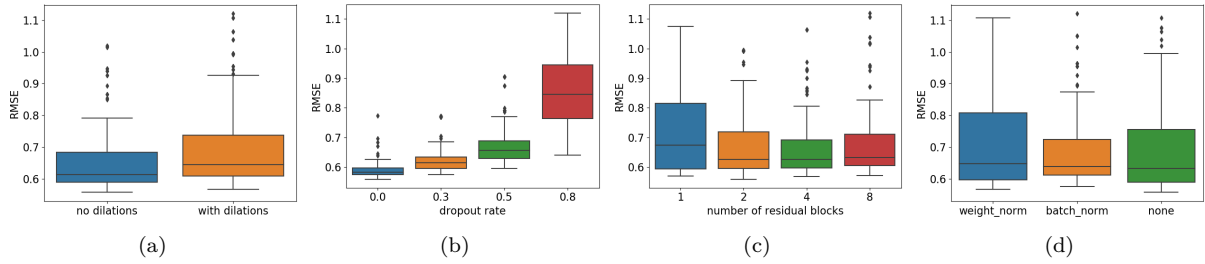


Figure 7.3 – **(Ablation study)** Box plots showing how different design choices affect the performance of the TCN for noise standard deviation $\sigma = 0.3$ and training data length $N = 2\,000$. On the y -axis the one-step-ahead RMSE on the validation set is displayed, and on the x -axis we have: in (a) the presence or absence of dilations; in (b) the dropout rate $\{0.0, 0.3, 0.5, 0.8\}$; in (c) the number of residual blocks $\{1, 2, 4, 8\}$; and, in (d) if *batch norm*, *weight norm* or nothing is used for normalizing the output of each convolutional layer. The variation in performance for the box plot quartiles is achieved through the variation for all the other hyper-parameters not fixed by the hyper-parameter choice indicated on the x -axis.

Fig. 7.3 shows how different hyper-parameter choices impact the performance of the TCN. We note that standard deep learning techniques such as dropout, batch normalization and weight normalization did not improve performance. The use of dropout actually hurts the model performance on the validation set. Furthermore, increasing the depth of the neural network does not actually improve its performance and the TCN yields better results in the training set without the use of dilations, which makes sense considering that this model does not require a long memory since the data were generated by a system of order 2.

7.3.2 Example 2: Silverbox

The Silverbox is an electronic circuit that mimics the input/output behavior of a mass-spring-damper with a cubic hardening spring. A benchmark dataset is available through [Wigren and Schoukens \(2013\)](http://www.nonlinearbenchmark.org/#Silverbox).¹

The training and validation input consists of 10 realizations of a random-phase multisine. Since the process noise and measurement noise is almost nonexistent in this system, we use all the multisine realizations for training data, simply training until convergence.

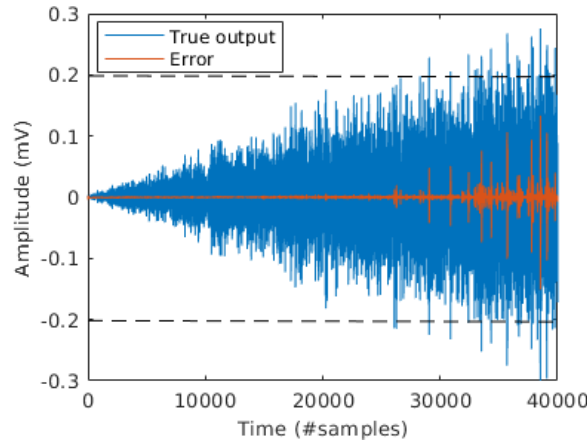
The test input consists of 40 400 samples of a Gaussian noise with a linearly increasing amplitude. This leads to the variance of the test output being larger than the variance seen in the training and validation dataset in the last third of the test data, hence the model needs to extrapolate in this region.

¹ Data available for download at:
<http://www.nonlinearbenchmark.org/#Silverbox>

Table 7.2 – **(Silverbox benchmark with no extrapolation)** Free-run simulation results for the Silverbox example on part of the test data (avoiding extrapolation).

RMSE (mV)	Which samples	Approach	Reference
0.7	first 25 000	Local Linear State Space	Verdult (2004)
0.24	first 30 000	NLSS with sigmoids	Marconato et al. (2012)
1.9	400 to 30 000	Wiener-Schetzen	Tiels (2015)
0.31	first 25 000	LSTM	this work
0.58	first 30 000	LSTM	this work
0.75	first 25 000	MLP	this work
0.95	first 30 000	MLP	this work
0.75	first 25 000	TCN	this work
1.16	first 30 000	TCN	this work

Fig. 7.4 visualizes this extrapolation problem and Table 7.2 shows the RMSE only in the region where no extrapolation is needed. The corresponding RMSE for the full dataset is presented in Table 7.3. Similarly to Section 7.3.1 we found that the TCN did not benefit from the standard deep learning techniques such as dropout and batch normalization. We also see that the LSTM outperforms the MLP and the TCN suggesting the Silverbox data is large enough to benefit of the increased complexity of the LSTM.

Figure 7.4 – **(Silverbox prediction error and data)** The true output and the prediction error of the TCN model in free-run simulation for the Silverbox data. The model needs to extrapolate approximately outside the region ± 0.2 marked by the dashed lines.

7.3.3 Example 3: F-16 ground vibration test

The F-16 vibration test was conducted on a real F-16 fighter equipped with dummy ordnances and accelerometers to measure the structural dynamics of the interface between the aircraft and the ordnance. A shaker mounted on the wing was used to generate multisine inputs to measure this dynamics.

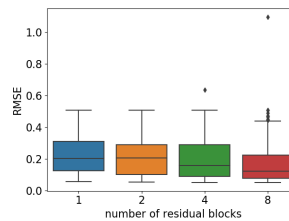
Figure 7.5 – **(TCN depth vs performance)** Box plot showing how different depths of the neural network affects the performance of the TCN in the Example 3. Should be interpreted in the same way as Fig. 7.3.

Table 7.3 – **(Silverbox benchmark)** Free-run simulation results for the Silverbox example on the full test data. (*Computed from FIT=92.2886%).

RMSE (mV)	Approach	Reference
0.96	Physical block-oriented	Hjalmarsson and Schoukens (2004)
0.38	Physical block-oriented	Paduart et al. (2004)
0.30	Nonlinear ARX	Ljung et al. (2004)
0.32	LSSVM with NARX	Espinoza et al. (2004)
1.3	Local Linear State Space	Verdult (2004)
0.26	PNLSS	Paduart (2008)
13.7	Best Linear Approximation	Paduart (2008)
0.35	Poly-LFR	Van Mulders et al. (2013)
0.34	NLSS with sigmoids	Marconato et al. (2012)
0.27	PWL-LSSVM with PWL-NARX	Espinoza et al. (2005)
7.8	MLP-ANN	Sragner et al. (2004)
4.08*	Piece-wise affine LFR	Pepona et al. (2011)
9.1	Extended fuzzy logic	Sabahi and Akbarzadeh-T (2016)
9.2	Wiener-Schetzen	Tiels (2015)
3.98	LSTM	this work
4.08	MLP	this work
4.88	TCN	this work

We used the multisine realizations with random frequency grid with 49.0 N RMS amplitude Schoukens and Noël (2017) for training, validating and testing the model.²

We trained the TCN, MLP and LSTM networks for all the same configurations used in Example 1. The analysis of the different architecture choices for the TCN in the validation set again reveals that common deep learning techniques such as dropout, batch normalization, weight normalization or the use of dilations do not improve performance. The major difference here is that the use of a deeper neural network actually outperforms shallow neural networks (Fig. 7.5).

The best results for each neural network architecture are compared in Table 7.4 for free-run simulation and one-step-ahead prediction. The results are averaged over the 3 outputs. The TCN performs similar to the LSTM and the MLP.

An earlier attempt on this dataset with a polynomial nonlinear state-space (PNLSS) model is reported in Tiels (2017). Due to the large amount of data and the large model order, the complexity of the PNLSS model had to be reduced and the optimization had to be focused in a limited frequency band (4.7 to 11 Hz). That PNLSS model only slightly improved on a linear model. Compared to that, the LSTM, MLP, and TCN perform better, also in the frequency band 4.7 to 11 Hz. This can be observed in Fig. 7.6, which compare the errors of these models with the noise floor and the total distortion level (= noise + nonlinear distortions), computed using the robust method Pintelon and Schoukens (2012). Around the main resonance at 7.3 Hz (the wing-torsion mode Schoukens and Noël (2017)), the errors of the neural networks are significantly smaller than the total distortion level, indicating that the models do capture significant nonlinear behavior. Similar results are obtained in free-run simulation (not shown here). In contrast to the PNLSS models, the neural networks did not have to be reduced in complexity. Due to the mini-batch gradient descent, it is possible to train complex models on large amounts of data.

7.3.4 Hyperparameter search and training time

All examples run with hardware acceleration provided by a single graphical processing unit (GPU). We run different experiments in machines with different configurations so the times are not directly comparable. Some of these machines have a NVIDIA Titan V and others a NVIDIA GTX 1080TI.

² Data available for download at:
<http://www.nonlinearbenchmark.org/#F16>

Table 7.4 – **(Performance in F-16 benchmark)** RMSE for free-run simulation and one-step-ahead prediction for the F16 example averaged over the 3 outputs. The average RMS value of the 3 outputs is 1.0046.

Mode	LSTM	MLP	TCN
Free-run simulation	0.74	0.48	0.63
One-step-ahead prediction	0.023	0.045	0.034

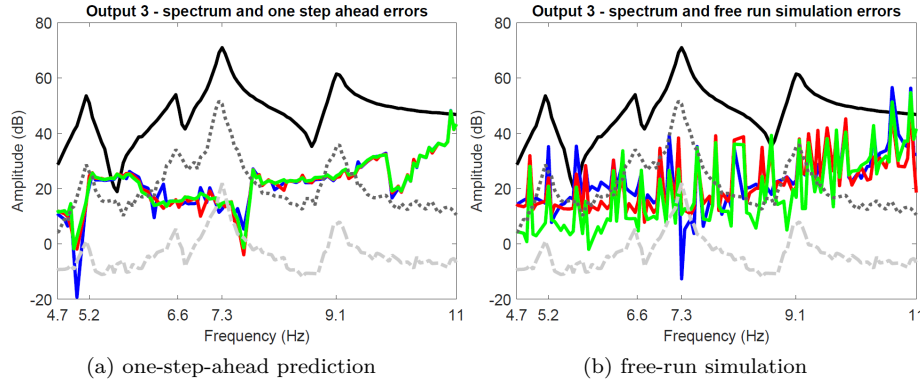


Figure 7.6 – **(F-16 model main resonance)** We plot the model frequency response in the interval $[4.7, 11]$ Hz. The true output spectrum is in black, the noise distortion, in grey dash-dotted line, the total distortion (= noise + nonlinear distortions), in grey dotted line, error LSTM, in green, error MLP, in blue, and error TCN, in red). All tested model structures perform similar and the error is close to the noise floor around the main resonance at 7.3 Hz.

A in depth analysis of the training time is beyond the scope of this chapter. The idea is to provide some basic notion of how much time is needed to run the neural network and the computational cost of doing hyperparameter search. For example 2, we provide the total time needed to do the hyperparameter search. It should be noticed, however, that grid search is an inefficient procedure. We choose to use it in order to study the effect of hyperparameters, rather than because of its efficiency. And, for example 3, we provide the total time for training the neural network with the best possible configuration.

7.3.4.1 Nonlinear toy problem

We used grid search for finding the hyperparameters. In each possible training configuration, we have trained the TCN for all possible combinations of: number of hidden layers in $\{16, 32, 64, 128, 256\}$; dropout rate in $\{0.0, 0.3, 0.5, 0.8\}$; number of residual blocks in $\{1, 2, 4, 8\}$; for the kernel size in $\{2, 4, 8, 16\}$; for the presence or absence of dilations; and, for the use of *batch norm*, *weight norm* or nothing after each convolutional layer. We have trained the MLP for all combination of: number of hidden layers in $\{16, 32, 64, 128, 256\}$; model order n in $\{2, 4, 8, 16, 32, 64, 128\}$; activation function in $\{\text{ReLU}, \text{sigmoid}\}$. Finally, we trained the LSTM for all combinations of: number of hidden layers in $\{16, 32, 64, 128\}$; number of stacked LSTM layers in $\{1, 2, 3\}$; dropout rate in $\{0.0, 0.3, 0.5, 0.8\}$. The best hyperparameters for each configuration are described in Table 7.5. For the TCN, it is better (in all configurations) to use no dilation, no normalization and kernel size equals to 2, hence these hyperparameters are omitted from the table.

7.3.4.2 Silverbox

Some hyperparameters were just experienced with manually to find good values and did not effect the results in any major fashion. For LSTM and MLP, dropout was disabled this way.

Table 7.5 – **(Toy problem model hyperparameters)** Best model hyperparameters in Example 1 for: different noise levels (σ) and lengths (N). The standard deviation of both the process noise v and the measurement noise w is denoted by σ .

(a) **TCN**: The hyperparameters are the dropout rate (drop.), the number of layers (n. layers and the number of hidden layers (h. size).

σ	N=500			N=2 000			N=8 000		
	drop.	n. layers	h. size	drop.	n. layers	h. size	drop.	n. layers	h. size
0.0	0.0	2	256	0.0	2	64	0.0	4	32
0.3	0.3	8	128	0.0	2	128	0.0	8	16
0.6	0.0	8	256	0.0	1	64	0.0	8	16

(b) **MLP**: The hyperparameters are the activation function (activ. fun.), the number of hidden layers (h. size) and the model order n .

σ	N=500			N=2 000			N=8 000		
	activ. fun.	h. size	n	activ. fun.	h. size	n	activ. fun.	h. size	n
0.0	relu	128	2	relu	128	3	relu	256	3
0.3	relu	256	2	sigmoid	64	3	relu	128	3
0.6	relu	256	2	sigmoid	128	4	relu	128	3

(c) **LSTM**: The hyperparameters are the dropout rate (drop.), the number of hidden layers (h. size) and the number of stacked layers (n. layers).

σ	N=500			N=2 000			N=8 000		
	drop.	h. size	n. layers	drop.	h. size	n. layers	drop.	h. size	n. layers
0.0	0.0	128	2	0.0	32	1	0.0	32	2
0.3	0.3	128	2	0.0	64	3	0.0	64	2
0.6	0.0	128	3	0.0	128	3	0.3	64	2

For the TCN, the kernel size was set to 2 after initial experimentation. The number of layers (range $\{2, 3\}$), the number of hidden units (range $\{4, 8, 16, 32\}$) and dropout (range $\{0, 0.05, 0.1, 0.2\}$) were optimized using grid search. Similarly to the other experiments dropout yielded no gain and the best network had 2 layers with 8 units per layer. Total time consumption for this optimization and hyperparameter search was 33 hours.

The MLP was implemented as a single hidden layer neural network with ReLU as activation function. The model order (range $\{1, 2, 4, 8, 16, 32, 64\}$) and the number of hidden units (range $\{4, 8, 16, 32, 64, 128, 256\}$) were optimized using grid search and the best hyper parameters were 4 and 32 respectively. Total time consumption for this optimization and hyperparameter search was 7 hours.

In the LSTM case, the hyperparameters for batch size (range $\{1, 2, 4, 8, 16, 32\}$) and the number hidden units (range $\{4, 16, 36, 64\}$) were optimized using grid search and the best hyper parameters were 8 and 36 respectively. Total time consumption for this optimization and hyperparameter search was 40 hours.

7.3.4.3 F-16 ground vibration test

Again, we used grid search for finding the hyperparameters. In each possible training configuration, we have trained the TCN for all possible combinations of: number of hidden layers in $\{16, 32, 64, 128\}$; dropout rate in $\{0.0, 0.3, 0.5, 0.8\}$; number of residual blocks in $\{1, 2, 4, 8\}$; for the kernel size in $\{2, 4, 8, 16\}$; for the presence or absence of dilations; and, for the use of *batch norm*, *weight norm* or nothing after each convolutional layer. The best result in this case, is to use batch norm, no dilation, kernel size

equals to 8, dropout rate equals to 0.3, and 8 layers. Training the network with this configuration took approximately 40 minutes.

We have trained the MLP for all combination of: number of hidden layers in {16, 32, 64, 128, 256}; model order n in {2, 4, 8, 16, 32, 64, 128}; activation function in {ReLU, sigmoid}. Use sigmoid, model order equals to 64 and 256 hidden units yields the best results. Training the network with this configuration took 4 minutes. Training the network with this configuration took approximately 5 minutes.

Finally, we trained the LSTM for all combinations of: number of hidden layers in {16, 32, 64, 128}; number of stacked LSTM layers in {1, 2}; dropout rate in {0.0, 0.3, 0.5, 0.8}. The best configuration is 2 stacked LSTM layers, dropout rate equals to 0 and hidden size equals to 128. Training the network with this configuration took approximately 50 minutes.

7.4 Conclusion and future work

In this work we applied recent deep learning methods to standard system identification benchmarks. Our initial results indicate that these models have potential to provide good results in system identification problems, even if this requires us to rethink how to train and regularize these models. Indeed, methods which are used in traditional deep learning settings do not always improve the performance. For example, dropout did not yield better results in any of the problems. Neither did the long memory offered by the dilation factor in TCNs offer any improvement, which is most likely due to the fact that these problems have a relatively short and exponentially decaying memory, as most dynamical systems do. Other findings are that TCNs work well also for small datasets and that LSTMs did show a good overall performance despite being very rarely applied to system identification problems.

Causal convolutions are effectively similar to NARX models and share statistical properties with this class of models. Hence, they are also expected to be biased for settings where the noise is not white. This could justify the limitations of TCNs observed in our experiments. Extending TCNs to handle situations where the data is contaminated with non-white noise seems to be a promising direction in improving the performance of these models. Furthermore, both LSTMs and the dilated TCNs are designed to work well for data with long memory dependencies. Therefore it would be interesting to apply these models to system identification problems where such long term memory is actually needed, e.g. switched systems, or to study if the long-term memory can be translated into accurate long-term predictions, which could have interesting applications in a model predictive control setting.

8 Automatic diagnosis of the 12-Lead ECG using a deep neural network

Cardiovascular diseases are the leading cause of death worldwide (Roth et al., 2018) and the electrocardiogram (ECG) is a major tool in their diagnoses. As ECGs transitioned from analog to digital, automated computer analysis of standard 12-lead electrocardiograms gained importance in the process of medical diagnosis (Willems et al., 1987; Schläpfer and Wellens, 2017). However, limited performance of classical algorithms (Willems et al., 1991; Shah and Rubin, Oct) precludes its usage as a standalone diagnostic tool and relegates them to an ancillary role (Estes, 2013; Schläpfer and Wellens, 2017).

There are great expectations when it comes to how Deep Neural Networks (DNNs) may improve health care and clinical practice (Stead, 2018; Naylor C, 2018; Hinton, 2018). So far, the most successful applications used a supervised learning setup to automate diagnosis from exams. Supervised learning models, which learn to map an input to an output based on example input-output pairs, have achieved better performance than a human specialist on their routine work-flow in diagnosing breast cancer (Bejnordi et al., 2017) and detecting retinal diseases from three-dimensional optical coherence tomography scans (De Fauw et al., 2018). While efficient, training DNNs in this setup introduces the need for large quantities of labeled data which, for medical applications, introduce several challenges, including those related to confidentiality and security of personal health information (Beck et al., 2016).

A convincing preliminary study of the use of DNNs in ECG analysis was recently presented in (Hannun et al., 2019). For single-lead ECGs, DNNs could match state-of-the-art algorithms when trained in openly available datasets (e.g. 2017 PhysioNet Challenge data (Clifford et al., 2017)) and, for a large enough training dataset, present superior performance when compared to practicing cardiologists. However, as pointed out by the authors, it is still an open question if the application of this technology would be useful in a realistic clinical setting, where 12-lead ECGs are the standard technique (Hannun et al., 2019).

The short-duration, standard, 12-lead ECG (S12L-ECG) is the most commonly used complementary exam for the evaluation of the heart, being employed across all clinical settings, from the primary care centers to the intensive care units. While long-term cardiac monitoring, such as in the Holter exam, provides information mostly about cardiac rhythm and repolarization, the S12L-ECG can provide a full evaluation of the cardiac electrical activity. This includes arrhythmias, conduction disturbances, acute coronary syndromes, cardiac chamber hypertrophy and enlargement and even the effects of drugs and electrolyte disturbances. Thus, a deep learning approach that allows for accurate interpretation of S12L-ECGs would have the greatest impact.

S12L-ECGs are often performed in settings, such as in primary care centers and emergency units, where there are no specialists to analyze and interpret the ECG tracings. Primary care and emergency department health professionals have limited diagnostic abilities in interpreting S12-ECGs (Mant et al., 2007; Veronese et al., 2016). The need for an accurate automatic interpretation is most acute in low and middle-income countries, which are responsible for more than 75% of deaths related to cardiovascular disease (World Health Organization, 2014), and where the population, often, do not have access to cardiologists with full expertise in ECG diagnosis.

The use of DNNs for S12L-ECG is less explored than its usage in the single-lead setup. A contributing factor for this is the shortage of full digital S12L-ECG databases, since most recordings

are still registered only on paper, archived as images, or stored in PDF format (Sassi et al., Dec). Most available databases comprise a few hundreds of tracings and no systematic annotation of the full list of ECG diagnoses (Lyon et al., 2018), limiting their usefulness as training datasets in a supervised learning setting. This lack of systematically annotated data is unfortunate, as training an accurate automatic method of diagnosis from S12L-ECG would be greatly beneficial.

In this chapter, we demonstrate the effectiveness of DNNs for automatic S12L-ECG classification. We build a large-scale dataset of labelled S12L-ECG exams for clinical and prognostic studies (the CODE - Clinical Outcomes in Digital Electrocardiology study) and use it to develop a DNN to classify 6 types of ECG abnormalities considered representative of both rhythmic and morphologic ECG abnormalities.

The content here is largely influenced by the content from the following publication:

Automatic Diagnosis of the 12-Lead ECG using a Deep Neural Network, Antônio H. Ribeiro, Manoel Horta Ribeiro, Gabriela M.M. Paixão, Derick M. Oliveira, Paulo R. Gomes, Jéssica A. Canazart, Milton P. S. Ferreira, Carl R. Andersson, Peter W. Macfarlane, Wagner Meira Jr., Thomas B. Schön, Antonio Luiz P. Ribeiro. *Nature Communications*, 2020. v. 11(1) n. 1760. doi: 10.1038/s41467-020-15432-4

This work was also presented at the SciLifeLab Science Summit in Uppsala, Sweden (2019), where it received the best poster award. And at the Swedish Symposium on Deep Learning - SSDL (2019), also as a poster. It was preceded by the pre-print:

Automatic Diagnosis of Short-Duration 12-Lead ECG using a Deep Convolutional Network Antônio H. Ribeiro, Manoel Horta Ribeiro, Gabriela Paixão, Derick Oliveira, Paulo R. Gomes, Jéssica A. Canazart, Milton Pifano, Wagner Meira Jr., Thomas B. Schön, Antonio Luiz Ribeiro. 2018. <https://arxiv.org/abs/1811.12194>

This pre-print was presented as a poster in the peer-reviewed non-archival workshop “*Machine Learning for Health (ML4H)*” at NeurIPS 2018, where it received a travel award. Earlier versions were also presented at the Brazilian Congress of Cardiology in 2018 and in the Brazilian Congress of Informatics in Health in 2018.

The outline of the chapter is the following: we present our findings in Section 8.1, which are then discussed in Section 8.2. The architecture choice, data collection and statistical analysis are detailed in Section 8.3.

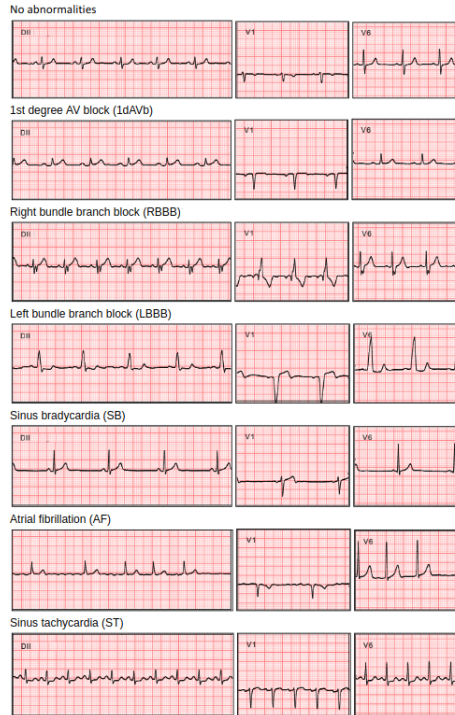
8.1 Results

8.1.1 Model specification and training

We collected a dataset consisting of 2,322,513 ECG records from 1,676,384 different patients of 811 counties in the state of Minas Gerais/Brazil from the Telehealth Network of Minas Gerais (TNMG) (Alkmim et al., 2012). The dataset characteristics are summarized in Table 8.1. The acquisition and annotation procedures of this dataset are described in Methods. We split this dataset into a training set and a validation set. The training set contains 98% of the data. The validation set consists of the remaining 2% (~50,000 exams) of the dataset and it was used for hyperparameter tuning.

We train a DNN to detect: 1st degree AV block (1dAVb), right bundle branch block (RBBB), left bundle branch block (LBBB), sinus bradycardia (SB), atrial fibrillation (AF) and sinus tachycardia (ST). These 6 abnormalities are displayed in Figure 8.1.

Abnormality	Train+Val (n = 2,322,513)	Test (n = 827)
1dAVb	35,759 (1.5 %)	28 (3.4 %)
RBBB	63,528 (2.7 %)	34 (4.1 %)
LBBB	39,842 (1.7 %)	30 (3.6 %)
SB	37,949 (1.6 %)	16 (1.9 %)
AF	41,862 (1.8 %)	13 (1.6 %)
ST	49,872 (2.1 %)	36 (4.4 %)
Age group		
16-25	155,531 (6.7 %)	43 (5.2 %)
26-40	406,239 (17.5 %)	122 (14.8 %)
41-60	901,456 (38.8 %)	340 (41.1 %)
61-80	729,300 (31.4 %)	278 (33.6 %)
≥81	129,987 (5.6 %)	44 (5.3 %)
Sex		
Male	922,780 (39.7 %)	321 (38.8 %)
Female	1,399,733 (60.3 %)	506 (61.2 %)

Table 8.1 – (**Dataset summary**) Patient characteristics and abnormalities prevalence, n (%).Figure 8.1 – (**Abnormalities examples**) A list of all the abnormalities the model classifies. We show only 3 representative leads (DII, V1 and V6).

We used a DNN architecture known as the residual network (He et al., 2016a), commonly used for images, which we here have adapted to unidimensional signals. A similar architecture has been successfully employed for detecting abnormalities in single-lead ECG signals (Hannun et al., 2019). Furthermore, in the 2017 Physionet challenge (Clifford et al., 2017), algorithms for detecting AF have been compared in an open dataset of single lead ECGs and, both the architecture described in (Hannun et al., 2019) and other convolutional architectures (Hong et al., 2017; Kamaleswaran et al., 2018) have achieved top scores.

The DNN parameters were learned using the training dataset and our design choices were made in order to maximize the performance on the validation dataset. We should highlight that, despite using a significantly larger training dataset, we got the best validation results with an architecture with, roughly, one quarter the number of layers and parameters of the network employed in (Hannun et al., 2019).

	Precision (PPV)				Recall (Sensitivity)				Specificity				F1 Score			
	DNN	cardio.	emerg.	stud.	DNN	cardio.	emerg.	stud.	DNN	cardio.	emerg.	stud.	DNN	cardio.	emerg.	stud.
1dAVb	0.867	0.905	0.639	0.605	0.929	0.679	0.821	0.929	0.995	0.997	0.984	0.979	0.897	0.776	0.719	0.732
RBBB	0.895	0.868	0.963	0.914	1.000	0.971	0.765	0.941	0.995	0.994	0.999	0.996	0.944	0.917	0.852	0.928
LBBB	1.000	1.000	0.963	0.931	1.000	0.900	0.867	0.900	1.000	1.000	0.999	0.997	1.000	0.947	0.912	0.915
SB	0.833	0.833	0.824	0.750	0.938	0.938	0.875	0.750	0.996	0.996	0.996	0.995	0.882	0.882	0.848	0.750
AF	1.000	0.769	0.800	0.571	0.769	0.769	0.615	0.923	1.000	0.996	0.998	0.989	0.870	0.769	0.696	0.706
ST	0.947	0.968	0.946	0.912	0.973	0.811	0.946	0.838	0.997	0.999	0.997	0.996	0.960	0.882	0.946	0.873

Table 8.2 – **(Performance indexes)** Scores of our DNN are compared on the test set with the average performance of: i) 4th year cardiology resident (*cardio.*); ii) 3rd year emergency resident (*emerg.*); and, iii) 5th year medical students (*stud.*). (PPV = positive predictive value)

8.1.2 Testing and performance evaluation

For *testing* the model we employed a dataset consisting of 827 tracings from distinct patients annotated by 3 different cardiologists with experience in electrocardiography (see Methods). The test dataset characteristics are summarized in Table 8.1. Table 8.2 shows the performance of the DNN on the test set. High-performance measures were obtained for all ECG abnormalities, with F1 scores above 80% and specificity indexes over 99%. We consider our model to have predicted the abnormality when its output — a number between 0 and 1 — is above a threshold. Figure 8.2 shows the precision-recall curve for our model, for different values of this threshold.

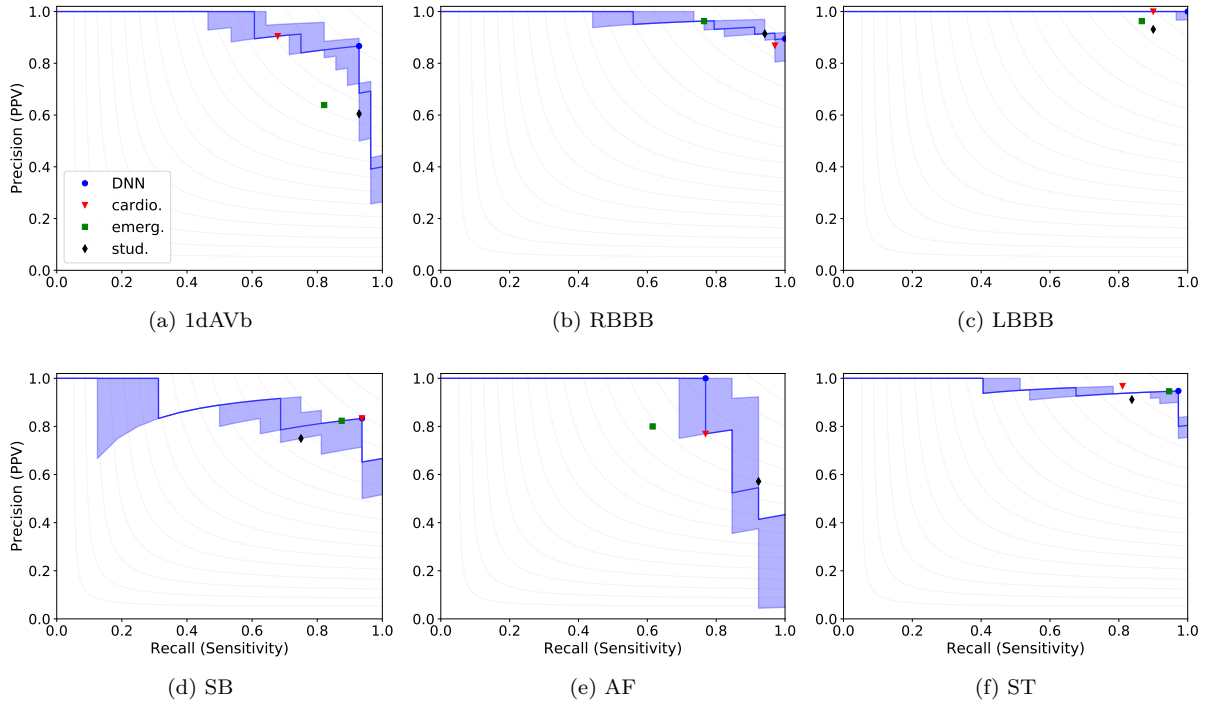


Figure 8.2 – **(Precision-recall curve)** Show precision-recall curve for our nominal prediction model on the test set (strong line) with regard to each ECG abnormalities. The shaded region show the range between maximum and minimum precision for neural networks trained with the same configuration and different initialization. Points corresponding the performance of resident medical doctors and students are also displayed, together with the point corresponding to the DNN performance for the same threshold used for generating Table 8.2. Gray dashed curves in the background correspond to iso- F_1 curves (i.e. curves in the precision-recall plane with constant F_1 score).

Neural networks are initialized randomly, and different initialization usually yield different results. In order to show the stability of the method, we have trained 10 neural networks with the same set of

	DNN			cardio.				emerg.				stud.			
	meas.	noise	unexplain.	meas.	noise	concep.	atte.	meas.	noise	concep.	atte.	meas.	noise	concep.	atte.
1dAVb	3	2	1	8	3			15	3			13	3	3	
RBBB	3		1	4		2		1		8		3		2	
LBBB				1	1	1			1	4			2	3	
SB	4			4				4			1	5		2	1
AF		2	1		4	2			2	5			3	7	
ST	2		1	2	1		5	1	1	1	1	1	2	1	5

Table 8.3 – **(Error analysis)** Present the analysis of misclassified exams. The errors were classified into the following categories: i) measurements errors (*meas.*) were ECG interval measurements preclude the given diagnosis by its textbook definition ; ii) errors due to *noise*, were we believe that the analyst or the DNN failed due to a lower than usual signal quality; and, iii) other type of errors (*unexplain.*). Those were further divided, for the medical residents and students, into two categories: conceptual errors (*concep.*), where our reviewer suggested that the doctor failed to understand the definitions of each abnormality, and attention errors (*atte.*), where we believe the error could be avoided if the reviewer had been more careful.

hyperparameters and different initializations. The range between the maximum and minimum precision among these realizations, for different values of threshold, are the shaded regions displayed in Figure 8.2. These realizations have micro average precision (mAP) between 0.946 and 0.961, we choose the one with mAP immediately above the median value of all executions (the one with $\text{mAP} = 0.951$)¹. All the analysis from now on will be for this realization of the neural network, which correspond both to the strong line in Figure 8.2 and to the scores presented in Table 8.2. For this model, Figure 8.2 shows the point corresponding to the maximum F1 score for each abnormality. The threshold corresponding to this point is used for producing the DNN scores displayed in Table 8.2.

The same dataset was evaluated by: i) two 4th year cardiology residents; ii) two 3rd year emergency residents; and, iii) two 5th year medical students. Each one annotated half of the exams in the test set. Their average performances are given, together with the DNN results, in the Table 8.2 and their precision-recall scores are plotted on Figure 8.2. Considering the F1 score, the DNN matches or outperforms the medical residents and students for all abnormalities. The confusion matrices and the inter-rater agreement (kappa coefficients) for the DNN, the resident medical doctors and students are provided, respectively, in Tables 8.4 and 8.5(a). Additionally, in Table 8.5(b) we compare the inter-rater agreement between the neural network and the certified cardiologists that annotated the test set.

A trained cardiologist reviewed all the mistakes made by the DNN, the medical residents and the students, trying to explain the source of the error. The cardiologist had meetings with the residents and students where they together agreed on which was the source of the error. The results of this analysis are given in Table 8.3.

In order to compare of the performance difference between the DNN and resident medical doctors and students, we compute empirical distributions for the precision (PPV), recall (sensitivity), specificity and F1 score using bootstrapping (Efron and Tibshirani, 1994). The boxplots corresponding to these bootstrapped distributions are presented in Figure 8.3. We have also applied the McNemar test (McNemar, 1947) to compare the misclassification distribution of the DNN, the medical residents and the students. Table 8.6 show the p -values of the statistical test. Both analyses do not indicate a statistically significant difference in performance among the DNN and the medical residents and students for most of the classes.

¹ We couldn't choose the model with mAP equal to the median value because 10 is even number, hence there is no single middle value.

		predicted label							
		DNN		cardio.		emerg.		stud.	
	true label	not present	present	not present	present	not present	present	not present	present
1dAVb	not present	795	4	797	2	786	13	782	17
	present	2	26	9	19	5	23	2	26
RBBB	not present	789	4	788	5	792	1	790	3
	present	0	34	1	33	8	26	2	32
LBBB	not present	797	0	797	0	796	1	795	2
	present	0	30	3	27	4	26	3	27
SB	not present	808	3	808	3	808	3	807	4
	present	1	15	1	15	2	14	4	12
AF	not present	814	0	811	3	812	2	805	9
	present	3	10	3	10	5	8	1	12
ST	not present	788	2	789	1	788	2	787	3
	present	1	36	7	30	2	35	6	31

Table 8.4 – (**Confusion matrices**) Show the absolute number of: i) false positives; ii) false negatives; iii) true positives; and, iv) true negatives, for each abnormality on the test set.

	1dAVb	RBBB	LBBB	SB	AF	ST
DNN vs cardio.	0.656	0.917	0.945	0.830	0.780	0.864
DNN vs emerg.	0.684	0.792	0.909	0.796	0.595	0.930
DNN vs stud.	0.642	0.928	0.912	0.760	0.574	0.855
cardio. vs emerg.	0.656	0.824	0.923	0.912	0.515	0.847
cardio. vs stud.	0.612	0.871	0.889	0.880	0.700	0.792
emerg. vs stud.	0.615	0.799	0.852	0.907	0.508	0.897

(a)

	1dAVb	RBBB	LBBB	SB	AF	ST
DNN vs Cert. cardiol. 1	0.758	0.928	0.964	0.770	0.696	0.847
DNN vs Certif. cardiol. 2	0.852	0.942	1.000	0.770	0.746	0.884
Cert. cardiol. 1 vs Certif. cardiol. 2	0.741	0.955	0.964	0.844	0.831	0.902

(b)

Table 8.5 – (**Kappa coefficients**) Show the Kappa scores measuring the inter-rater agreement on the test set. In (a), we compare the DNN, the medical residents and the students two at a time. In (b), we compare the DNN, and the certified cardiologists that annotated the test set (**certif. cardiol.**). If the raters are in complete agreement then it is equal to 1. If there is no agreement among the raters other than what would be expected by chance it is equal to 0.

8.2 Discussion

This work demonstrates the effectiveness of "end-to-end" automatic S12L-ECG classification. This presents a paradigm shift from the classical ECG automatic analysis methods (Jambukia et al., 2015). These classical methods, such as the University of Glasgow ECG analysis program (Macfarlane et al., 2005), first extract the main features of the ECG signal using traditional signal processing techniques and then use these features as inputs to a classifier. End-to-end learning presents an alternative to these two-step approaches, where the raw signal itself is used as an input to the classifier which learns, by itself, to extract the features. This approach have presented, in a emergency room setting, performance superior to commercial ECG software based on traditional signal processing techniques (Smith et al., 2019)

Neural networks have previously been used for classification of ECGs both in a classical — feature-based — setup (CUBANSKI et al., 1994; Tripathy et al., 2019) and in an end-to-end learn

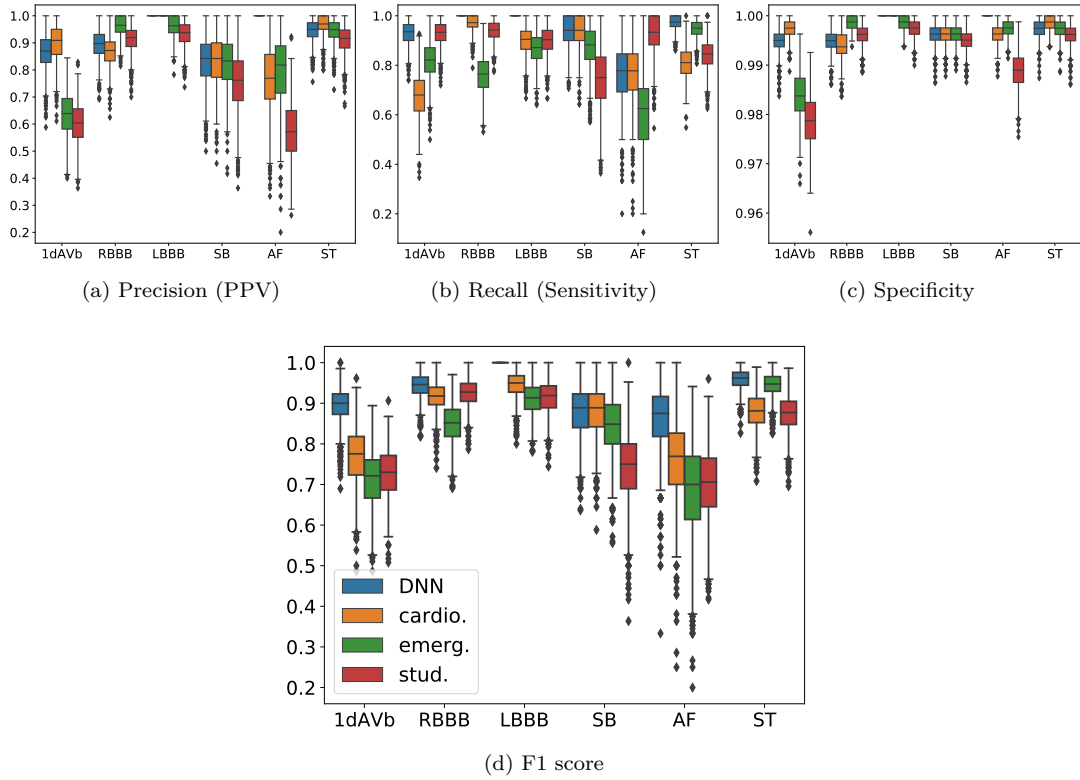


Figure 8.3 – **(Bootstrapped scores)** Display boxplots of empirical distribution of precision, recall, specificity and F1 score on the test set. Sampling with replacement (i.e. bootstrapping) from the test set was used to generate $n = 1000$ samples. The results are given for the DNN, the medical residents and students. Source data are provided as a Source Data file. The boxplots should be read as follows: the central line correspond to the median value of the empirical distribution, the box region correspond to the range of values between the first and third quartile (also known as interquartile range or IQR), the whiskers extend from 1.5 IQR below and above the first and third quartiles, values outside of that range are considered outliers and show as diamonds.

	1dAVb	RBBB	LBBB	SB	AF	ST
DNN vs cardio.	0.225	0.414	0.083	1.000	0.180	0.096
DNN vs emerg.	0.007	0.166	0.025	0.705	0.157	0.655
DNN vs stud.	0.009	0.655	0.025	0.157	0.052	0.058
cardio. vs emerg.	0.108	0.366	0.317	0.564	0.763	0.206
cardio. vs stud.	0.102	0.739	0.414	0.046	0.206	0.782
emerg. vs stud.	0.853	0.248	1.000	0.083	0.439	0.059

Table 8.6 – **(McNemar test)** Display the p -values for the McNemar test comparing the misclassification on the test set. The DNN, the medical residents and the students were compared two at a time. Entries with statistical significance (with 0.05 significance level) are displayed in **boldface**.

setup (Rubin et al., 2017; Acharya et al., 2017; Hannun et al., 2019). Hybrid methods combining the two paradigms are also available: the classification may be done using a combination of handcrafted and learned features (Shashikumar et al., 2018) or by using a two-stage training, obtaining one neural network to learn the features and another to classify the exam according to these learned features (Rahhal et al., 2016).

The paradigm shift towards end-to-end learning had a significant impact on the size of the datasets used for training the models. Many results using classical methods (Jambukia et al., 2015; Acharya et al.,

2017; Rahhal et al., 2016) train their models on datasets with few examples, such as the MIT-BIH arrhythmia database (Goldberger et al., 2000), with only 47 unique patients. The most convincing papers using end-to-end deep learning or mixed approaches, on the other hand, have constructed large datasets, ranging from 3,000 to 100,000 unique patients, for training their models (Shashikumar et al., 2018; Clifford et al., 2017; Hannun et al., 2019; Smith et al., 2019).

Large datasets from previous work (Shashikumar et al., 2018; Clifford et al., 2017; Hannun et al., 2019), however either were obtained from cardiac monitors and Holter exams, where patients are usually monitored for several hours and the recordings are restricted to one or two leads. Or, consist of 12-lead ECGs obtained in a emergency room setting (Goto et al., 2019; Smith et al., 2019). Our dataset with well over 2 million entries, on the other hand, consists of short duration (7 to 10 seconds) S12L-ECG tracings obtained from in-clinic exams and is orders of magnitude larger than those used in previous studies. It encompasses not only rhythm disorders, like AF, SB and ST, as in previous studies (Hannun et al., 2019), but also conduction disturbances, such as 1dAVb, RBBB and LBBB. Instead of beat to beat classification, as in the MIT-BIH arrhythmia database, our dataset provides annotation for S12L-ECG exams, which are the most common in clinical practice.

The availability of such a large database of S12L-ECG tracings, with annotation for the whole spectrum of ECG abnormalities, opens up the possibility of extending initial results of end-to-end DNN in ECG automatic analysis (Hannun et al., 2019) to a system with applicability in a wide range of clinical settings. The development of such technologies may yield high-accuracy automatic ECG classification systems that could save clinicians considerable time and prevent wrong diagnoses. Millions of S12L-ECGs are performed every year, many times in places where there is a shortage of qualified medical doctors to interpret them. An accurate classification system could help to detect wrong diagnoses and improve the access of patients from deprived and remote locations to this essential diagnostic tool of cardiovascular diseases.

The error analysis shows that most of the DNN mistakes were related to measurements of ECG intervals. Most of those were borderline cases, where the diagnosis relies on a consensus definitions (Rautaharju et al., 2009) that can only be ascertained when a measurement is above a sharp cutoff point. The mistakes can be explained by the DNN failing to encode these very sharp thresholds. For example, the DNN wrongly detecting a SB with a heart rate slightly above 50 bpm or a ST with a heart rate slightly below 100 bpm. Figure 8.4 illustrate this effect. Noise and interference in the baseline are established causes of error (Luo and Johnston, 2010) and affected both automatic and manual diagnosis of ECG abnormalities. Nevertheless, the DNN seems to be more robust to noise and it made fewer mistakes of this type compared to the medical residents and students. Conceptual errors (where our reviewer suggested that the doctor failed to understand the definitions of each abnormality) were more frequent for emergency residents and medical students than for cardiology residents. Attention errors (where we believe that the error could have been avoided if the manual reviewer were more careful) were present at a similar ratio for cardiology residents, emergency residents and medical students.

Interestingly, the performance of the emergency residents is worse than medical students for many abnormalities. This might seem counter-intuitive because they do have less years of medical training. It might, however, be justified by the fact that emergency residents, unlike cardiology residents, do not have to interpret these exams on a daily basis, while medical students still have these concepts fresh from their studies.

Our work is perhaps best understood in the context of its limitations. While we obtained the highest F1 scores for the DNN, the McNemar statistical test and bootstrapping suggest that we do not have confidence enough to assert that the DNN is actually better than the medical residents and students

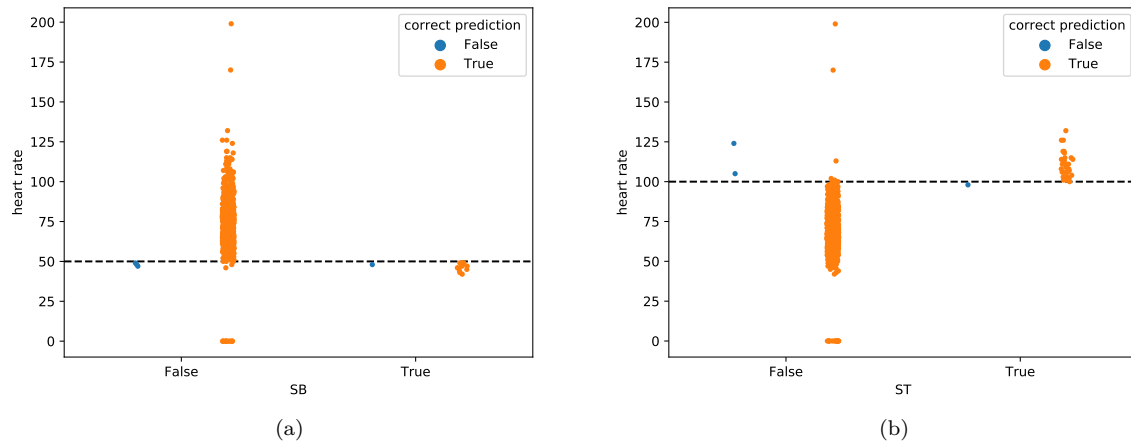


Figure 8.4 – **(Heart rate vs DNN predictions)** Heart rate measured by the Uni-G software for samples in the test set is given on the y -axis. The color indicates if the *DNN* make the correct prediction or not. The x-axis separates the dataset accordingly to the presence of: SB in (a); and, ST in (b). A horizontal line show the threshold of 50 bpm for SB in (a); and, of 100 bpm for ST in (b), which delimit the consensus definition of SB and ST. Notice that most exams for which the neural network fails to get the correct result are very close to this threshold line and are the borderline cases we mentioned in the discussion. It should be highlighted that this automatic measurement system is not perfect, and measurements that may indicate some of the conditions do not necessarily agree with our board of cardiologist (e.g. there are exams with heart rate above 100 according to Uni-G that are not classified by our cardiologist as ST).

with statistical significance. We attribute this lack of confidence in the comparison to the presence of relatively infrequent classes, where a few erroneous classifications may significantly affect the scores. Furthermore, we did not test the accuracy of the DNN in the diagnosis of other classes of abnormalities, like those related to acute coronary syndromes or cardiac chamber enlargements and we cannot extend our results to these untested clinical situations. Indeed, the real clinical setting is more complex than the experimental situation tested in this study and, in complex and borderline situations, ECG interpretation can be extremely difficult and may demand the input of highly specialized personnel. Thus, even if a DNN is able to recognize typical ECG abnormalities, further analysis by an experienced specialist will continue to be necessary to these complex exams.

This proof-of-concept study, showing that a DNN can accurately recognize ECG rhythm and morphological abnormalities in clinical S12L-ECG exams, opens a series of perspectives for future research and clinical applications. A next step would be to prove that a DNN can effectively diagnose multiple and complex ECG abnormalities, including myocardial infarction, cardiac chamber enlargement and hypertrophy and less common forms of arrhythmia, and to recognize a normal ECG. Subsequently, the algorithm should be tested in a controlled real-life situation, showing that accurate diagnosis could be achieved in real-time, to be reviewed by clinical specialists with solid experience in ECG diagnosis. This real-time, continuous evaluation of the algorithm, would provide rapid feedback that could be incorporated as further improvements of the DNN, making it even more reliable.

The TNMG, the large telehealth service from which the dataset used was obtained ([Alkmim et al., 2012](#)), is a natural laboratory for these next steps, since it performs more than 2,000 ECGs a day and it is currently expanding its geographical coverage over a large part of a continental country (Brazil). An optimized system for ECG interpretation, where most of the classification decisions are made

automatically would imply that the cardiologists would only be needed for the more complex cases. If such a system is made widely available, it could be of striking utility to improve access to health care in low and middle-income countries, where cardiovascular diseases are the leading cause of death and systems of care for cardiac diseases are lacking or not working well (Nascimento et al., 2019).

In conclusion, we developed an end-to-end DNN capable of accurately recognizing six ECG abnormalities in S12L-ECG exams, with a diagnostic performance at least as good as medical residents and students. This study shows the potential of this technology, which, when fully developed, might lead to more reliable automatic diagnosis and improved clinical practice. Although expert review of complex and borderline cases seems to be necessary even in this future scenario, the development of such automatic interpretation by a DNN algorithm may expand the access of the population to this basic and useful diagnostic exam.

8.3 Methods

8.3.1 Dataset acquisition

All S12L-ECGs analyzed in this study were obtained by the Telehealth Network of Minas Gerais (TNMG), a public telehealth system assisting 811 out of the 853 municipalities in the state of Minas Gerais, Brazil (Alkmim et al., 2012). Since September 2017, the TNMG has also provided teleradiologic services to other Brazilian states in the Amazonian and Northeast regions. The S12L-ECG exam was performed mostly in primary care facilities using a tele-electrocardiograph manufactured by Tecnologia Eletrônica Brasileira (São Paulo, Brazil) – model TEB ECGPC - or Micromed Biotecnologia (Brasília, Brazil) - model ErgoPC 13. The duration of the ECG recordings is between 7 and 10 seconds sampled at frequencies ranging from 300 to 600 Hz. A specific software developed in-house was used to capture ECG tracings, to upload the exam together with the patient’s clinical history and to send it electronically to the TNMG analysis center. Once there, one cardiologist from the TNMG experienced team analyzes the exam and a report is made available to the health service that requested the exam through an online platform.

We have incorporated the University of Glasgow (Uni-G) ECG analysis program (release 28.5, issued in January 2014) in the in-house software since December 2017. The analysis program was used to automatically identify waves and to calculate axes, durations, amplitudes and intervals, to perform rhythm analysis and to give diagnostic interpretation (Macfarlane et al., 1990, 2005). The Uni-G analysis program also provides Minnesota codes (Macfarlane and Latif, 1996), a standard ECG classification used in epidemiological studies (Prineas et al., 2009). Since April 2018 the automatic measurements are being shown to the cardiologists that give the medical report. All clinical information, digital ECGs tracings and the cardiologist report were stored in a database. All previously stored data was also analyzed by Uni-G software in order to have measurements and automatic diagnosis for all exams available in the database, since the first recordings. The CODE study was established to standardize and consolidate this database for clinical and epidemiological studies. In the present study, the data (for patients above 16 years old) obtained between 2010 and 2016 was used in the training and validation set and, from April to September 2018, in the test set.

8.3.2 Labelling training data from text report

For the training and validation sets, the cardiologist report is available only as a textual description of the abnormalities in the exam. We extract the label from this textual report using a three-step procedure. First, the text is pre-processed by removing stop-words and generating n-grams from the medical report. Then, the *Lazy Associative Classifier* (LAC) (Velooso et al., 2006), trained on a 2800-sample dictionary

created from real diagnoses text reports, is applied to the n-grams. Finally, the text label is obtained using the LAC result in a rule-based classifier for class disambiguation. The classification model reported above was tested on 4557 medical reports manually labeled by a certified cardiologist who was presented with the free-text and was required to choose among the pre-specified classes. The classification step recovered the true medical label with good results, the macro F1 score achieved were: 0.729 for 1dAVb; 0.849 for RBBB; 0.838 for LBBB; 0.991 for SB; 0.993 for AF; 0.974 for ST.

8.3.3 Training and validation set annotation

To annotate the training and validation datasets, we used: i) the Uni-G statements and Minnesota codes obtained by the Uni-G automatic analysis (*automatic diagnosis*); ii) automatic measurements provided by the Uni-G software; and, iii) the text labels extracted from the expert text reports using the semi-supervised methodology (*medical diagnosis*). Both the automatic and medical diagnosis are subject to errors: automatic classification has limited accuracy (Willems et al., 1991; Shah and Rubin, Oct; Schl pfer and Wellens, 2017; Estes, 2013) and text labels are subject both to errors of the practicing expert cardiologists and the labeling methodology. Hence, we combine the expert annotation with the automatic analysis to improve the quality of the dataset. The following procedure is used for obtaining the ground truth annotation:

1. We:

- a) *Accept a diagnosis* (consider an abnormality to be present) if both the expert *and* either the Uni-G statement or the Minnesota code provided by the automatic analysis indicated the same abnormality.
- b) *Reject a diagnosis* (consider an abnormality to be absent) if only one automatic classifier indicates the abnormality in disagreement with both the doctor and the other automatic classifier.

After this initial step, there are two scenarios where we still need to accept or reject diagnoses. They are: i) both classifiers indicate the abnormality, but the expert does not; or ii) only the expert indicates the abnormality, whereas none of the classifiers indicates anything.

2. We used the following rules *to reject some of the remaining diagnoses*:

- a) Diagnoses of ST where the heart rate was below 100 (8376 medical diagnoses and 2 automatic diagnoses) were *rejected*.
- b) Diagnoses of SB where the heart rate was above 50 (7361 medical diagnoses and 16427 automatic diagnosis) were *rejected*.
- c) Diagnoses of LBBB or RBBB where the duration of the QRS interval was below 115 ms (9313 medical diagnoses for RBBB and 8260 for LBBB) were *rejected*.
- d) Diagnoses of 1dAVb where the duration of the PR interval was below 190 ms (3987 automatic diagnoses) were *rejected*.

3. Then, using the sensitivity analysis of 100 manually reviewed exams per abnormality, we came up with the following rules *to accept some of the remaining diagnoses*:

- a) For RBBB, d1AVb, SB and ST, we *accepted* all medical diagnoses. 26033, 13645, 12200 and 14604 diagnoses were *accepted* in this fashion, respectively.

- b) For AF, we required not only that the exam was classified by the doctors as true, but also that the standard deviation of the NN intervals was higher than 646. 14604 diagnoses were *accepted* using this rule.

According to the sensitivity analysis, the number of false positives that would be introduced by this procedure was smaller than 3% of the total number of exams.

4. After this process, we were still left with 34512 exams where the corresponding diagnoses could neither be accepted nor rejected. These were *manually reviewed* by medical students using the Telehealth ECG diagnostic system, under the supervision of a certified cardiologist with experience in ECG interpretation. The process of manually reviewing these ECGs took several months.

It should be stressed that information from previous medical reports and automatic measurements were used only for obtaining the ground truth for training and validation sets and not on later stages of the DNN training.

8.3.4 Test set annotation

The dataset used for testing the DNN was also obtained from TNMG's ECG system. It was independently annotated by two certified cardiologists with experience in electrocardiography. The kappa coefficients (Cohen, 1960) indicate the inter-rater agreement for the two cardiologist and are: 0.741 for 1dAVb; 0.955 for RBBB; 0.964 for LBBB; 0.844 for SB; 0.831 for AF; 0.902 for ST. When they agreed, the common diagnosis was considered as ground truth. In cases where there was *any* disagreement, a third senior specialist, aware of the annotations from the other two, decided the diagnosis. The American Heart Association standardization (Kligfield et al., 2007) was used as the guideline for the classification.

It should be highlighted that the annotation was performed in an upgraded version of the TNMG software, in which the automatic measurements obtained by the Uni-G program are presented to the specialist, that has to choose the ECG diagnosis among a number of pre-specified classes of abnormalities. Thus, the diagnosis was codified directly into our classes and there was no need to extract the label from a textual report, as it was done for the training and validation sets.

8.3.5 Neural network architecture and training

We used a convolutional neural network similar to the residual network (He et al., 2016a), but adapted to unidimensional signals. This architecture allows deep neural networks to be efficiently trained by including skip connections. We have adopted the modification in the residual block proposed in (He et al., 2016b), which place the skip connection in the position displayed in Figure 8.5.

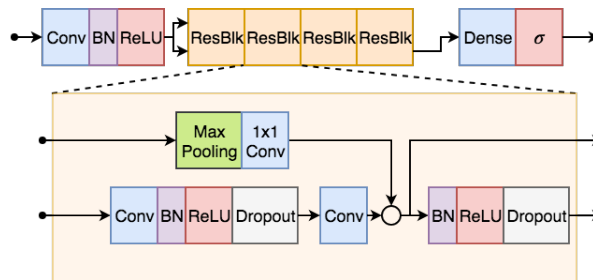


Figure 8.5 – (DNN architecture) The uni-dimensional residual neural network architecture used for ECG classification.

All ECG recordings are re-sampled to a 400 Hz sampling rate. The ECG recordings, which have between 7 and 10 seconds, are zero-padded resulting in a signal with 4096 samples for each lead. This signal is the input for the neural network.

The network consists of a convolutional layer (**Conv**) followed by 4 residual blocks with two convolutional layers per block. The output of the last block is fed into a fully connected layer (**Dense**) with a sigmoid activation function, σ , which was used because the classes are not mutually exclusive (i.e. two or more classes may occur in the same exam). The output of each convolutional layer is rescaled using batch normalization, (**BN**), (Ioffe and Szegedy, 2015) and fed into a rectified linear activation unit (**ReLU**). **Dropout** (Srivastava et al., 2014) is applied after the nonlinearity.

The convolutional layers have filter length 16, starting with 4096 samples and 64 filters for the first layer and residual block and increasing the number of filters by 64 every second residual block and subsampling by a factor of 4 every residual block. **Max Pooling** (Hutchison et al., 2010) and convolutional layers with filter length 1 (**1x1 Conv**) are included in the skip connections to make the dimensions match those from the signals in the main branch.

The average cross-entropy is minimized using the Adam optimizer (Kingma and Ba, 2014) with default parameters and learning rate $lr = 0.001$. The learning rate is reduced by a factor of 10 whenever the validation loss does not present any improvement for 7 consecutive epochs. The neural network weights was initialized as in (He et al., 2015) and the bias were initialized with zeros. The training runs for 50 epochs with the final model being the one with the best validation results during the optimization process.

8.3.6 Hyperparameter tuning

This final architecture and configuration of hyperparameters was obtained after approximately 30 iterations of the procedure: i) find the neural network weights in the training set; ii) check the performance in the validation set; and, iii) manually chose new hyperparameters and architecture using insight from previous iterations. We started this procedure from the set of hyperparameters and architecture used in (Hannun et al., 2019). It is also important to highlight that the choice of architecture and hyperparameters was done together with improvements in the dataset. Expert knowledge was used to take decision about how to incorporate, on the manual tuning procedure, information about previous iteration that were evaluated on slightly different versions of the dataset.

The hyperparameters were choosen among the following options: residual neural networks with $\{2, 4, 8, 16\}$ residual blocks, kernel size $\{8, 16, 32\}$, batch size $\{16, 32, 64\}$, initial learning rate $\{0.01, 0.001, 0.0001\}$, optimization algorithms $\{SGD, ADAM\}$, activation functions $\{ReLU, ELU\}$, dropout rate $\{0, 0.5, 0.8\}$, number of epochs without improvement in plateaus between 5 and 10, that would result in a reduction in the learning rate between 0.1 and 0.5. Besides that, we also tried to: i) use vectorcardiogram linear transformation to reduce the dimensionality of the input; ii) include LSTM layer before convolutional layers; iii) use residual network without the preactivation architecture proposed in (He et al., 2016b); iv) Use the convolutional architecture known as VGG; v) swiching the order of activation and batch normalization layer.

8.3.7 Statistical and empirical analysis of test results

We computed the precision-recall curve to assess the model discrimination of each rhythm class. This curve shows the relationship between precision (PPV) and recall (sensitivity), calculated using binary decision thresholds for each rhythm class. For imbalanced classes, such as our test set, this plot is more informative than the ROC plot (Saito and Rehmsmeier, 2015). For the remaining analyses we fixed the DNN threshold to the value that maximized the F1 score, which is the harmonic mean between precision

and recall. The F1 score was chosen here due to its robustness to class imbalance (Saito and Rehmsmeier, 2015).

For the DNN with a fixed threshold, and for the medical residents and students, we computed the precision, the recall, the specificity, the F1 score and, also, the confusion matrix. This was done for each class. Bootstrapping (Efron and Tibshirani, 1994) was used to analyze the empirical distribution of each of the scores: we generated 1000 different sets by sampling with replacement from the test set, each set with the same number samples as in the test set, and computed the precision, the recall, the specificity and the F1 score for each. The resulting distributions are presented as a boxplot. We used the McNemar test (McNemar, 1947) to compare the misclassification distribution of the DNN and the medical residents and students on the test set and the kappa coefficient (Cohen, 1960) to compare the inter-rater agreement.

All the misclassified exams were reviewed by an experienced cardiologist and, after an interview with the ECG reviewers, the errors were classified into: measurement errors, noise errors and unexplained errors (for the DNN only) and conceptual and attention errors (for medical residents and students only).

Data availability

The test dataset used in this study is openly available, and can be downloaded at <https://doi.org/10.5281/zenodo.3625006>. The weights of all deep neural network models we developed for this work are available at <https://doi.org/10.5281/zenodo.3625017>. Restrictions apply to the availability of the training set. Requests to access the training data will be considered on an individual basis by the Telehealth Network of Minas Gerais.

Code availability

The code for training and evaluating the DNN model, and, also, for generating figures and tables in this work, is available at: <https://github.com/antonior92/automatic-ecg-diagnosis>.

Research ethics statement

This study complies with with all relevant ethical regulations and was approved by the Research Ethics Committee of the Universidade Federal de Minas Gerais, protocol 49368496317.7.0000.5149. Since this is a study with secondary data, informed consent was not required by the Research Ethics Committee.

Conclusion

The enormous success deep learning had in the last decade in solving hard problems generates interest in applying it to the most diverse areas. Exciting new findings might come out of the interplay between deep learning, signal processing, control theory, and system identification. Pitfalls, however, must be avoided so all the fields might benefit the most from the cross-fertilization. On the one hand, it is fundamental not to let the eagerness and excitement with deep neural networks result in naive one-size-fits-all solutions that ignore previous theory and knowledge. On the other hand, the wariness of researches from other communities concerning deep learning might not get in the way of trying to understand it from different perspectives and apply it to the most diverse application.

In this work, we hope not only to have avoided these pitfalls but also provided knowledge for helping others to avoid them as well. Translating deep learning concepts into familiar concepts from signal processing and system identification is an important step toward the dissemination of these ideas into these communities. We tried to do that to some extent, for instance in Section 2.3 and Chapter 7. On the other hand, systems, control, and signal processing theories are very rich. They can be quite useful in trying to better understand ideas that have arisen in deep learning applications. One example is the content of Chapter 6, which explores concepts from system theory to help better understand recurrent neural networks.

Table 1.1 summarize the design choices we made during the implementations of the numerical examples. The tradeoffs involved are extensively discussed in Section 1.3. Chapters 6 and 8 deal with large models, large datasets or both. The choices we take are the default for deep learning applications (Goodfellow et al., 2016) and are appropriate for the type of problems and models this community is interested in. Chapters 4 and 5, on the other hand, are focused on smaller problems and make choices that are common for system identification practitioners. In Chapter 7, we use choices that are default in the deep learning community in system identification benchmarks. We believe this diversity of design choices is one of the strong points of this work and shows that the findings in this thesis translate well into different domains of knowledge.

We dedicate a significant portion of this thesis to analyze the advantages of recurrence in models and also its limitations. Despite of that, we are far from a simple and definitive answer to the questions: “should I use a recurrent or feedforward models? One-step-ahead or free-run simulation minimization? Parallel or series-parallel configurations?” All of which reflect different angles on the same problem. Nevertheless, we provide insight into various aspects that should be taken into consideration.

On the one hand, recurrent models offer a richer representation, which might yield better asymptotic properties in the presence of noise (cf. Section 3.2 and Chapter 4) and more flexibility when learning attractors of a dynamical system (see Chapter 6). On the other hand, feedforward models are easier to parallelize and yield more stable training since its objective function has better smoothness properties, as studied in Chapters 5 and 6. Mixed approaches combining advantages of feedforward and recurrent models are also possible, and the multiple shooting method presented in Chapter 5 is one example of such an approach. Mixed approaches also introduce new challenges; for instance, when using multiple shooting, we have to deal with extra dimensions and constraints in the optimization.

Better techniques for modeling signals, systems, and sequences can yield huge technological gains. Be by better diagnosing diseases from biomedical signals by allowing computers to better understand and generate language; or, by making current engineering technology for aviation, communication, and industry more precise and efficient, unlocking a wide new range of possibilities. We believe nonlinear differentiable models are a powerfull tool, and we hope to have shown some of the strengths of such models and addressed some of the challenges for their usage.

Appendix

APPENDIX A – Sequential quadratic programming solver

This appendix describe the implementation of the solver described in (Lalee et al., 1998). The algorithm was implemented in Python (together with an extension by Byrd et al. (1999) that allows the inclusion of inequality constraints) as part of my Google Summer of Code project and have been integrated to SciPy as `scipy.optimize.minimize(..., method='trust-constr')`.

The method is able to solve large equality-constrained nonlinear programming problems:

$$\begin{aligned} \min_{\boldsymbol{\theta}} V(\boldsymbol{\theta}), \\ \text{subject to } \mathbf{c}(\boldsymbol{\theta}) = 0, \end{aligned} \tag{A.1}$$

where $V: \mathbb{R}^n \rightarrow \mathbb{R}$ and $\mathbf{c}: \mathbb{R}^n \rightarrow \mathbb{R}^m$ are twice continuously differentiable functions.

The algorithm solves a sequence of Taylor approximations in order to gradually converge towards a *local* solution of (A.1). At the k -th iteration, the algorithm builds a local model around the current iterate $\boldsymbol{\theta}_k$, computes the update \mathbf{p}_k by solving the resulting *quadratic programming* problem and then update the solution ($\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \mathbf{p}_k$).

At each iteration the algorithm computes Δ_k , which should reflects the trust the algorithm has on the local approximation, and then imposes that the step size respect $\|\mathbf{p}_k\| < \Delta_k$. Hence, if the current local approximation is not a good one, the methods does not allow a very large step to be taken. Methods that use this strategy are known as *trust-region methods* (Conn et al., 2000).

Quadratic programming subproblem

At the k -th iteration this method would solve the following trust-region quadratic programming (QP) subproblem in order to compute the step update \mathbf{p}_k :

$$\begin{aligned} \min_{\mathbf{p}} \quad & \nabla V(\boldsymbol{\theta}_k)^T \mathbf{p} + \frac{1}{2} \mathbf{p}^T \nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)^T \mathbf{p}, \\ \text{subject to:} \quad & J(\boldsymbol{\theta}_k) \mathbf{p} + \mathbf{c}(\boldsymbol{\theta}_k) = 0; \\ & \|\mathbf{p}\| \leq \Delta_k, \end{aligned} \tag{A.2}$$

where $\nabla V(\boldsymbol{\theta}_k)^T$ is the function gradient, $\nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)^T$ is the Hessian (in relation to the variable $\boldsymbol{\theta}$) of the Lagrangian¹, $\mathbf{c}(\boldsymbol{\theta}_k)$ is the constraint and $J(\boldsymbol{\theta}_k)$ is the Jacobian of the constraint. Besides that, $\boldsymbol{\theta}_k$ is the current iterate and $\boldsymbol{\lambda}_k$ is the current approximation of lagrange multipliers.

This problem may not be feasible since the linear constraints $J(\boldsymbol{\theta}_k) \mathbf{p} + \mathbf{c}(\boldsymbol{\theta}_k) = 0$ may not be compatible with the trust-region constraint $\|\mathbf{p}\| \leq \Delta_k$. Nocedal and Wright book comments on how to appropriately deal with this incompatibility:

To resolve the possible conflict between the linear constraints and the trust-region constraints, it is not appropriate simply to increase the trust-region until the step satisfying the linear constraints intersects the trust region. This approach would defeat the purpose of using the trust region in the first place as a

¹ The Lagrangian is given by: $\mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k) = V(\boldsymbol{\theta}_k) + \boldsymbol{\lambda}_k^T \mathbf{c}(\boldsymbol{\theta}_k)$.

way to define a region within which we trust the model to accurately reflect the behavior of the objective and constraint functions. Analytically, it would harm the convergence properties of the algorithm.

A more appropriate viewpoint is that there is no reason to satisfy the linearized constraints exactly at every step; rather, we should aim to improve the feasibility of these constraints at each step and to satisfy them exactly only if the trust-region constraint permits it. (Nocedal and Wright, 2006)

The implemented algorithm approach solves, rather than (A.2), the subproblem:

$$\begin{aligned} \min_{\mathbf{p}} \quad & \nabla V(\boldsymbol{\theta}_k)^T \mathbf{p} + \frac{1}{2} \mathbf{p}^T \nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k) \mathbf{p}, \\ \text{subject to} \quad & J(\boldsymbol{\theta}_k) \mathbf{p} + \mathbf{c}(\boldsymbol{\theta}_k) = \mathbf{r}_k; \\ & \|\mathbf{p}\| \leq \Delta_k, \end{aligned} \tag{A.3}$$

where \mathbf{r}_k is adjusted such that the constraints are always compatible. This problem is solved in a two steps procedure:

1. First it solves:

$$\begin{aligned} \min_{\mathbf{v}} \quad & \|J(\boldsymbol{\theta}_k) \mathbf{v} + \mathbf{c}(\boldsymbol{\theta}_k)\|^2, \\ \text{subject to} \quad & \|\mathbf{v}\| \leq \eta \Delta_k, \end{aligned} \tag{A.4}$$

$$\tag{A.5}$$

for $0 < \eta < 1$ (In our implementation $\eta = 0.8$ is being used). And then defines \mathbf{r}_k as:

$$\mathbf{r}_k = J(\boldsymbol{\theta}_k) \mathbf{v}_k + \mathbf{c}(\boldsymbol{\theta}_k). \tag{A.6}$$

Where \mathbf{v}_k is the solution of (A.4). Note that for this choice of \mathbf{r}_k the linear constraints $J(\boldsymbol{\theta}_k) \mathbf{p} + \mathbf{c}(\boldsymbol{\theta}_k) = \mathbf{r}_k$ are always compatible with trust-region constraints $\|\mathbf{p}\| \leq \Delta_k$.

2. Next it solves (A.3) for the value of \mathbf{r}_k computed at the last step.

Both substeps are solved in rather economical fashion using efficient methods to get an *inexact* solution to each of the subproblems. The first step is solved using a variation of the *dogleg* procedure (described in (Byrd et al., 1999), p.886) and the second step using the projected conjugate gradient (CG) algorithm (Gould et al., 2001).

Implementation details

There are a few points about this algorithm that deserve some attention. The first of them is that while the solution of the trust-region QP subproblem doesn't give a way of calculating the Lagrange multipliers $\boldsymbol{\lambda}_k$, these Lagrange multipliers are still needed in order to compute $\nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)$.

An approximation of the Lagrange multipliers is obtained by solving the least squares problem:

$$\min_{\boldsymbol{\lambda}} \|\nabla_x \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)\|^2 \Rightarrow \min_{\boldsymbol{\lambda}} \|\nabla V(\boldsymbol{\theta}_k) + J(\boldsymbol{\theta}_k) \boldsymbol{\lambda}\|^2. \tag{A.7}$$

Another important element of the algorithm is the merit function:

$$\phi(\boldsymbol{\theta}; \mu) = V(\boldsymbol{\theta}) + \mu \|\mathbf{c}(\boldsymbol{\theta})\|. \tag{A.8}$$

It combines the constraints and the objective function into a single number that can be used to compare two points and to reject or accept a given step. The penalty parameter μ is usually update through the iterations and it is important for the global convergence of the algorithm it to be monotonically increasing. Some guidelines about how to choose it are provided on (Byrd et al., 1999), p.891.

The trust region radius selection and the step rejection mechanism are both based on the ratio ρ_k . This ratio measures the agreement between the model and the obtained results. It is computed as:

$$\rho_k = \frac{\text{actual reduction}}{\text{predicted reduction}}, \quad (\text{A.9})$$

where:

$$\text{actual reduction} = \phi(\boldsymbol{\theta}_k; \mu) - \phi(\boldsymbol{\theta}_k + p_k; \mu), \quad (\text{A.10})$$

is the reduction on the merit function. And:

$$\text{predicted reduction} = q_\mu(0) - q_\mu(p_k), \quad (\text{A.11})$$

is the reduction of the local model:

$$q_\mu(p) = \nabla V(\boldsymbol{\theta}_k)^T p + \frac{1}{2} p^T \nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k) p + \mu \|c(\boldsymbol{\theta}_k) + J(\boldsymbol{\theta}_k) p\|. \quad (\text{A.12})$$

Algorithm overview

The trust-region sequential quadratic programming (SQP) basic steps for a single iteration are summarized next.

ALGORITHM A.1 (TRUST-REGION SQP). At each iteration, until some stop criteria is met (e.g. $\|\nabla_x \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)\|_\infty < 10^{-8}$), repeat:

1. Compute $V(\boldsymbol{\theta}_k)$, $\nabla V(\boldsymbol{\theta}_k)$, $c(\boldsymbol{\theta}_k)$ and $J(\boldsymbol{\theta}_k)$;
2. Compute least squares Lagrange multipliers $\boldsymbol{\lambda}_k$;
3. Compute $\nabla_{xx}^2 \mathcal{L}(\boldsymbol{\theta}_k, \boldsymbol{\lambda}_k)$;
4. Apply dogleg method in order to compute v_k and r_k (such that the resulting problem is feasible);
5. Compute p_k using the projected CG method;
6. Choose penalty parameter μ_k ;
7. Compute reduction ratio ρ_k ;
8. Accept or reject step p_k depending on reduction ratio ρ_k ;
9. Enlarge or reduce trust-radius depending on reduction ratio ρ_k ;

□

APPENDIX B – Proofs

Theorem 6.2 is a slightly more general version of Theorem 5.1. The consequences of this theorem are extensively studied in Chapters 5 and 6. Its proof is provided next

Notation and setup

We use the same notation used in Section 6.2. Let the Jacobian matrices of $\mathbf{f}(\mathbf{x}, \mathbf{u}; \boldsymbol{\theta})$ with respect to \mathbf{x} and to $\boldsymbol{\theta}$ evaluated at the point $(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta})$ be denoted, respectively, as A_t and B_t . Similarly, the Jacobian matrices of $\mathbf{g}(\mathbf{x}, \mathbf{u}_t; \boldsymbol{\theta})$ are denoted as C_t and F_t . A direct application of the chain rule to (6.2) gives a recursive formula for computing the derivatives of the predicted output in relation to the parameters in the interval $1 \leq t \leq N$:

$$\begin{aligned} D_{t+1} &= A_t D_t + B_t \text{ for } D_0 = \mathbf{0}, \\ J_t &= C_t D_t + F_t, \end{aligned} \tag{B.1}$$

where we denote the Jacobian matrices of $\hat{\mathbf{y}}_t$ and \mathbf{x}_t with respect to $\boldsymbol{\theta}$ respectively as J_t and D_t .

For the cost function V defined as in (6.1), its gradient ∇V is given by:

$$\nabla V = \frac{1}{N} \sum_{t=1}^N J_t l'(\hat{\mathbf{y}}_t, \mathbf{y}_t), \tag{B.2}$$

where $l'(\hat{\mathbf{y}}_t, \mathbf{y}_t)$ denotes the gradient of the loss function regarding its first argument, evaluated at $(\hat{\mathbf{y}}_t, \mathbf{y}_t)$.

Preliminary results

Lemma B.1. *For $i = 1, \dots, n$, let \mathbf{f}_i be a Lipschitz function on Ω with constants L_i . Then,*

1. $\sum_{i=1}^n \mathbf{f}_i$ is also a Lipschitz function on Ω with Lipschitz constant upper bounded by $\sum_{i=1}^n L_i$;
2. if, additionally, \mathbf{f}_i are bounded by M_i on Ω , then $\prod_{i=1}^n \mathbf{f}_i$ is also a Lipschitz function on Ω with Lipschitz constant upper bounded by $(\sum_{i=1}^n M_1 \cdots M_{i-1} L_i M_{i+1} \cdots M_n)$.

Lemma B.2. *Let us define the properties:*

1. $|l(\hat{\mathbf{y}}, \mathbf{y}) - l(\hat{\mathbf{z}}, \mathbf{y})| < (K_1 \|\mathbf{y}\| + K_2 \max(\|\hat{\mathbf{y}}\|, \|\hat{\mathbf{z}}\|)) \|\hat{\mathbf{y}} - \hat{\mathbf{z}}\|;$
2. $l'(\hat{\mathbf{y}}, \mathbf{y}) = \Psi(\hat{\mathbf{y}}) - K_3 \mathbf{y},$

where $l'(\hat{\mathbf{y}}, \mathbf{y})$ denotes the first derivative of the loss function regarding its first argument, evaluated at $(\hat{\mathbf{y}}, \mathbf{y})$. There exist constants K_1 , K_2 , and K_3 and a function Ψ that is Lipschitz continuous with constant K_4 and for which Ψ such that these properties hold for both: a) $l(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$; b) $l(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y}^T \log(\sigma(\hat{\mathbf{y}})) - (1 - \mathbf{y})^T \log(1 - \sigma(\hat{\mathbf{y}}))$. In (b), the sigmoid function, $\sigma(x) = \frac{1}{1 + \exp(-x)}$, and the logarithm are evaluated element-wise. We assume in (b) that the elements in \mathbf{y} are either 0 or 1.

Proof. For (a) and (b), property 2 follows from differentiating $l(\hat{\mathbf{y}}, \mathbf{y})$ with regard to its first argument. For (a), $\phi(\hat{\mathbf{y}}) = 2\hat{\mathbf{y}}$ and $K_3 = 2$; for (b), $\phi(\hat{\mathbf{y}}) = \sigma(\hat{\mathbf{y}})$ and $K_3 = 1$.

For loss function (a), property 1 holds because of the following reasoning:

$$\begin{aligned} \left| \|\hat{\mathbf{y}} - \mathbf{y}\|^2 - \|\hat{\mathbf{z}} - \mathbf{y}\|^2 \right| &= \left| \|\hat{\mathbf{y}}\|^2 - \|\hat{\mathbf{z}}\|^2 - 2\mathbf{y}^T (\hat{\mathbf{y}} - \hat{\mathbf{z}}) \right| \leq \\ &\leq \left| (\|\hat{\mathbf{y}}\| - \|\hat{\mathbf{z}}\|) (\|\hat{\mathbf{y}}\| + \|\hat{\mathbf{z}}\|) - 2\mathbf{y}^T (\hat{\mathbf{y}} - \hat{\mathbf{z}}) \right| \leq \\ &\leq (2\|\mathbf{y}\| + 2\max(\|\hat{\mathbf{y}}\|, \|\hat{\mathbf{z}}\|)) \|\hat{\mathbf{y}} - \hat{\mathbf{z}}\|. \end{aligned}$$

For loss function (b), let \hat{y} and \hat{z} be two scalar values. Furthermore, consider, without loss of generality, that $\hat{y} \geq \hat{z}$. Then:

$$0 \leq \log(\sigma(\hat{y})) - \log(\sigma(\hat{z})) = (\hat{y} - \hat{z}) - \log\left(\frac{\exp(\hat{y}) + 1}{\exp(\hat{z}) + 1}\right) \leq (\hat{y} - \hat{z}). \quad (\text{B.3})$$

The first inequality follows from the fact that $\log(\sigma(\cdot))$ is a monotonically increasing function. The last inequality holds because $\log\left(\frac{\exp(\hat{y})+1}{\exp(\hat{z})+1}\right) \geq 0$. Analogously,

$$\begin{aligned} 0 &\leq \log(1 - \sigma(\hat{z})) - \log(1 - \sigma(\hat{y})) \\ &= (\hat{y} - \hat{z}) - \log\left(\frac{\exp(-\hat{z}) + 1}{\exp(-\hat{y}) + 1}\right) \\ &\leq (\hat{y} - \hat{z}). \end{aligned} \quad (\text{B.4})$$

For $l(\mathbf{y}, \hat{\mathbf{y}})$ defined as in (b), it follows from (B.3), (B.4), and the fact that \mathbf{y} contains only values in the set $\{0, 1\}$, that:

$$|l(\hat{\mathbf{y}}, \mathbf{y}) - l(\hat{\mathbf{z}}, \mathbf{y})| \leq \|\hat{\mathbf{y}} - \hat{\mathbf{z}}\|_1 \leq \sqrt{N_y} \|\hat{\mathbf{y}} - \hat{\mathbf{z}}\|_2, \quad (\text{B.5})$$

where $\|\cdot\|_1$ and $\|\cdot\|_2$ denote l_1 and l_2 norm of a vector and N_y is the number of outputs. \square

Proof of Theorem 6.2.1

Assume two different trajectories resulting from simulating the system (6.2) with parameters and initial conditions $(\mathbf{x}_0, \boldsymbol{\theta})$ and $(\mathbf{w}_0, \boldsymbol{\phi})$, respectively. We denote the corresponding trajectories by \mathbf{x} and \mathbf{w} . Let us call:

$$\|\Delta \hat{\mathbf{y}}_t\| = \|\mathbf{g}(\mathbf{x}_t, \mathbf{u}_t; \boldsymbol{\theta}) - \mathbf{g}(\mathbf{w}_t, \mathbf{u}_t; \boldsymbol{\phi})\|. \quad (\text{B.6})$$

Because \mathbf{f} and \mathbf{g} are Lipschitz in $(\mathbf{x}, \boldsymbol{\theta})$ we have:

$$\begin{aligned} \|\mathbf{f}(\mathbf{x}, \mathbf{u}_t, \boldsymbol{\theta}) - \mathbf{f}(\mathbf{w}, \mathbf{u}_t, \boldsymbol{\phi})\|^2 &\leq L_f^2 (\|\mathbf{x} - \mathbf{w}\|^2 + \|\boldsymbol{\theta} - \boldsymbol{\phi}\|^2), \\ \|\mathbf{g}(\mathbf{x}, \mathbf{u}_t, \boldsymbol{\theta}) - \mathbf{g}(\mathbf{w}, \mathbf{u}_t, \boldsymbol{\phi})\|^2 &\leq L_g^2 (\|\mathbf{x} - \mathbf{w}\|^2 + \|\boldsymbol{\theta} - \boldsymbol{\phi}\|^2), \end{aligned}$$

for all $(\mathbf{x}, \mathbf{u}_t, \boldsymbol{\theta})$ and $(\mathbf{w}, \mathbf{u}_t, \boldsymbol{\phi})$ in $(\Omega_{\mathbf{x}}, \Omega_{\mathbf{u}}, \Omega_{\boldsymbol{\theta}})$. Applying these relations recursively we get that:

$$\|\Delta \hat{\mathbf{y}}_t\|^2 \leq L_g^2 L_f^{2t} \|\mathbf{x}_0 - \mathbf{w}_0\|^2 + L_g^2 \left(\sum_{\ell=0}^t L_f^{2\ell} \right) \|\boldsymbol{\theta} - \boldsymbol{\phi}\|^2.$$

Since L_f is positive, the constant multiplying the second term in the above equation is always larger than the constant multiplying the first term. Hence, taking the square root on both sides of the above inequality and after simple manipulations, we get:

$$\|\Delta \hat{\mathbf{y}}_t\| \leq L_g S(t) \|\boldsymbol{\theta}, \mathbf{x}_0\|^T - [\boldsymbol{\phi}, \mathbf{w}_0]^T\|. \quad (\text{B.7})$$

where:

$$S(t) = \sqrt{\sum_{\ell=0}^t L_f^{2\ell}} = \begin{cases} \sqrt{t+1} & \text{if } L_f = 1; \\ \sqrt{\frac{L_f^{2t+2}-1}{L_f^2-1}} & \text{if } L_f \neq 1. \end{cases} \quad (\text{B.8})$$

Since Ω is compact and $\hat{\mathbf{y}}_t$ is a (Lipschitz) continuous function of the parameters and initial conditions, then $\hat{\mathbf{y}}_t$ is bounded in Ω , i.e. $\|\hat{\mathbf{y}}_t\| \leq M(t)$. And, it follows from (B.7) and from the existence of an invariant set¹ in Ω that $M(t) = \mathcal{O}(S(t))$.

The following inequality follows from (6.1), and from property 1 from Lemma B.2:

$$|V(\boldsymbol{\theta}, \mathbf{x}_0) - V(\boldsymbol{\phi}, \mathbf{w}_0)| \leq \frac{1}{N} \sum_{t=1}^N (K_1 L_y + K_2 M(t)) \|\Delta \hat{\mathbf{y}}_t\|, \quad (\text{B.9})$$

where $L_y = \max_{1 \leq t \leq N} \|\mathbf{y}_t\|$. And, by putting together (B.9) and (B.7):

$$|V(\boldsymbol{\theta}, \mathbf{x}_0) - V(\boldsymbol{\phi}, \mathbf{w}_0)| \leq L_{V_1} \left\| [\mathbf{x}_0, \boldsymbol{\theta}]^T - [\mathbf{w}_0, \boldsymbol{\phi}]^T \right\|,$$

for $L_V = \left(\frac{L_g}{N} \sum_{t=1}^N (K_1 L_y + K_2 M(t)) S(t) \right)$. The asymptotic analysis of this expression with regard to N yields (6.9).

Proof of Theorem 6.2.2

It follows from (B.2), and from property 2 from Lemma B.2, that:

$$\|\nabla V(\boldsymbol{\theta}, \mathbf{x}_0) - \nabla V(\boldsymbol{\phi}, \mathbf{w}_0)\| \leq \frac{1}{N} \sum_{t=1}^N K_3 L_y \|\Delta J_t\| + \|\Delta(J_t \Psi(\hat{\mathbf{y}}_t))\|, \quad (\text{B.10})$$

where we have used the notation ΔJ_t to denote the difference between J_t evaluated at $(\boldsymbol{\theta}, \mathbf{x}_0)$ and $(\boldsymbol{\phi}, \mathbf{w}_0)$. Analogously, $\Delta(J_t \Psi(\hat{\mathbf{y}}_t))$ denotes the difference between $J_t \Psi(\hat{\mathbf{y}}_t)$ evaluated at the two distinct points, where Ψ is the Lipschitz continuous function with constant K_4 defined in Lemma B.2.

From equation (B.1) it follows that:

$$J_t = C_t \sum_{\ell=1}^t \left(\prod_{j=1}^{t-\ell} A_{t-j+1} \right) B_\ell + F_t. \quad (\text{B.11})$$

Since the Jacobian of \mathbf{f} is Lipschitz with Lipschitz constant L'_f , it follows that:

$$\|\Delta A_j\|^2 \leq (L'_f)^2 (\|\mathbf{x}_j - \mathbf{w}_j\|^2 + \|\boldsymbol{\theta} - \boldsymbol{\phi}\|^2). \quad (\text{B.12})$$

Using a procedure analogous to the one used to get Equation (B.7), it follows that:

$$\|\Delta A_j\| \leq L'_f S(j) \left\| [\boldsymbol{\theta}, \mathbf{x}_0]^T - [\boldsymbol{\phi}, \mathbf{w}_0]^T \right\|, \quad (\text{B.13})$$

where $S(j)$ is defined as in (B.8). An identical formula holds for B_j and a similar formula, replacing L'_f with L'_g , holds for C_j and F_j .

Since \mathbf{f} and \mathbf{g} are Lipschitz with Lipschitz constants L_f and L_g it follows that $\|A_j\| \leq L_f$, $\|B_j\| \leq L_f$, $\|C_j\| \leq L_g$ and $\|F_j\| \leq L_g$. Hence, it follows from (B.7), (B.11), (B.13) and the repetitive application of Lemma B.1 that $\|\Delta J_t\|$ and $\|\Delta(J_t \Psi(\hat{\mathbf{y}}_t))\|$ are upper bounded by $\|[\boldsymbol{\theta}, \mathbf{x}_0]^T - [\boldsymbol{\phi}, \mathbf{w}_0]^T\|$ multiplied by the following constants:

$$\begin{aligned} L_{J,t} &= \sum_{\ell=1}^t P(t, \ell) + L'_g S(t); \\ L_{J\hat{\mathbf{y}},t} &= \sum_{\ell=1}^t Q(t, \ell) + T(t) S(t), \end{aligned}$$

¹ There are multiple ways to guarantee the invariant set premise will hold, but a very simple way is to just choose \mathbf{f} such that $\mathbf{f}(\mathbf{0}, \mathbf{u}_t; \mathbf{0}) = \mathbf{0}$. In this case, $\{\mathbf{0}\}$ is an invariant set and if $\Omega_{\boldsymbol{\theta}}$ contains this point the premise is satisfied. For this specific case, one can just choose $[\boldsymbol{\phi}, \mathbf{w}_0] = \mathbf{0}$ and it follows from (B.7) that $\|\hat{\mathbf{y}}_t\| \leq L_g S(t) \|\boldsymbol{\theta}, \mathbf{x}_0\| = \mathcal{O}(S(t))$. The more general case, for any invariant set, follows from a similar deduction.

where $T(t) = K_4(L'_g M(t) + L_g^2)$ and:

$$\begin{aligned} P(t, \ell) &= L_f^{t-\ell} \left(L_g L'_f \sum_{j=\ell}^t S(j) + L_f L'_g S(t) \right); \\ Q(t, \ell) &= L_f^{t-\ell} \left(K_4 M(t) L_g L'_f \sum_{j=\ell}^t S(j) + L_f T(t) S(t) \right). \end{aligned}$$

Hence,

$$\|\nabla V(\boldsymbol{\theta}, \mathbf{x}_0) - \nabla V(\boldsymbol{\phi}, \mathbf{w}_0)\| \leq L'_V \|[\boldsymbol{\theta}, \mathbf{x}_0]^T - [\boldsymbol{\phi}, \mathbf{w}_0]^T\|,$$

where:

$$L'_V = \frac{1}{N} \sum_{t=1}^N (K_3 L_y L_{J,t} + L_{J\dot{y},t}).$$

Putting everything together, the asymptotic analysis of L'_V results in (6.10).

Bibliography

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Technical report. Software available from tensorflow.org. Cited 2 times on pages 42 and 100.
- Acharya, U. R., Fujita, H., Oh, S. L., Hagiwara, Y., Tan, J. H., and Adam, M. (2017). Application of deep convolutional neural network for automated detection of myocardial infarction using ECG signals. *Information Sciences*, 415-416:190–198. Cited on page 138.
- Adiwardana, D., Luong, M.-T., So, D. R., Hall, J., Fiedel, N., Thoppilan, R., Yang, Z., Kulshreshtha, A., Nemade, G., Lu, Y., and Le, Q. V. (2020). Towards a Human-like Open-Domain Chatbot. *arXiv:2001.09977*. Cited on page 40.
- Aguirre, L. A. (2004). *Introdução à Identificação de Sistemas—Técnicas Lineares e Não-Lineares Aplicadas a Sistemas Reais*. Editora UFMG. 00654. Cited 2 times on pages 59 and 67.
- Aguirre, L. A., Barbosa, B. H., and Braga, A. P. (2010). Prediction and simulation errors in parameter estimation for nonlinear systems. *Mechanical Systems and Signal Processing*, 24(8):2855–2867. Cited 3 times on pages 24, 69, and 70.
- Aguirre, L. A. (2019). A Bird’s Eye View of Nonlinear System Identification *arXiv:1907.06803*. Cited on page 59.
- Akaike, H. (1974). A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723. Cited on page 36.
- Alkmim, M. B., Figueira, R. M., Marcolino, M. S., Cardoso, C. S., Pena de Abreu, M., Cunha, L. R., da Cunha, D. F., Antunes, A. P., Resende, A. G. d. A., Resende, E. S., and Ribeiro, A. L. P. (2012). Improving patient access to specialized health care: The Telehealth Network of Minas Gerais, Brazil. *Bulletin of the World Health Organization*, 90(5):373–378. Cited 3 times on pages 133, 140, and 141.
- Amorim, D. (2019). Mais 1 milhão de brasileiros passaram a trabalhar como motorista de aplicativo ou ambulante em 2018. *O Estado de S. Paulo*. Cited on page 22.
- Arjovsky, M., Shah, A., and Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML’16, pages 1120–1128, New York, NY, USA. JMLR.org. Cited 2 times on pages 101 and 118.
- Ashton, K. (2009). That ‘internet of things’ thing. *RFID journal*, 22(7):97–114. Cited on page 21.
- Baake, E., Baake, M., Bock, H., and Briggs, K. (1992). Fitting ordinary differential equations to chaotic data. *Physical Review A*, 45(8):5524. Cited on page 87.
- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473*. Cited 3 times on pages 22, 47, and 50.

- Bai, S., Kolter, J. Z., and Koltun, V. (2018a). An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. Cited 5 times on pages [120](#), [121](#), [122](#), [123](#), and [124](#).
- Bai, S., Kolter, J. Z., and Koltun, V. (2018b). An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. page 14. Cited 3 times on pages [24](#), [52](#), and [102](#).
- Beale, M. H., Hagan, M. T., and Demuth, H. B. (2017). Neural network toolbox for use with MATLAB. Technical report, Mathworks. Cited 3 times on pages [69](#), [70](#), and [84](#).
- Beck, E. J., Gill, W., and De Lay, P. R. (2016). Protecting the confidentiality and security of personal health information in low- and middle-income countries in the era of SDGs and Big Data. *Global Health Action*, 9:32089. Cited on page [132](#).
- Bejnordi, B. E., Veta, M., Johannes van Diest, P., van Ginneken, B., Karssemeijer, N., Litjens, G., van der Laak, J. A. W. M., and the CAMELYON16 Consortium, Hermsen, M., Manson, Q. F., Balkenhol, M., Geessink, O., Stathonikos, N., van Dijk, M. C., Bult, P., Beca, F., Beck, A. H., Wang, D., Khosla, A., Gargeya, R., Irshad, H., Zhong, A., Dou, Q., Li, Q., Chen, H., Lin, H.-J., Heng, P.-A., Haß, C., Bruni, E., Wong, Q., Halici, U., Öner, M. Ü., Cetin-Atalay, R., Berseth, M., Khvatkov, V., Vylegzhanin, A., Kraus, O., Shaban, M., Rajpoot, N., Awan, R., Sirinukunwattana, K., Qaiser, T., Tsang, Y.-W., Tellez, D., Annuscheit, J., Hufnagl, P., Valkonen, M., Kartasalo, K., Latonen, L., Ruusuvuori, P., Liimatainen, K., Albarqouni, S., Mungal, B., George, A., Demirci, S., Navab, N., Watanabe, S., Seno, S., Takenaka, Y., Matsuda, H., Ahmady Phoulady, H., Kovalev, V., Kalinovskiy, A., Liauchuk, V., Bueno, G., Fernandez-Carrobles, M. M., Serrano, I., Deniz, O., Racocanu, D., and Venâncio, R. (2017). Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer. *JAMA*, 318(22):2199. Cited 2 times on pages [23](#) and [132](#).
- Bengio, Y., Frasconi, P., and Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188 vol.3. Cited on page [106](#).
- Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166. Cited 4 times on pages [72](#), [101](#), [106](#), and [116](#).
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York. Cited on page [29](#).
- Bock, H. (1983). Recent Advances in Parameter Identification Problems for ODE. *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, pages 95–121. Cited on page [87](#).
- Bock, H. G. and Plitt, K.-J. (1984). A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608. Cited on page [87](#).
- Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2):223–311. Cited on page [42](#).
- Bousquet, O. and Bottou, L. (2008). The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, pages 161–168. Cited on page [42](#).
- Boyd, S. and Chua, L. (1985a). Fading memory and the problem of approximating nonlinear operators with Volterra series. *IEEE Transactions on Circuits and Systems*, 32(11):1150–1161. Cited on page [64](#).

- Boyd, S. and Chua, L. O. (1985b). Fading memory and the problem of approximating nonlinear operators with Volterra series. *IEEE Transactions on Circuits and Systems*, CAS-32(11):1150–1161. Cited on page [124](#).
- Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231. Cited on page [29](#).
- Byrd, R. H., Hribar, M. E., and Nocedal, J. (1999). An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900. Cited 2 times on pages [149](#) and [150](#).
- Carraro, T., Geiger, M., and Rannacher, R. (2014). Indirect Multiple Shooting for Nonlinear Parabolic Optimal Control Problems with Control Constraints. *SIAM Journal on Scientific Computing*, 36(2):A452–A481. Cited on page [87](#).
- Chandar, S., Sankar, C., Vorontsov, E., Kahou, S. E., and Bengio, Y. (2019). Towards Non-Saturating Recurrent Units for Modelling Long-Term Dependencies. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:3280–3287. Cited on page [101](#).
- Chen, S., Billings, S. A., Cowan, C. F. N., and Grant, P. M. (1990a). Practical identification of NARMAX models using radial basis functions. *International Journal of Control*, 52(6):1327–1350. Cited on page [61](#).
- Chen, S., Billings, S. A., and Grant, P. M. (1990b). Non-linear system identification using neural networks. *International Journal of Control*, 51(6):1191–1214. Cited 2 times on pages [81](#) and [125](#).
- Chen, S., Billings, S. A., and Luo, W. (1989). Orthogonal Least Squares Methods and Their Application to Non-Linear System Identification. *International Journal of Control*, 50(5):1873–1896. 01591. Cited on page [61](#).
- Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv:1409.1259*. Cited 2 times on pages [101](#) and [103](#).
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics. Cited 2 times on pages [50](#) and [51](#).
- Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555*. Cited on page [51](#).
- Clifford, G. D., Liu, C., Moody, B., Lehman, L.-w. H., Silva, I., Li, Q., Johnson, A. E., and Mark, R. G. (2017). AF Classification from a Short Single Lead ECG Recording: The PhysioNet/Computing in Cardiology Challenge 2017. *Computing in Cardiology*, 44. Cited 3 times on pages [132](#), [134](#), and [139](#).
- Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46. Cited 2 times on pages [143](#) and [145](#).
- Conn, A. R., Gould, N. I. M., and Toint, P. L. (2000). *Trust-Region Methods*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA. Cited on page [149](#).
- CUBANSKI, D., CYGANSKI, D., ANTMAN, E. M., and FELDMAN, C. L. (1994). A Neural Network System for Detection of Atrial Fibrillation in Ambulatory Electrocardiograms. *Journal of Cardiovascular Electrophysiology*, 5(7):602–608. Cited on page [137](#).

- Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. (2017). Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 933–941. JMLR. org. Cited 3 times on pages 24, 52, and 102.
- De Fauw, J., Ledsam, J. R., Romera-Paredes, B., Nikolov, S., Tomasev, N., Blackwell, S., Askham, H., Glorot, X., O’Donoghue, B., Visentin, D., van den Driessche, G., Lakshminarayanan, B., Meyer, C., Mackinder, F., Bouton, S., Ayoub, K., Chopra, R., King, D., Karthikesalingam, A., Hughes, C. O., Raine, R., Hughes, J., Sim, D. A., Egan, C., Tufail, A., Montgomery, H., Hassabis, D., Rees, G., Back, T., Khaw, P. T., Suleyman, M., Cornebise, J., Keane, P. A., and Ronneberger, O. (2018). Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature Medicine*, 24(9):1342–1350. Cited 2 times on pages 23 and 132.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805*. Cited 2 times on pages 47 and 50.
- Diaconescu, E. (2008). The use of NARX neural networks to predict chaotic time series. *WSEAS Transactions on Computer Research*, 3(3):182–191. Cited 3 times on pages 69, 70, and 84.
- Doya, K. (1993). Bifurcations of Recurrent Neural Networks in Gradient Descent Learning. Cited 2 times on pages 107 and 116.
- Dvoretzky, A., Kiefer, J., and Wolfowitz, J. (1956). Asymptotic Minimax Character of the Sample Distribution Function and of the Classical Multinomial Estimator. *The Annals of Mathematical Statistics*, 27(3):642–669. Cited 2 times on pages 11 and 97.
- Eckhard, D., Bazanella, A. S., Rojas, C. R., and Hjalmarsson, H. (2017). Cost function shaping of the output error criterion. *Automatica*, 76:53–60. Cited on page 86.
- Edouard Grave, Armand Joulin, N. U. (2017). Improving neural language models with a continuous cache. In *International Conference on Learning Representations*. Cited on page 113.
- Efron, B. and Tibshirani, R. J. (1994). *An Introduction to the Bootstrap*. CRC press. Cited 2 times on pages 136 and 145.
- Espinoza, M., Pelckmans, K., Hoegaerts, L., Suykens, J. A., and De Moor, B. (2004). A comparative study of LS-SVM’s applied to the Silver Box identification problem. *IFAC Proceedings Volumes*, 37(13):369–374. Cited on page 128.
- Espinoza, M., Suykens, J. A., and De Moor, B. (2005). Kernel based partially linear models and nonlinear identification. *IEEE Trans. Autom. Control*, 50(10):1602–1606. Cited on page 128.
- Estes, N. A. M. (2013). Computerized interpretation of ECGs: Supplement not a substitute. *Circulation. Arrhythmia and Electrophysiology*, 6(1):2–4. Cited 2 times on pages 132 and 142.
- Farina, M. and Piroddi, L. (2008). Some convergence properties of multi-step prediction error identification criteria. In *2008 47th IEEE Conference on Decision and Control*, pages 756–761. Cited 3 times on pages 71, 91, and 92.
- Farina, M. and Piroddi, L. (2010). An iterative algorithm for simulation error based identification of polynomial input–output models using multi-step prediction. *International Journal of Control*, 83(7):1442–1456. Cited 2 times on pages 61 and 71.
- Farina, M. and Piroddi, L. (2011). Simulation error minimization identification based on multi-stage prediction. *International Journal of Adaptive Control and Signal Processing*, 25(5):389–406. Cited 3 times on pages 71, 91, and 92.

- Farina, M. and Piroddi, L. (2012). Identification of polynomial input/output recursive models with simulation error minimisation methods. *International Journal of Systems Science*, 43(2):319–333. Cited 3 times on pages 24, 61, and 71.
- Fletcher, R., Authority, U. K. A. E., and H.M.S.O. (1971). *A Modified Marquardt Subroutine for Non-Linear Least Squares*. AERE Report. Theoretical Physics Division, Atomic Energy Research Establishment. 00003. Cited on page 73.
- Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The Elements of Statistical Learning*, volume 1. Springer series in statistics New York. Cited 2 times on pages 29 and 36.
- Gehring, J., Auli, M., Grangier, D., and Dauphin, Y. (2017). A Convolutional Encoder Model for Neural Machine Translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 123–135. Cited 3 times on pages 24, 52, and 102.
- Geiger, A., Lenz, P., and Urtasun, R. (2012). Are we ready for autonomous driving? The KITTI vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, Providence, RI. IEEE. Cited on page 47.
- Geisert, M. and Mansard, N. (2016). Trajectory Generation for Quadrotor Based Systems Using Numerical Optimal Control. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2958–2964. IEEE. Cited on page 87.
- Giri, F. and Bai, E.-W., editors (2010). *Block-oriented Nonlinear System Identification*. Springer London. Cited on page 124.
- Goldberg, D. E. (2000). *Genetic Algorithms—In Search, Optimization & Machine Learning, Revised Ed.* Addison Wesley, New Delhi. Cited on page 37.
- Goldberger, A. L., Amaral, L. A. N., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K., and Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. *Circulation*, 101(23). Cited on page 139.
- Golub, G. H. and Van Loan, C. F. (2012). *Matrix Computations*, volume 3. JHU Press. Cited on page 76.
- Gong, C., He, D., Tan, X., Qin, T., Wang, L., and Liu, T.-Y. (2018). Frage: Frequency-agnostic word representation. *arXiv:1809.06858*. Cited on page 113.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. Cited on page 146.
- Goto, S., Kimura, M., Katsumata, Y., Goto, S., Kamatani, T., Ichihara, G., Ko, S., Sasaki, J., Fukuda, K., and Sano, M. (2019). Artificial intelligence to predict needs for urgent revascularization from 12-leads electrocardiography in emergency patients. *PLOS ONE*, 14(1):e0210103. Cited on page 139.
- Gould, N. I., Hribar, M. E., and Nocedal, J. (2001). On the solution of equality constrained quadratic programming problems arising in optimization. *SIAM Journal on Scientific Computing*, 23(4):1376–1395. Cited on page 150.
- Graves, A., Jaitly, N., and Mohamed, A.-r. (2013). Hybrid speech recognition with Deep Bidirectional LSTM. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 273–278, Olomouc, Czech Republic. IEEE. Cited on page 23.
- Hagan, M. T. and Menhaj, M. B. (1994). Training feedforward networks with the Marquardt algorithm. *Neural Networks, IEEE Transactions on*, 5(6):989–993. Cited 2 times on pages 72 and 74.

- Hannun, A. Y., Rajpurkar, P., Haghpanahi, M., Tison, G. H., Bourn, C., Turakhia, M. P., and Ng, A. Y. (2019). Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network. *Nature Medicine*, 25(1):65–69. Cited 5 times on pages [132](#), [134](#), [138](#), [139](#), and [144](#).
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852*. Cited on page [144](#).
- He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. Cited 5 times on pages [56](#), [58](#), [123](#), [134](#), and [143](#).
- He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity Mappings in Deep Residual Networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision – ECCV 2016*, pages 630–645. Springer International Publishing. Cited 2 times on pages [143](#) and [144](#).
- He, W. and Zhang, S. (2017). Control Design for Nonlinear Flexible Wings of a Robotic Aircraft. *IEEE Transactions on Control Systems Technology*, 25(1):351–357. Cited on page [21](#).
- Helfrich, K., Willmott, D., and Ye, Q. (2018). Orthogonal recurrent neural networks with scaled Cayley transform. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1969–1978, Stockholmsmässan, Stockholm Sweden. PMLR. Cited on page [101](#).
- Hinton, G. (2018). Deep learning—a technology with the potential to transform health care. *JAMA*, 320(11):1101–1102. Cited 2 times on pages [47](#) and [132](#).
- Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. (2012). Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97. Cited 2 times on pages [22](#) and [47](#).
- Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, volume 1, page 12. Amherst, MA. Cited on page [47](#).
- Hjalmarsson, H. and Schoukens, J. (2004). On direct identification of physical parameters in non-linear models. *IFAC Proceedings Volumes*, 37(13):375–380. Cited on page [128](#).
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780. Cited 8 times on pages [51](#), [101](#), [103](#), [104](#), [108](#), [110](#), [116](#), and [118](#).
- Hong, S., Wu, M., Zhou, Y., Wang, Q., Shang, J., Li, H., and Xie, J. (2017). ENCASE: An ENsemble CLASsifiEr for ECG Classification Using Expert Features and Deep Neural Networks. In *2017 Computing in Cardiology Conference*. Cited on page [134](#).
- Horbelt, W., Timmer, J., Büchner, M., Meucci, R., and Ciofini, M. (2001). Identifying physical properties of a CO₂ LASER by dynamical modeling of measured time series. *Physical Review E*, 64(1):016222. Cited on page [87](#).
- Hornik, K., Stinchcombe, M., and White, H. (1989a). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366. Cited on page [61](#).

- Hornik, K., Stinchcombe, M., and White, H. (1989b). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366. Cited on page [120](#).
- Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Scherer, D., Müller, A., and Behnke, S. (2010). Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In Diamantaras, K., Duch, W., and Iliadis, L. S., editors, *Artificial Neural Networks – ICANN 2010*, volume 6354, pages 92–101. Springer Berlin Heidelberg. Cited on page [144](#).
- Ifju, P., Jenkins, D., Ettinger, S., Lian, Y., Shyy, W., and Waszak, M. (2002). Flexible-wing-based micro air vehicles. In *40th AIAA Aerospace Sciences Meeting & Exhibit*, page 705. Cited on page [21](#).
- Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456. PMLR. Cited 3 times on pages [122](#), [123](#), and [144](#).
- Jambukia, S. H., Dabhi, V. K., and Prajapati, H. B. (2015). Classification of ECG signals using machine learning techniques: A survey. In *Proceedings of the International Conference on Advances in Computer Engineering and Applications (ICACEA)*, pages 714–721. IEEE. Cited 2 times on pages [137](#) and [138](#).
- Jing, L., Shen, Y., Dubcek, T., Peurifoy, J., Skirlo, S., LeCun, Y., Tegmark, M., and Soljagic, M. (2017). Tunable Efficient Unitary Neural Networks (EUNN) and their application to RNNs. *Proceedings of the 34th International Conference on Machine Learning*, page 9. Cited on page [101](#).
- Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016). Neural Machine Translation in Linear Time. *arXiv:1610.10099*. Cited 3 times on pages [24](#), [52](#), and [102](#).
- Kamaleswaran, R., Mahajan, R., and Akbilgic, O. (2018). A robust deep convolutional neural network for the classification of abnormal cardiac rhythm using single lead electrocardiograms of variable length. *Physiological Measurement*, 39(3):035006. Cited on page [134](#).
- Kamiński, W., Strumitto, P., and Tomczak, E. (1996). Genetic algorithms and artificial neural networks for description of thermal deterioration processes. *Drying Technology*, 14(9):2117–2133. Cited 3 times on pages [69](#), [70](#), and [84](#).
- Kanuparthi, B., Arpit, D., Kerg, G., Ke, N. R., Mitliagkas, I., and Bengio, Y. (2019). H-DETACH: Modifying the LSTM Gradient Towards Better Optimization. *Proceedings of the International Conference for Learning Representations (ICLR)*, page 19. Cited on page [101](#).
- Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks. Cited on page [24](#).
- Kayacan, E., Kayacan, E., and Khanesar, M. A. (2015). Identification of nonlinear dynamic systems using type-2 fuzzy neural networks—A novel learning algorithm and a comparative study. *IEEE Transactions on Industrial Electronics*, 62(3):1716–1724. Cited on page [61](#).
- Kerg, G., Goyette, K., Touzel, M. P., Gidel, G., Vorontsov, E., Bengio, Y., and Lajoie, G. (2019). Non-normal Recurrent Neural Network (nnRNN): Learning long time dependencies while improving expressivity with transient dynamics. *arXiv:1905.12080*. Cited on page [101](#).
- Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836*. Cited on page [42](#).

- Khalil, H. K. (2002). *Nonlinear Systems*. Upper Saddle River, third edition. Cited on page 107.
- Khan, E. A., Elgamal, M. A., and Shaarawy, S. M. (2015). Forecasting the Number of Muslim Pilgrims Using NARX Neural Networks with a Comparison Study with Other Modern Methods. *British Journal of Mathematics & Computer Science*, 6(5):394. Cited 3 times on pages 69, 70, and 84.
- Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference for Learning Representations (ICLR)*. Cited 6 times on pages 15, 46, 108, 123, 125, and 144.
- Kligfield, P., Gettes, L. S., Bailey, J. J., Childers, R., Deal, B. J., Hancock, E. W., van Herpen, G., Kors, J. A., Macfarlane, P., Mirvis, D. M., Pahlm, O., Rautaharju, P., and Wagner, G. S. (2007). Recommendations for the Standardization and Interpretation of the Electrocardiogram. *Journal of the American College of Cardiology*, 49(10):1109. Cited on page 143.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105. Cited 4 times on pages 22, 47, 52, and 56.
- LabVolt (2015). Mobile Instrumentation and Process Control Training Systems. Technical report, Festo. 00000. Cited on page 79.
- Lalee, M., Nocedal, J., and Plantenga, T. (1998). On the implementation of an algorithm for large-scale equality constrained optimization. *SIAM Journal on Optimization*, 8(3):682–706. Cited 4 times on pages 15, 46, 92, and 149.
- Laurent, T. and von Brecht, J. (2016). A recurrent neural network without chaos. *arXiv:1612.06212*. Cited 2 times on pages 114 and 117.
- LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444. Cited 2 times on pages 22 and 47.
- Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. 14818. Cited 2 times on pages 52 and 56.
- LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient BackProp. In *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pages 9–50. Springer, Berlin, Heidelberg. Cited 5 times on pages 10, 11, 79, 80, and 97.
- Leontaritis, I. J. and Billings, S. A. (1987). Experimental design and identifiability for non-linear systems. *International Journal of Systems Science*, 18(1):189–202. Cited on page 67.
- Lezcano-Casado, M. (2019). Trivializations for Gradient-Based Optimization on Manifolds. *Advances in Neural Information Processing Systems*. Cited on page 101.
- Lezcano-Casado, M. and Martínez-Rubio, D. (2019). Cheap Orthogonal Constraints in Neural Networks: A Simple Parametrization of the Orthogonal and Unitary Group. In *International Conference on Machine Learning*, pages 3794–3803. Cited 3 times on pages 101, 102, and 108.
- Liu, D. C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528. Cited on page 41.
- Ljung, L. (1976). On The Consistency of Prediction Error Identification Methods. In Mehra, R. K. and Lainiotis, D. G., editors, *Mathematics in Science and Engineering*, volume 126 of *System Identification Advances and Case Studies*, pages 121–164. Elsevier. Cited on page 62.

- Ljung, L. (1978). Convergence analysis of parametric identification methods. *IEEE Transactions on Automatic Control*, 23(5):770–783. Cited 4 times on pages 62, 65, 66, and 69.
- Ljung, L. (1998). *System Identification*. Springer. Cited 6 times on pages 21, 59, 61, 62, 67, and 69.
- Ljung, L. (2010). Perspectives on system identification. *Annual Reviews in Control*, 34(1):1–12. Cited on page 61.
- Ljung, L. and Caines, P. E. (1980). Asymptotic normality of prediction error estimators for approximate system models. *Stochastics*, 3(1-4):29–46. Cited 2 times on pages 62 and 66.
- Ljung, L., Zhang, Q., Lindskog, P., and Juditski, A. (2004). Estimation of grey box and black box models for non-linear circuit data. *IFAC Proceedings Volumes*, 37(13):399–404. Cited on page 128.
- Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440. Cited on page 52.
- Luo, S. and Johnston, P. (2010). A review of electrocardiogram filtering. *Journal of Electrocardiology*, 43(6):486–496. Cited on page 139.
- Luong, T., Pham, H., and Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics. Cited on page 50.
- Lyon, A., Minchol  , A., Mart  nez, J. P., Laguna, P., and Rodriguez, B. (2018). Computational techniques for ECG analysis and interpretation in light of their contribution to medical advances. *Journal of the Royal Society Interface*, 15(138). Cited on page 133.
- Macfarlane, P., Devine, B., Latif, S., McLaughlin, S., Shoat, D., and Watts, M. (1990). Methodology of ECG interpretation in the Glasgow program. *Methods of information in medicine*, 29(04):354–361. Cited on page 141.
- Macfarlane, P. W., Devine, B., and Clark, E. (2005). The university of glasgow (Uni-G) ECG analysis program. In *Computers in Cardiology*, pages 451–454. Cited 2 times on pages 137 and 141.
- Macfarlane, P. W. and Latif, S. (1996). Automated serial ECG comparison based on the Minnesota code. *Journal of Electrocardiology*, 29:29–34. Cited on page 141.
- Maduranga, K. D., Helfrich, K. E., and Ye, Q. (2019). Complex unitary recurrent neural networks using scaled cayley transform. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4528–4535. Cited on page 101.
- Maheswaranathan, N., Williams, A., Golub, M. D., Ganguli, S., and Sussillo, D. (2019). Reverse engineering recurrent networks for sentiment classification reveals line attractor dynamics. *arXiv:1906.10720*. Cited 2 times on pages 106 and 116.
- Mant, J., Fitzmaurice, D. A., Hobbs, F. D. R., Jowett, S., Murray, E. T., Holder, R., Davies, M., and Lip, G. Y. H. (2007). Accuracy of diagnosing atrial fibrillation on electrocardiogram by primary care practitioners and interpretative diagnostic software: Analysis of data from screening for atrial fibrillation in the elderly (SAFE) trial. *BMJ (Clinical research ed.)*, 335(7616):380. Cited 2 times on pages 23 and 132.

- Marconato, A., Sjöberg, J., Suykens, J., and Schoukens, J. (2012). Identification of the Silverbox benchmark using nonlinear state-space models. *IFAC Proceedings Volumes*, 45(16):632–637. Cited 2 times on pages 127 and 128.
- Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441. Cited 3 times on pages 15, 46, and 73.
- May, R. M. (1976). Simple mathematical models with very complicated dynamics. *Nature*, 261(5560):459–467. Cited on page 94.
- McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133. Cited on page 47.
- McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157. Cited 2 times on pages 136 and 145.
- Menezes, J. M. P. and Barreto, G. A. (2008). Long-term time series prediction with the NARX network: An empirical evaluation. *Neurocomputing*, 71(16):3335–3343. Cited on page 79.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *arXiv:1609.07843*. Cited on page 113.
- Mhammedi, Z., Hellicar, A., Rahman, A., and Bailey, J. (2017). Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2401–2409. JMLR. org. Cited on page 101.
- Miller, J. and Hardt, M. (2018). Stable Recurrent Models. *arXiv:1805.10369*. Cited 6 times on pages 101, 102, 107, 108, 113, and 118.
- Mokus, J. (1974). On Bayesian Methods for Seeking the Extremum. *Optimization Techniques*, pages 400–404. Cited on page 37.
- More, J. J. (1978). The Levenberg-Marquardt algorithm: Implementation and theory. In *Numerical Analysis*, pages 105–116. Springer. Cited on page 78.
- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. MIT Press, Cambridge, MA. Cited on page 29.
- Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1):4–27. Cited 6 times on pages 24, 61, 69, 70, 72, and 84.
- Nascimento, B. R., Brant, L. C. C., Marino, B. C. A., Passaglia, L. G., and Ribeiro, A. L. P. (2019). Implementing myocardial infarction systems of care in low/middle-income countries. *Heart*, 105(1):20. Cited on page 141.
- Naylor C (2018). On the prospects for a (deep) learning health care system. *JAMA*, 320(11):1099–1100. Cited 2 times on pages 47 and 132.
- Nelles, O. (2013). *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models*. Springer Science & Business Media. Cited 4 times on pages 59, 61, 62, and 86.
- Nesterov, Y. (1998). *Introductory Lectures On Convex Programming*. Springer Science & Business Media. Cited 2 times on pages 88 and 106.

- Newey, W. K. and McFadden, D. (1994). Large sample estimation and hypothesis testing. *Handbook of econometrics*, 4:2111–2245. Cited on page 66.
- Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 2nd ed edition. OCLC: ocm68629100. Cited 7 times on pages 38, 39, 72, 73, 78, 110, and 150.
- Noël, J. P. and Schoukens, J. (2018). Grey-box state-space identification of nonlinear mechanical vibrations. *International Journal of Control*, 91(5):1118–1139. Cited on page 64.
- Nørgaard, P. M., Ravn, O., Poulsen, N. K., and Hansen, L. K. (2000). *Neural Networks for Modelling and Control of Dynamic Systems-A Practitioner's Handbook*. Springer-London. Cited 2 times on pages 61 and 62.
- Oppenheim, A. V. (1999). *Discrete-Time Signal Processing*. Pearson Education India. Cited on page 55.
- Osborne, M. R. (1969). On Shooting Methods for Boundary Value Problems. *Journal of Mathematical Analysis and Applications*, 27(2):417–433. Cited on page 87.
- Paduart, J. (2008). *Identification of Nonlinear Systems using Polynomial Nonlinear State Space Models*. PhD thesis, Vrije Universiteit Brussel. Cited on page 128.
- Paduart, J., Horváth, G., and Schoukens, J. (2004). Fast identification of systems with nonlinear feedback. *IFAC Proceedings Volumes*, 37(13):381–385. Cited on page 128.
- Paduart, J., Lauwers, L., Swevers, J., Smolders, K., Schoukens, J., and Pintelon, R. (2010). Identification of nonlinear systems using Polynomial Nonlinear State Space models. *Automatica*, 46(4):647–656. 00000. Cited on page 61.
- Palm, G. (1979). On representation and approximation of nonlinear systems. *Biological Cybernetics*, 34(1):49–52. Cited on page 124.
- Park, J. and Sandberg, I. W. (1991). Universal Approximation Using Radial-Basis-Function Networks. *Neural computation*, 3(2):246–257. 03486. Cited on page 61.
- Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13*, pages III–1310–III–1318, Atlanta, GA, USA. JMLR.org. Cited 7 times on pages 88, 101, 102, 103, 107, 108, and 116.
- Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. Technical report. Cited 2 times on pages 42 and 100.
- Patan, K. and Korbicz, J. (2012). Nonlinear Model Predictive Control of a Boiler Unit: A Fault Tolerant Control Study. *International Journal of Applied Mathematics and Computer Science*, 22(1):225–237. Cited 4 times on pages 69, 70, 82, and 86.
- Peifer, M. and Timmer, J. (2007). Parameter estimation in ordinary differential equations for biochemical processes using the method of multiple shooting. *IET Systems Biology*, 1(2):78–88. 00000. Cited on page 87.
- Pepona, E., Paoletti, S., Garulli, A., and Date, P. (2011). Identification of piecewise affine LFR models of interconnected systems. *IEEE Trans. Control Syst. Technol.*, 19(1):148–155. Cited on page 128.

- Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv:1802.05365*. Cited 2 times on pages 23 and 103.
- Petrović, E., Čojbašić, Ž., Ristić-Durrant, D., Nikolić, V., Ćirić, I., and Jan Matić, S. (2013). Kalman Filter and NARX Neural Network for Robot Vision Based Human Tracking. *Facta Universitatis, Series: Automatic Control And Robotics*, 12(1):43–51. 00009. Cited 3 times on pages 69, 70, and 84.
- Pintelon, R. and Schoukens, J. (2012). *System identification: A frequency domain approach*. Wiley-IEEE Press, 2 edition. Cited on page 128.
- Piroddi, L. and Spinelli, W. (2003). An identification algorithm for polynomial NARX models based on simulation error minimization. *International Journal of Control*, 76(17):1767–1781. Cited 3 times on pages 24, 71, and 86.
- Poljak, B. T. (1987). *Introduction to Optimization*. Optimization Software. Cited on page 38.
- Prineas, R. J., Crow, R. S., and Zhang, Z.-M. (2009). *The Minnesota Code Manual of Electrocardiographic Findings*. Springer Science & Business Media. Cited on page 141.
- Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151. Cited on page 123.
- Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training. page 12. Cited on page 102.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. page 24. Cited on page 102.
- Rahhal, M. A., Bazi, Y., AlHichri, H., Alajlan, N., Melgani, F., and Yager, R. (2016). Deep learning approach for active classification of electrocardiogram signals. *Information Sciences*, 345:340–354. Cited 2 times on pages 138 and 139.
- Rahman, M. F., Devanathan, R., and Kuanyi, Z. (2000). Neural Network Approach for Linearizing Control of Nonlinear Process Plants. *IEEE Transactions on Industrial Electronics*, 47(2):470–477. 00030. Cited 3 times on pages 69, 70, and 84.
- Rautaharju, P. M., Surawicz, B., and Gettes, L. S. (2009). AHA/ACCF/HRS Recommendations for the Standardization and Interpretation of the Electrocardiogram: Part IV: The ST Segment, T and U Waves, and the QT Interval A Scientific Statement From the American Heart Association Electrocardiography and Arrhythmias Committee, Council on Clinical Cardiology; the American College of Cardiology Foundation; and the Heart Rhythm Society Endorsed by the International Society for Computerized Electrocardiology. *Journal of the American College of Cardiology*, 53(11):982–991. Cited on page 139.
- Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788. 00000. Cited on page 52.
- Ribeiro, A. H. and Aguirre, L. A. (2015). Selecting transients automatically for the identification of models for an oil well. *IFAC-PapersOnLine*, 48(6):154–158. Cited 2 times on pages 67 and 72.
- Ribeiro, A. H. and Aguirre, L. A. (2017). Shooting Methods for Parameter Estimation of Output Error Models. *IFAC-PapersOnLine*, 50(1):13998–14003. Cited on page 87.

- Ribeiro, A. H. and Aguirre, L. A. (2018). “Parallel Training Considered Harmful?”: Comparing series-parallel and parallel feedforward network training. *Neurocomputing*, 316:222–231. Cited 3 times on pages [11](#), [96](#), and [97](#).
- Ribeiro, A. H., Tiels, K., Umenberger, J., Schön, T. B., and Aguirre, L. A. (2019). On the Smoothness of Nonlinear System Identification. *Provisionally accepted at Automatica*. Cited on page [108](#).
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. Cited on page [47](#).
- Rossiter, J. A. and Kouvaritakis, B. (2001). Modelling and implicit modelling for predictive control. *International Journal of Control*, 74(11):1085–1095. Cited 2 times on pages [91](#) and [92](#).
- Roth, G. A., Abate, D., Abate, K. H., Abay, S. M., Abbafati, C., Abbasi, N., Abbastabar, H., Abd-Allah, F., Abdela, J., Abdelalim, A., Abdollahpour, I., Abdulkader, R. S., Abebe, H. T., Abebe, M., Abebe, Z., Abejie, A. N., Abera, ... and Murray, C. J. L. (2018). Global, regional, and national age-sex-specific mortality for 282 causes of death in 195 countries and territories, 1980–2017: A systematic analysis for the Global Burden of Disease Study 2017. *The Lancet*, 392(10159):1736–1788. Cited on page [132](#).
- Rubin, J., Parvaneh, S., Rahman, A., Conroy, B., and Babaeizadeh, S. (2017). Densely Connected Convolutional Networks and Signal Quality Analysis to Detect Atrial Fibrillation Using Short Single-Lead ECG Recordings. *arXiv:1710.05817*. Cited on page [138](#).
- Rudin, W. (1964). *Principles of Mathematical Analysis*. International Series in Pure and Applied Mathematics. McGraw-Hill. 00000. Cited 2 times on pages [89](#) and [105](#).
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1. Cited on page [47](#).
- Saad, M., Bigras, P., Dessaint, L.-A., and Al-Haddad, K. (1994). Adaptive robot control using neural networks. *IEEE Transactions on Industrial Electronics*, 41(2):173–181. 00000. Cited 3 times on pages [69](#), [70](#), and [84](#).
- Sabahi, F. and Akbarzadeh-T, M. R. (2016). Extended fuzzy logic: Sets and systems. *IEEE Trans. Fuzzy Syst.*, 24(3):530–543. Cited on page [128](#).
- Saggar, M., Merigli, T., Andoni, S., and Miikkulainen, R. (2007). System Identification for the Hodgkin-Huxley Model Using Artificial Neural Networks. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference On*, pages 2239–2244. IEEE. 00016. Cited 3 times on pages [69](#), [70](#), and [84](#).
- Saito, T. and Rehmsmeier, M. (2015). The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLOS ONE*, 10(3):e0118432. Cited 2 times on pages [144](#) and [145](#).
- Salimans, T. and Kingma, D. P. (2016). Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. Cited on page [122](#).
- Sarode, K. D., Kumar, V. R., and Kulkarni, B. (2015). Embedded Multiple Shooting Methodology in a Genetic Algorithm Framework for Parameter Estimation and State Identification of Complex Systems. *Chemical Engineering Science*, 134:605–618. Cited on page [87](#).
- Sassi, R., Bond, R. R., Cairns, A., Finlay, D. D., Guldenring, D., Libretti, G., Isola, L., Vaglio, M., Poeta, R., Campana, M., Cuccia, C., and Badilini, F. (2017 Nov - Dec). PDF-ECG in clinical practice: A model for long-term preservation of digital 12-lead ECG data. *Journal of Electrocardiology*, 50(6):776–780. Cited on page [133](#).

- Schetzen, M. (2006). *The Volterra & Wiener Theories of Nonlinear Systems*. Krieger Publishing Company, Malabar, Florida. Cited on page [123](#).
- Schläpfer, J. and Wellens, H. J. (2017). Computer-Interpreted Electrocardiograms: Benefits and Limitations. *Journal of the American College of Cardiology*, 70(9):1183. Cited 2 times on pages [132](#) and [142](#).
- Schoukens, M., Marconato, A., Pintelon, R., Vandersteen, G., and Rolain, Y. (2015). Parametric identification of parallel Wiener-Hammerstein systems. *Automatica*, 51:111–122. Cited on page [124](#).
- Schoukens, M. and Noël, J.-P. (2017). F-16 aircraft benchmark based on ground vibration test data. In *Workshop on Nonlinear System Identification Benchmarks*, Brussels, Belgium. Cited 2 times on pages [120](#) and [128](#).
- Schoukens, M. and Tiels, K. (2017). Identification of block-oriented nonlinear systems starting from linear approximations: A survey. *Automatica*, 85:272–292. Cited on page [124](#).
- Schwarz, G. (1978). Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464. 00000. Cited on page [36](#).
- Shah, A. P. and Rubin, S. A. (2007 Sep-Oct). Errors in the computerized electrocardiogram interpretation of cardiac rhythm. *Journal of Electrocardiology*, 40(5):385–390. Cited 2 times on pages [132](#) and [142](#).
- Shashikumar, S. P., Shah, A. J., Clifford, G. D., and Nemati, S. (2018). Detection of Paroxysmal Atrial Fibrillation Using Attention-based Bidirectional Recurrent Neural Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 715–723, New York, NY, USA. ACM. Cited 2 times on pages [138](#) and [139](#).
- Sinai, Y. G. (1959). On the notion of entropy of a dynamical system. In *Dokl. Akad. Nauk. SSSR*, volume 124, page 768. Cited on page [117](#).
- Singh, M., Singh, I., and Verma, A. (2013). Identification on non linear series-parallel model using neural network. *MIT Int. J. Electr. Instrumen. Eng*, 3(1):21–23. 00000. Cited 3 times on pages [69](#), [70](#), and [84](#).
- Singya, P. K., Kumar, N., and Bhatia, V. (2017). Mitigating NLD for Wireless Networks: Effect of Nonlinear Power Amplifiers on Future Wireless Communication Networks. *IEEE Microwave Magazine*, 18(5):73–90. Cited on page [21](#).
- Smith, S. W., Walsh, B., Grauer, K., Wang, K., Rapin, J., Li, J., Fennell, W., and Taboulet, P. (2019). A deep neural network learning algorithm outperforms a conventional algorithm for emergency department electrocardiogram interpretation. *Journal of Electrocardiology*, 52:88–95. Cited 2 times on pages [137](#) and [139](#).
- Söderström, T. and Stoica, P. (1988). *System Identification*. Prentice-Hall, Inc. 00000. Cited 3 times on pages [21](#), [59](#), and [62](#).
- Spivak, M. (1998). *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*. Mathematics Monograph Series. Perseus Books, Cambridge, Mass. Cited on page [104](#).
- Sragner, L., Schoukens, J., and Horváth, G. (2004). Modelling of a slightly nonlinear system: a neural network approach. *IFAC Proceedings Volumes*, 37(13):387–392. Cited on page [128](#).
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958. Cited 3 times on pages [122](#), [123](#), and [144](#).

- Stead, W. W. (2018). Clinical implications and challenges of artificial intelligence and deep learning. *JAMA*, 320(11):1107–1108. Cited 2 times on pages 47 and 132.
- Su, H.-T. and McAvoy, T. J. (1993). Neural Model Predictive Control of Nonlinear Chemical Processes. In *Intelligent Control, 1993., Proceedings of the 1993 IEEE International Symposium On*, pages 358–363. IEEE. Cited 3 times on pages 69, 70, and 86.
- Su, H. T., McAvoy, T. J., and Werbos, P. (1992). Long-term predictions of chemical processes using recurrent neural networks: A parallel training approach. *Industrial & Engineering Chemistry Research*, 31(5):1338–1352. 00000. Cited 3 times on pages 69, 70, and 86.
- Sussillo, D. and Barak, O. (2013). Opening the Black Box: Low-Dimensional Dynamics in High-Dimensional Recurrent Neural Networks. *Neural Computation*, 25(3):626–649. Cited 3 times on pages 106, 108, and 116.
- Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112. 00000. Cited 2 times on pages 23 and 50.
- Sutton, R. S. and Barto, A. G. (2011). Reinforcement learning: An introduction. Cited on page 29.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9. 00000. Cited on page 56.
- Terzi, E., Fagiano, L., Farina, M., and Scattolini, R. (2018). Learning multi-step prediction models for receding horizon control. In *2018 European Control Conference (ECC)*, pages 1335–1340. Cited 2 times on pages 91 and 92.
- Tibshirani, R. (1996). Regression shrinkage and selection via the LASSO. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288. 00000. Cited on page 33.
- Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning. Cited on page 123.
- Tiels, K. (2015). *Wiener system identification with generalized orthonormal basis functions*. PhD thesis, Vrije Universiteit Brussel. Cited 2 times on pages 127 and 128.
- Tiels, K. (2017). Polynomial nonlinear state-space modeling of the F-16 aircraft benchmark. In *Workshop on Nonlinear System Identification Benchmarks*, Brussels, Belgium. Cited on page 128.
- Tijani, I. B., Akmeliawati, R., Legowo, A., and Budiyo, A. (2014). Nonlinear Identification of a Small Scale Unmanned Helicopter Using Optimized NARX Network with Multiobjective Differential Evolution. *Engineering Applications of Artificial Intelligence*, 33:99–115. Cited 3 times on pages 69, 70, and 84.
- Tikk, D., Kóczy, L. T., and Gedeon, T. D. (2003). A survey on universal approximation and its limits in soft computing techniques. *International Journal of Approximate Reasoning*, 33(2):185–202. 00000. Cited on page 61.
- Timmer, J., Rust, H., Horbelt, W., and Voss, H. (2000). Parametric, nonparametric and parametric modelling of a chaotic circuit time series. *Physics Letters A*, 274(3):123–134. 00000. Cited on page 87.
- Topol, E. (2019). *Deep Medicine: How Artificial Intelligence Can Make Healthcare Human Again*. Hachette UK. Cited 2 times on pages 23 and 47.

- Tripathy, R. K., Bhattacharyya, A., and Pachori, R. B. (2019). A Novel Approach for Detection of Myocardial Infarction From ECG Signals of Multiple Electrodes. *IEEE Sensors Journal*, 19(12):4509–4517. Cited on page 137.
- van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. *arXiv:1609.03499*. Cited 3 times on pages 24, 52, and 102.
- Van Mulders, A., Schoukens, J., and Vanbeylen, L. (2013). Identification of systems with localised nonlinearity: From state-space to block-structured models. *Automatica*, 49(5):1392–1396. Cited on page 128.
- Van Mulders, A., Schoukens, J., Volckaert, M., and Diehl, M. (2010). Two nonlinear optimization methods for black box identification compared. *Automatica*, 46(10):1675–1681. Cited on page 87.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is All you Need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc. Cited 5 times on pages 22, 23, 47, 50, and 102.
- Veloso, A., Meira Jr, W., and Zaki, M. J. (2006). Lazy Associative Classification. In *Proceedings of the 6th International Conference on Data Mining (ICDM)*, pages 645–654. Cited on page 141.
- Verdult, V. (2004). Identification of local linear state-space models: The Silver-box case study. *IFAC Proceedings Volumes*, 37(13):393–398. Cited 2 times on pages 127 and 128.
- Veronese, G., Germini, F., Ingrassia, S., Cutuli, O., Donati, V., Bonacchini, L., Marcucci, M., Fabbri, A., and Italian Society of Emergency Medicine (SIMEU) (2016). Emergency physician accuracy in interpreting electrocardiograms with potential ST-segment elevation myocardial infarction: Is it enough? *Acute Cardiac Care*, 18(1):7–10. Cited 2 times on pages 23 and 132.
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . . (2020). SciPy 1.0—Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. Cited 2 times on pages 27 and 92.
- Vorontsov, E., Trabelsi, C., Kadoury, S., and Pal, C. (2017). On orthogonality and learning recurrent networks with long term dependencies. *arXiv:1702.00071*. Cited on page 101.
- Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Trans. Acoust., Speech, Signal Process.*, 37(3):328–339. Cited on page 124.
- Wang, N., Er, M. J., and Han, M. (2014). Parsimonious Extreme Learning Machine Using Recursive Orthogonal Least Squares. *IEEE Transactions on Neural Networks and Learning Systems*, 25(10):1828–1841. 00083. Cited on page 69.
- Wang, N., Sun, J.-C., Er, M. J., and Liu, Y.-C. (2016). Hybrid recursive least squares algorithm for online sequential identification using data chunks. *Neurocomputing*, 174:651–660. 00014. Cited on page 69.

- Warwick, K. and Craddock, R. (1996). An introduction to radial basis functions for system identification. A comparison with other neural network methods. In *Decision and Control, 1996., Proceedings of the 35th IEEE Conference On*, volume 1, pages 464–469. IEEE. 00038. Cited 3 times on pages 69, 70, and 84.
- Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. Technical report, Stanford Univ Ca Stanford Electronics Labs. Cited on page 47.
- Wigren, T. and Schoukens, J. (2013). Three free data sets for development and benchmarking in nonlinear system identification. In *2013 European Control Conference (ECC)*. Cited 2 times on pages 120 and 126.
- Willems, J. L., Abreu-Lima, C., Arnaud, P., van Bommel, J. H., Brohet, C., Degani, R., Denis, B., Gehring, J., Graham, I., and van Herpen, G. (1991). The diagnostic performance of computer programs for the interpretation of electrocardiograms. *The New England Journal of Medicine*, 325(25):1767–1773. Cited 2 times on pages 132 and 142.
- Willems, J. L., Abreu-Lima, C., Arnaud, P., van Bommel, J. H., Brohet, C., Degani, R., Denis, B., Graham, I., van Herpen, G., and Macfarlane, P. W. (1987). Testing the performance of ECG computer programs: The CSE diagnostic pilot study. *Journal of Electrocardiology*, 20 Suppl:73–77. Cited on page 132.
- Williams, R. J. and Zipser, D. (1989). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111. 00369. Cited 2 times on pages 73 and 108.
- Wills, A. and Ninness, B. (2012). Generalised Hammerstein–Wiener system estimation and a benchmark application. *Control Engineering Practice*, 20(11):1097–1108. Cited on page 124.
- Wisdom, S., Powers, T., Hershey, J., Le Roux, J., and Atlas, L. (2016). Full-Capacity Unitary Recurrent Neural Networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 4880–4888. Curran Associates, Inc. Cited on page 101.
- World Health Organization (2014). *Global Status Report on Noncommunicable Diseases 2014: Attaining the Nine Global Noncommunicable Diseases Targets; a Shared Responsibility*. World Health Organization, Geneva. OCLC: 907517003. Cited 2 times on pages 23 and 132.
- Wray, J. and Green, G. G. R. (1994). Calculation of the Volterra kernels of non-linear dynamic systems using an artificial neural network. *Biological Cybernetics*, 71(3):187–195. Cited on page 124.
- Xia, F., Yang, L. T., Wang, L., and Vinel, A. (2012). Internet of things. *International journal of communication systems*, 25(9):1101. Cited on page 21.
- Zhang, C., Li, K., Yang, Z., Pei, L., and Zhu, C. (2014). A new battery modelling method based on simulation error minimization. In *2014 IEEE PES General Meeting/ Conference & Exposition*. 00003. Cited 3 times on pages 71, 82, and 86.
- Zhang, D.-y., Sun, L.-p., and Cao, J. (2006). Modeling of temperature-humidity for wood drying based on time-delay neural network. *Journal of Forestry Research*, 17(2):141–144. 00028. Cited 3 times on pages 69, 70, and 84.
- Zhang, J., Lei, Q., and Dhillon, I. S. (2018). Stabilizing Gradients for Deep Neural Networks via Efficient SVD Parameterization. *Proceedings of the 35 th International Conference on Machine Learning*, page 9. Cited on page 101.

- Zhao, J., Zhu, Y., and Patwardhan, R. (2014). Identification of k-step-ahead prediction error model and MPC control. *Journal of Process Control*, 24(1):48–56. Cited 2 times on pages [91](#) and [92](#).