NON-UNIFORMLY SAMPLED SEQUENTIAL DATA PROCESSING

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL AND ELECTRONICS ENGINEERING

By

Safa Onur Şahin

September 2019

Non-Uniformly Sampled Sequential Data Processing
By Safa Onur Şahin
September 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Süleyman Serdar Kozat(Advisor)

Sinan Gezici

Çağatay Candan

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

NON-UNIFORMLY SAMPLED SEQUENTIAL DATA PROCESSING

Safa Onur Şahin
M.S. in Electrical and Electronics Engineering
Advisor: Süleyman Serdar Kozat
September 2019

We study classification and regression for variable length sequential data, which is either non-uniformly sampled or contains missing samples. In most sequential data processing studies, one considers data sequence is uniformly sampled and complete, i.e., does not contain missing input values. However, non-uniformly sampled sequences and the missing data problem appear in a wide range of fields such as medical imaging and financial data. To resolve these problems, certain preprocessing techniques, statistical assumptions and imputation methods are usually employed. However, these approaches suffer since the statistical assumptions do not hold in general and the imputation of artificially generated and unrelated inputs deteriorate the model. To mitigate these problems, in chapter 2, we introduce a novel Long Short-Term Memory (LSTM) architecture. In particular, we extend the classical LSTM network with additional time gates, which incorporate the time information as a nonlinear scaling factor on the conventional gates. We also provide forward pass and backward pass update equations for the proposed LSTM architecture. We show that our approach is superior to the classical LSTM architecture, when there is correlation between time samples. In chapter 3, we investigate regression for variable length sequential data containing missing samples and introduce a novel tree architecture based on the Long Short-Term Memory (LSTM) networks. In our architecture, we employ a variable number of LSTM networks, which use only the existing inputs in the sequence, in a tree-like architecture without any statistical assumptions or imputations on the missing data. In particular, we incorporate the missingness information by selecting a subset of these LSTM networks based on presence-pattern of a certain number of previous inputs.

Keywords: Long Short-Term Memory, Recurrent Neural Networks, Non-uniform Sampling, Missing Data, Supervised Learning.

ÖZET

DÜZGÜN OLMAYAN ŞEKİLDE ÖRNEKLENMİŞ SIRALI VERİNİN İŞLENMESİ

Safa Onur Şahin
Elektrik - Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: Süleyman Serdar Kozat
Eylül 2019

Bu tezde, düzgün olmayan bir şekilde örneklenmiş veya eksik örnek içeren değişken uzunluktaki sıralı veri kümelerinin sınıflandırılması ve bağlanımı üzerinde çalışılmıştır. Sıralı veri işleme çalışmalarında veri genellikle düzgün olarak örneklenmiş ve eksiksiz olarak kabul edilir. Ancak, medikal görüntüleme ve finansal tahmin uygulamalarının da içerisinde bulunduğu bir çok gerçek hayat uygulamasında düzgün örneklenmemiş veya eksik veri ile karşılaşılmaktadır. Bu problemlere çözüm olarak, belirli ön-işleme teknikleri, istatiksel varsayımlar ve yerine koyma metotları kullanılmaktadır. Ancak, bu modeller veriyi her zaman tam olarak modelleyememektedir. Bu nedenle yüksek performans artışları gözlenememektedir. Bu problemlere çözüm olarak, ikinci bölümde, yeni ve özgün bir Uzun Kısa-Soluklu Bellek (UKSB) sinir ağı mimarisi sunulmaktadır. Bu mimaride, geleneksel UKSB sinir ağı mimarisi zaman kapılarıyla genişletilmiş ve zaman bilgisini doğrusal olmayan bir ölçekleme faktörü olarak kullanacak şekilde gelişmiştir. Ayrıca önerilen LSTM mimarisi için ileri geçiş ve geri geçiş güncelleme denklemleri de sunulmaktadır. Yaklaşımımızın, zaman örnekleri arasında bir ilişki olduğu zaman, klasik LSTM mimarisine üstün olduğunu gösteriyoruz. Üçüncü bölümde ise eksik örnekleri içeren değişken uzunluklu sıralı verilerin bağlanımı çalışıldı ve Uzun Kısa-Soluklu Bellek (UKSB) sinir ağlarına dayanan yeni bir ağaç mimarisi tanıtıldı. Bu mimaride, sadece dizideki mevcut girdileri kullanan, ağaç benzeri bir mimaride, eksik verilerle ilgili herhangi bir istatistiksel varsayım yapmadan, değişken sayıda LSTM ağı kullanıyoruz. Burada, belirli sayıdaki geçmiş girdinin mevcudiyet modeline dayanarak kullanılabilecek UKSB ağlarını belirleyip bu ağların tahminlerini uyarlanabilir bir şekilde birleştiriyoruz.

Anahtar sözcükler: Uzun Kısa-Soluklu Bellek, Yineleyici Sinir Ağları, Düzgün Olmayan Örnekleme, Eksik Veri, Denetimli Öğrenme.

Acknowledgement

I would like to express my sincere gratitude to my advisor, Prof. Süleyman Serdar Kozat, for his magnificent guidance and motivation throughout my M.S. study.

I would like to thank TÜBİTAK for supporting me through BİDEB 2210-A Scholarship Program.

I would like to thank my parents for supporting me throughout my life.

I would like to thank my wife, Çağla, for her support and belief in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We study classification and regression of non-uniformly sampled variable length data sequences, where we sequentially receive a non-uniformly sampled data sequence or a sequence containing missing samples and then, estimate an unknown desired signal related to this sequence. In the classical data processing applications, data sequences are usually assumed to be uniformly sampled, however, this is not the case in many real life applications. For example, non-uniform sampling is used in many medical imaging applications [1], measurements in astronomy due to day and night conditions [2] and financial data [3], where the stock market values are re-determined by each transaction. Although non-uniformly sampled data frequently arises in these problems, there exist a few studies on non-uniformly sampled sequential data processing in neural networks [4], [5], machine learning [6] and signal processing literatures [7], [8]. Nonlinear approaches are usually used in these studies since linear approaches are usually incapable of capturing highly complex underlying structures [9]. Here, we study classification and regression problems particularly for non-uniformly sampled variable length data sequences in a supervised framework. These data sequences are either *i*) originally non-uniformly sampled or *ii*) originally uniformly sampled, however, they contain missing samples. We sequentially receive a data sequence with the corresponding desired data or signal and we find a nonlinear relation between them.

Even though there exist several nonlinear modeling approaches to process the sequential data [10], [9], neural network based methods are more practical in general because of their capability of modeling highly nonlinear and complex underlying relations [11]. Especially, recurrent neural networks (RNNs) are employed to process sequential data since they are able to identify sequential patterns and learn temporal behaviour, thanks to their internal memory exploiting past information. Although simple RNNs improve the performance in sequential processing tasks, they fail to capture long term dependencies due to vanishing and exploding gradient problems [12]. The LSTM networks are introduced as a special class of the RNNs to remedy these vanishing and exploding gradient problems and capture the long term dependencies [12]. The LSTM networks provide performance gains with their gating mechanisms, which control the amount of the information entering the network and the past information stored in the memory [11].

Even though the classical LSTM networks have satisfactory performance in the applications using uniformly sampled sequential data, they usually perform poorly in the case of non-uniformly sampled data [4], [5]. Among few proposed solutions for processing non-uniformly sampled sequential data, [4] and [13] first convert non-uniformly sampled data to uniformly sampled data by windowing and averaging the samples on large intervals. In the case that no sample exists in that interval, they impute these missing values by forward- and back-filling techniques, i.e., they replace the missing inputs with the next and previous values, respectively. Then, they process this uniformly sampled data using the LSTM network. In this setup, they still feed the LSTM network with uniformly sampled data, which is obtained after preprocessing the non-uniformly sampled data. These windowing and averaging operations cause information loss in data entering the LSTM networks. As an example, this preprocessing may cause failure in the corresponding tasks, where the aim is to detect whether a value is greater than a certain threshold or not, since averaging smooths the peaks. Furthermore, they also lose the time information contained in the sampling intervals instead of incorporating this information to the network. In [5], the authors provide a new LSTM architecture, namely, the PLSTM architecture, which basically learns a periodic sampler function and responds to only a small portion of the input sequence, which is sampled by this function. The sampler function is described by three parameters: period, shift and on-ratio. In each period, the network is updated by only the samples corresponding to its open phase, where on-ratio is ratio of open phase to the period, and shift is the initial time of the open phase. Processing only a small portion of the data accelerates the learning process and provides capability to work on non-uniformly sampled data by incorporating the time information. Although the PLSTM network performs better compared to the classical LSTM network in the classification tasks using non-uniformly sampled data, the model has two drawbacks. Firstly, an important amount of information is lost since the PLSTM architecture processes only a small percentage of the data sequence corresponding to its open phase. Secondly, the PLSTM network generates the output only at the end of the sequence, therefore, in the vanilla form, it can only be used for the sequential data processing tasks requiring only one output for the whole sequence.

In chapter 2, we resolve these problems by introducing a sequential nonlinear learning algorithm based on a novel LSTM network, which is extended with additional time gates, for classification and sequential regression tasks while keeping the computational load in the same level. These time gates incorporate the time information into the network as an adaptive nonlinear scaling factor on the conventional gates. Since we use the whole data sequence there is no loss in the incoming information unlike [4], [13], [5]. Moreover, our LSTM architecture can generate output at each time step unlike PLSTM, hence, it has a wide range of application areas from sequence labelling to online regression.

The main contributions of Chapter 2 are as follows. 1) We introduce a novel LSTM network architecture for processing non-uniformly sampled sequential data, where for the first time in the literature we incorporate the time information as a nonlinear scaling factor using additional time gates. 2) We show that the sampling intervals have a scaling effect on the conventional gates of the classical LSTM architecture. To show this, we first model non-uniform sampling with the missing input case and then extend it to the arbitrary non-uniform sampling case. 3) Our architecture can generate output at each time step as well as at the end of the input sequence unlike the PLSTM network. Therefore, our

LSTM architecture has a wide range of application areas from online regression to sequence labelling. 4) Our architecture contains the classical LSTM network and simplifies to it when the time intervals do not carry any information related to the underlying model. 5) Our LSTM architecture enables us to use the whole data sequence without any loss in the information entering to the LSTM network unlike [4], [13] and [5]. 6) We achieve this substantial performance improvement with the same order of computational complexity with the vanilla LSTM network. The computational cost due to the time gates is only linear in the number of hidden neurons in the LSTM network. 7) Through extensive set of experiments involving synthetic and real datasets, we demonstrate significant performance gains achieved by our algorithm for both regression and classification problems.

In chapter 3, we investigate regression for variable length sequential data containing missing samples as a special case of non-uniformly sampled data and introduce a novel tree architecture based on the LSTM networks. We resolve the problems due to missing data by introducing a sequential nonlinear learning algorithm based on the LSTM networks, where the outputs of a variable number of LSTM networks are adaptively combined in a tree-like architecture. Particularly, our architecture grants each LSTM network the capability of modelling an input sequence with a specific missingness pattern and learns to combine the outputs of these LSTM networks. By this way, the proposed algorithm incorporates the missingness information by selecting the particular LSTM networks based on the existence of the certain input patterns. Hence, it exploits both the input signal itself as well as the missingness pattern to mitigate the effects of the missing samples without making any statistical or artificial assumptions on the underlying data. In addition, our architecture keeps the computational load in terms of number of multiplication operations less than the computational load of the conventional algorithms especially when the number of missing samples is high.

Notation: In the thesis, all vectors are column vectors and denoted by boldface lower case letters. For a vector $\boldsymbol{x}$, $||\boldsymbol{x}|| = \sqrt{\boldsymbol{x}^T\boldsymbol{x}}$ is the ℓ^2-norm, where $\boldsymbol{x}^T$ is the ordinary transpose. $\langle \cdot, \cdot \rangle$ represents the outer product of two vectors, i.e., $\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle = \boldsymbol{x}_1\boldsymbol{x}_2^T$. Vector sequences are denoted by boldface upper case letters, e.g., $\boldsymbol{X}$. $\boldsymbol{X}^{(i)}$ represents the i^{th} vector sequence in the dataset $\{\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(N)}\}$, where N is the number of vector sequences in the set. $\mathcal{X}$ is the space of variable length vector sequences, i.e., $\boldsymbol{X}^{(i)} \in \mathcal{X}$. $\boldsymbol{X}^{(i)} = [\boldsymbol{x}_{t_1}^{(i)}, ..., \boldsymbol{x}_{t_{n_i}}^{(i)}]$ are the ordered sequence of vectors with length n_i, where $\boldsymbol{x}_{t_k}^{(i)}$ stands for the vector of $\boldsymbol{X}^{(i)}$ at time t_k, and k is the time index. x_j and $x_{t_k,j}$ represent the j^{th} elements in the vector $\boldsymbol{x}$ and $\boldsymbol{x}_{t_k}$, respectively. $\mathbf{1}_n \in \mathbb{R}^n$ stands for the vector, where all elements equal to 1. $W_{i,\{j,k\}}$ represents the element of the matrix $\boldsymbol{W}_i$ in j^{th} row and k^{th} column.

Chapter 2

Non-Uniformly Sampled Data Processing Using LSTM Networks

2.1 Problem Description

In this chapter, we study nonlinear regression and classification of non-uniformly sampled sequential data. We observe variable length vector sequences $\boldsymbol{X}^{(i)} = [\boldsymbol{x}_{t_1}^{(i)}, ..., \boldsymbol{x}_{t_{n_i}}^{(i)}] \in \mathcal{X}$, $\boldsymbol{x}_{t_k}^{(i)} \in \mathbb{R}^m$. The corresponding desired signal is given by $d_{t_k}^{(i)} \in \mathbb{R}$ in regression and $d_{n_i}^{(i)} \in \{1, \dots, C\}$ for classification, where C is the number of classes. Our goal is to estimate $d_{t_k}^{(i)}$ by

$$\hat{d}_{t_k}^{(i)} = f_{t_k}(\boldsymbol{x}_{t_1}^{(i)}, \dots, \boldsymbol{x}_{t_k}^{(i)}),$$

where $f_{t_k}(\cdot)$ is a possibly time varying and adaptive nonlinear function at time step t_k. For the input vector $\boldsymbol{x}_{t_k}^{(i)}$, we suffer the loss $l(d_{t_k}^{(i)}, \hat{d}_{t_k}^{(i)})$ and the loss for the vector sequence $\boldsymbol{X}^{(i)}$ is the average of individual losses, which is denoted by $L^{(i)} = \frac{1}{n_i} \sum_{k=1}^{n_i} l(d_{t_k}^{(i)}, \hat{d}_{t_k}^{(i)})$. The total performance of the network is evaluated by the mean of the losses over all sequences:

$$L = \frac{1}{N}\sum_{i=1}^{N} L^{(i)}. \tag{2.1}$$

Since the data is non-uniformly sampled, the sampling times of the input vectors $\boldsymbol{x}_{t_k}$ are not regular, i.e., the time intervals between the consecutive input vectors, $\boldsymbol{x}_{t_k}$ and $\boldsymbol{x}_{t_{k+1}}$, may vary and we denote these sampling intervals by Δt_k's,

$$\Delta t_k \triangleq t_{k+1} - t_k.$$

As an example, in target tracking and position estimation application with a camera system [14], we sequentially receive position vectors of a target $\boldsymbol{x}_{t_k}$ and estimate its distance from a certain point $\boldsymbol{p}$ in the next position by $\hat{d}_{t_k}$. Here, the desired signal is given by $d_{t_k} = ||\boldsymbol{x}_{t_{k+1}} - \boldsymbol{p}||$ and under squared error loss, $l(d_{t_k}, \hat{d}_{t_k}) = (d_{t_k} - \hat{d}_{t_k})^2$. In the case of occlusions or when the camera misses frames, we do not receive position vectors and time intervals between consecutively received position vectors change, which corresponds to non-uniform sampling.

We use recurrent neural networks to process the sequential data. A generic RNN is given by [15]

$$\begin{aligned} \boldsymbol{h}_{t_k} &= f(\boldsymbol{W}_h \boldsymbol{x}_{t_k} + \boldsymbol{R}_h \boldsymbol{h}_{t_{k-1}}) \\ \boldsymbol{y}_{t_k} &= g(\boldsymbol{R}_y \boldsymbol{h}_{t_k}), \end{aligned} \tag{2.2}$$

where $\boldsymbol{x}_{t_k} \in \mathbb{R}^m$ is the input vector, $\boldsymbol{h}_{t_k} \in \mathbb{R}^q$ is the state vector and $\boldsymbol{y}_{t_k} \in \mathbb{R}^p$ is the output at time t_k. $\boldsymbol{W}_h \in \mathbb{R}^{q\times m}$, $\boldsymbol{R}_h \in \mathbb{R}^{q\times q}$ and $\boldsymbol{R}_h \in \mathbb{R}^{q\times q}$ are the input weight matrices. $f(\cdot)$ and $g(\cdot)$ are point-wise nonlinear functions. We drop the sample index i to simplify the notation.

We focus on a special kind of the RNNs, the LSTM networks without the peephole connections. The LSTM network is described by the following equations

[16]:

$$\boldsymbol{z}_{t_k} = g(\boldsymbol{W}_z \boldsymbol{x}_{t_k} + \boldsymbol{R}_z \boldsymbol{y}_{t_{k-1}}) \tag{2.3}$$
$$\boldsymbol{i}_{t_k} = \sigma(\boldsymbol{W}_i \boldsymbol{x}_{t_k} + \boldsymbol{R}_i \boldsymbol{y}_{t_{k-1}}) \tag{2.4}$$
$$\boldsymbol{f}_{t_k} = \sigma(\boldsymbol{W}_f \boldsymbol{x}_{t_k} + \boldsymbol{R}_f \boldsymbol{y}_{t_{k-1}}) \tag{2.5}$$
$$\boldsymbol{o}_{t_k} = \sigma(\boldsymbol{W}_o \boldsymbol{x}_{t_k} + \boldsymbol{R}_o \boldsymbol{y}_{t_{k-1}}) \tag{2.6}$$
$$\boldsymbol{c}_{t_k} = \boldsymbol{i}_{t_k} \odot \boldsymbol{z}_{t_k} + \boldsymbol{f}_{t_k} \odot \boldsymbol{c}_{t_{k-1}} \tag{2.7}$$
$$\boldsymbol{y}_{t_k} = \boldsymbol{o}_{t_k} \odot h(\boldsymbol{c}_{t_k}), \tag{2.8}$$

where $\boldsymbol{x}_{t_k} \in \mathbb{R}^m$ is the input vector, $\boldsymbol{c}_{t_k} \in \mathbb{R}^q$ is the state vector and $\boldsymbol{y}_{t_k} \in \mathbb{R}^q$ is the output vector at time t_k. $\boldsymbol{z}_{t_k}$ is the block input, $\boldsymbol{i}_{t_k}$, $\boldsymbol{f}_{t_k}$ and $\boldsymbol{o}_{t_k}$ are the input, forget and output gates, respectively. Nonlinear activation functions $g(\cdot)$, $h(\cdot)$ and $\sigma(\cdot)$ apply the point-wise operations. $\tanh(\cdot)$ is commonly used for $g(\cdot)$ and $h(\cdot)$ functions and $\sigma(\cdot)$ is the sigmoid function, i.e., $\sigma(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{x}}}$. $\odot$ is the element-wise (Hadamard) product and operates on the two vectors of the same size. $\boldsymbol{W}_z$, $\boldsymbol{W}_i$, $\boldsymbol{W}_f$, $\boldsymbol{W}_o \in \mathbb{R}^{q \times m}$ are the input weight matrices and $\boldsymbol{R}_z$, $\boldsymbol{R}_i$, $\boldsymbol{R}_f$, $\boldsymbol{R}_o \in \mathbb{R}^{q \times q}$ are the recurrent weight matrices. With the abuse of notation, we incorporate the bias weights, $\boldsymbol{b}_z$, $\boldsymbol{b}_i$, $\boldsymbol{b}_f$, $\boldsymbol{b}_o \in \mathbb{R}^q$, into the input weight matrices and denote them by $\boldsymbol{W}_\theta = [\boldsymbol{W}_\theta; \boldsymbol{b}_\theta]$, $\theta \in \{z, i, f, o\}$, where $\boldsymbol{x}_{t_k} = [\boldsymbol{x}_{t_k}; 1]$. For the regression problem, we generate the estimate $\hat{d}_{t_k}$ as

$$\hat{d}_{t_k} = \boldsymbol{w}_{t_k}^T \boldsymbol{y}_{t_k},$$

where $\boldsymbol{w}_{t_k} \in \mathbb{R}^q$ is the final regression coefficients, which can be trained in an online or batch manner depending on the application.

For the classification problem, we focus on the sequence classification, i.e., we have only one desired signal $d^{(i)}$ for each vector sequence $\boldsymbol{X}^{(i)}$. As shown in Fig. 2.1, our final decision $\hat{d}^{(i)}$ is given by

$$\hat{d}^{(i)} = \max_j \operatorname{softmax}(\boldsymbol{W} \tilde{\boldsymbol{y}}^{(i)})_j,$$

where $\boldsymbol{W} \in \mathbb{R}^{q \times c}$ is the weight matrix, c is the number of classes, and $\tilde{\boldsymbol{y}}^{(i)}$ is the combination of the LSTM network outputs, $\boldsymbol{y}_{t_1}^{(i)}, \ldots, \boldsymbol{y}_{t_{n_i}}^{(i)}$. To obtain $\tilde{\boldsymbol{y}}^{(i)}$, we may

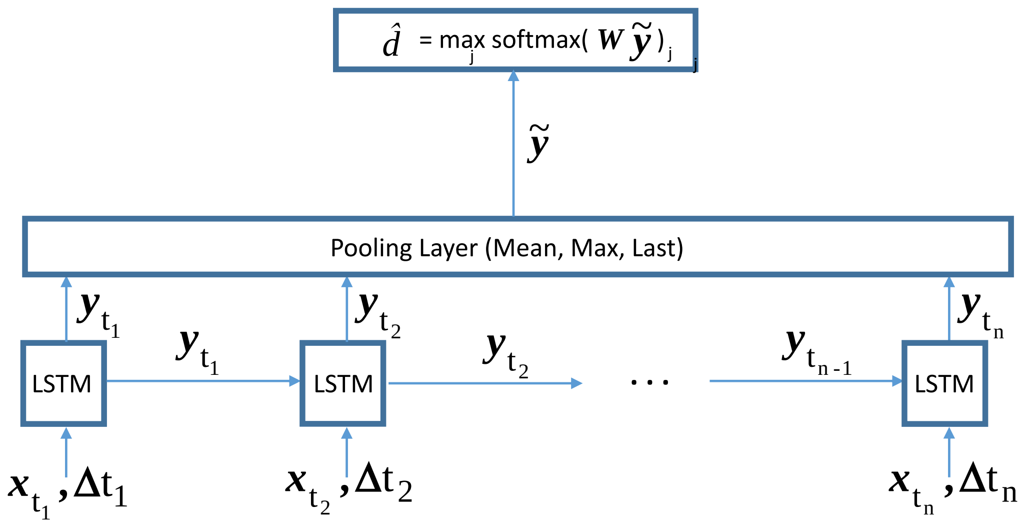


Figure 2.1: Detailed schematic of the classification architecture. Note that the index i is dropped in order to simplify the notation.

use three different pooling methods: mean, max and last pooling as

$$\begin{aligned}
\tilde{\boldsymbol{y}}^{(i)}_{mean} &= \frac{1}{n_i}\sum_{k=1}^{n_i} \boldsymbol{y}^{(i)}_{t_k} \\
\tilde{y}^{(i)}_{max_j} &= \max_k(y^{(i)}_{t_k,j}) \\
\tilde{\boldsymbol{y}}^{(i)}_{last} &= \boldsymbol{y}^{(i)}_{t_{n_i}}.
\end{aligned}$$

In Section III, we introduce a novel LSTM architecture working on non-uniformly sampled data, and also provide its forward-pass and backward-pass update formulas.

2.2 A Novel LSTM Architecture

We need to incorporate the time information into the LSTM network to enhance the performance [5]. For this purpose, one can directly append the sampling intervals, Δt_k's, to the input vector, i.e., $\tilde{\boldsymbol{x}}_{t_k} = [\boldsymbol{x}_{t_k}; \Delta t_k]$. However, in this solution, Δt_k is incorporated as an additional feature and its effect is only additive to the weighted sum of the other features, e.g., as multiplied by $\tilde{\boldsymbol{W}}\tilde{\boldsymbol{x}_{t_k}}$, where $\tilde{\boldsymbol{W}} \in \mathbb{R}^{q\times(m+1)}$ is the extended weight matrix. For example, the input gate $\boldsymbol{i}_{t_k}$ is calculated by

$$\boldsymbol{i}_{t_k} = \sigma(\tilde{\boldsymbol{W}}_i\tilde{\boldsymbol{x}}_{t_k} + \boldsymbol{R}_i\boldsymbol{y}_{t_{k-1}}), \tag{2.9}$$

instead of (4), where $\boldsymbol{W}_i\boldsymbol{x}_{t_k}$ term changes as $\tilde{\boldsymbol{W}}_i\tilde{\boldsymbol{x}}_{t_k}$. In that case, the only difference between (2.9) and (3.3) is the additive term of $\tilde{W}_{i,\{j,m+1\}}\Delta t_k$ inside $\sigma(\cdot)$. In the following, we will demonstrate that the Δt_k should also have a scaling effect on the conventional gates, i.e., the input, forget and output gates.

To this end, in subsection 2.2.1, we first consider a special case of non-uniform sampling, where $\boldsymbol{X}^{(i)}$ is uniformly sampled, however, certain columns of $\boldsymbol{X}^{(i)}$ are missing. We then extend our approach to arbitrary non-uniform sampling case in subsection 2.2.2.

2.2.1 Modeling Non-uniform Sampling with Missing Input Case

We make our derivations first for the RNN case for one step ahead prediction problem, i.e., the aim is to estimate the next signal $\boldsymbol{x}_{t_{k+1}}$, where the current input is $\boldsymbol{x}_{t_k}$. We first consider the case when we have uniform sampling, i.e., $t_{k+1} - t_k = \Delta$ for all time steps, where Δ is some fixed time interval. In this framework, we simply combine the RNN equations (3.1), then, the RNN model estimates the next sample as

$$\begin{aligned}\hat{\boldsymbol{x}}_{t_{k+1}} &= g\left(\boldsymbol{R}_y f\left(\boldsymbol{W}_h\boldsymbol{x}_{t_k} + \boldsymbol{R}_h\boldsymbol{h}_{t_{k-1}}\right)\right)\\ &= \bar{f}(\boldsymbol{x}_{t_k}, \boldsymbol{h}_{t_{k-1}}),\end{aligned} \tag{2.10}$$

where $\bar{f}(\cdot)$ is a composite function, which includes $f(\cdot)$ and $g(\cdot)$. Assume that $\boldsymbol{x}_{t_k}$ are the samples of an infinitely differentiable continuous function of time, $\boldsymbol{x}$.

In this case, $\boldsymbol{x}_{t_{k+1}}$ is calculated by the Taylor series expansion of $\boldsymbol{x}$ around $\boldsymbol{x}_{t_k}$ as

$$\begin{aligned}\boldsymbol{x}_{t_{k+1}} &= \boldsymbol{x}_{t_k+\Delta} \\ &= \boldsymbol{x}_{t_k} + \frac{\Delta}{1!}\frac{\partial \boldsymbol{x}_{t_k}}{\partial t} + \frac{\Delta^2}{2!}\frac{\partial^2 \boldsymbol{x}_{t_k}}{\partial t^2} + \frac{\Delta^3}{3!}\frac{\partial^3 \boldsymbol{x}_{t_k}}{\partial t^3} + \dots\end{aligned} \tag{2.11}$$

We now model the non-uniform sampling case with missing instances, i.e., any Δt_k is an integer multiple of the fixed time interval Δ. For example, if the next input $\boldsymbol{x}_{t_{k+1}}$ is not missing, then the time interval $\Delta t_k = \Delta$. Similarly, if $\boldsymbol{x}_{t_{k+1}}$ is missing, but we have $\boldsymbol{x}_{t_{k+2}}$, then $\Delta t_k = 2\Delta$. Assume that the $\boldsymbol{x}_{t_{k-1}}$ and $\boldsymbol{x}_{t_{k+1}}$ are available, while the $\boldsymbol{x}_{t_k}$ is missing from our data sequence. In this case, we cannot directly apply the same Taylor series expansion in (2.11) to calculate $\boldsymbol{x}_{t_{k+1}}$ since the data $\boldsymbol{x}_{t_k}$ is missing. However, we have an estimate $\hat{\boldsymbol{x}}_{t_k}$, which is obtained by the model in (2.10) using the input $\boldsymbol{x}_{t_{k-1}}$. Therefore, we estimate $\boldsymbol{x}_{t_{k+1}}$ by using $\hat{\boldsymbol{x}}_{t_k}$ instead of $\boldsymbol{x}_{t_k}$ in (2.11) as

$$\begin{aligned}\boldsymbol{x}_{t_{k+1}} &\approx \sum_{n=0}^{\infty} \frac{\Delta^n}{n!}\frac{\partial^n \hat{\boldsymbol{x}}_{t_k}}{\partial t^n} \\ &= \hat{\boldsymbol{x}}_{t_k} + \frac{\Delta}{1!}\frac{\partial \hat{\boldsymbol{x}}_{t_k}}{\partial t} + \frac{\Delta^2}{2!}\frac{\partial^2 \hat{\boldsymbol{x}}_{t_k}}{\partial t^2} + \frac{\Delta^3}{3!}\frac{\partial^3 \hat{\boldsymbol{x}}_{t_k}}{\partial t^3} + \dots\end{aligned} \tag{2.12}$$

We next substitute $\hat{\boldsymbol{x}}_{t_k}$ with $\bar{f}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}})$ by using (2.10) to yield

$$\begin{aligned}\hat{\boldsymbol{x}}_{t_{k+1}} = \bar{f}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}) + \frac{\Delta}{1!}\frac{\partial \bar{f}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}})}{\partial t} \\ + \frac{\Delta^2}{2!}\frac{\partial^2 \bar{f}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}})}{\partial t^2} + \dots\end{aligned} \tag{2.13}$$

We write (2.13) in the vector form as

$$\hat{x}_{t_{k+1},j} = \begin{bmatrix} 1 & \frac{\Delta}{1!} & \frac{\Delta^2}{2!} & \frac{\Delta^3}{3!} & \dots \end{bmatrix} \begin{bmatrix} \bar{f}\left(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}\right)_j \\ \bar{f}'\left(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}\right)_j \\ \bar{f}''\left(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}\right)_j \\ \bar{f}'''\left(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}\right)_j \\ \vdots \end{bmatrix}, \tag{2.14}$$

where $\bar{f}'(\cdot)$ represents the derivative with respect to t and similarly for the other derivative terms. We approximate this equation as

$$\hat{\boldsymbol{x}}_{t_{k+1}} \approx f_{\Delta}(\Delta) \odot f_{\boldsymbol{x},\boldsymbol{h}}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}), \tag{2.15}$$

where $f_\Delta(\cdot)$ is a nonlinear function of Δ, whereas $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$ represents a nonlinear function of $\boldsymbol{x}_{t_{k-1}}$ and $\boldsymbol{h}_{t_{k-2}}$. Note that both $f_\Delta(\cdot)$ and $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$ return vectors as their outputs in the length of $\boldsymbol{x}_{t_k}$. This derivation can be extended to any length of missing instances such as for 2Δ this yields

$$\hat{\boldsymbol{x}}_{t_{k+2}} \approx f_\Delta(2\Delta) \odot f_{\boldsymbol{x},\boldsymbol{h}}(\boldsymbol{x}_{t_{k-1}}, \boldsymbol{h}_{t_{k-2}}). \tag{2.16}$$

Hence, the time interval Δ has a nonlinear scaling effect on $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$. Note that in uniform sampling case, the classical RNNs use only $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$ to estimate the next sample, i.e., $\bar{f}(\cdot)$ in (2.10), although the scaling effect of time interval still exists. However, $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$ is able to handle this scaling effect since Δ's and $f_\Delta(\Delta)$ are constant.

In subsection 2.2.2, we focus on the arbitrary non-uniform sampling case.

2.2.2 Arbitrary Non-uniform Sampling

In this subsection, we consider the arbitrary non-uniform sampling, i.e., sampling without any constant sampling interval and Δt_k is not an integer multiple of a fixed time interval Δ. The Taylor series expansion for the missing data case is similarly extended to arbitrary non-uniform sampling case for the one step ahead estimation problem, i.e.,

$$\begin{aligned} \boldsymbol{x}_{t_{k+1}} &= \sum_{n=0}^{\infty} \frac{\Delta t_k^n}{n!} \frac{\partial^n \boldsymbol{x}_{t_k}}{\partial t^n} \\ &= \boldsymbol{x}_{t_k} + \frac{\Delta t_k}{1!} \frac{\partial \boldsymbol{x}_{t_k}}{\partial t} + \frac{\Delta t_k^2}{2!} \frac{\partial^2 \boldsymbol{x}_{t_k}}{\partial t^2} + \frac{\Delta t_k^3}{3!} \frac{\partial^3 \boldsymbol{x}_{t_k}}{\partial t^3} + \dots \end{aligned} \tag{2.17}$$

Similar derivations lead to

$$\hat{\boldsymbol{x}}_{t_{k+1}} = f_\Delta(\Delta t_k) \odot f_{\boldsymbol{x},\boldsymbol{h}}(\boldsymbol{x}_{t_k}, \boldsymbol{h}_{t_{k-1}}). \tag{2.18}$$

In the non-uniform sampling case, $f_\Delta(\Delta t_k)$ have unique scaling effect on $f_{\boldsymbol{x},\boldsymbol{h}}(\cdot)$ at each time step since Δt_k differs. Therefore, ignoring the time information in estimation process results in a limited performance. Extending input vector with

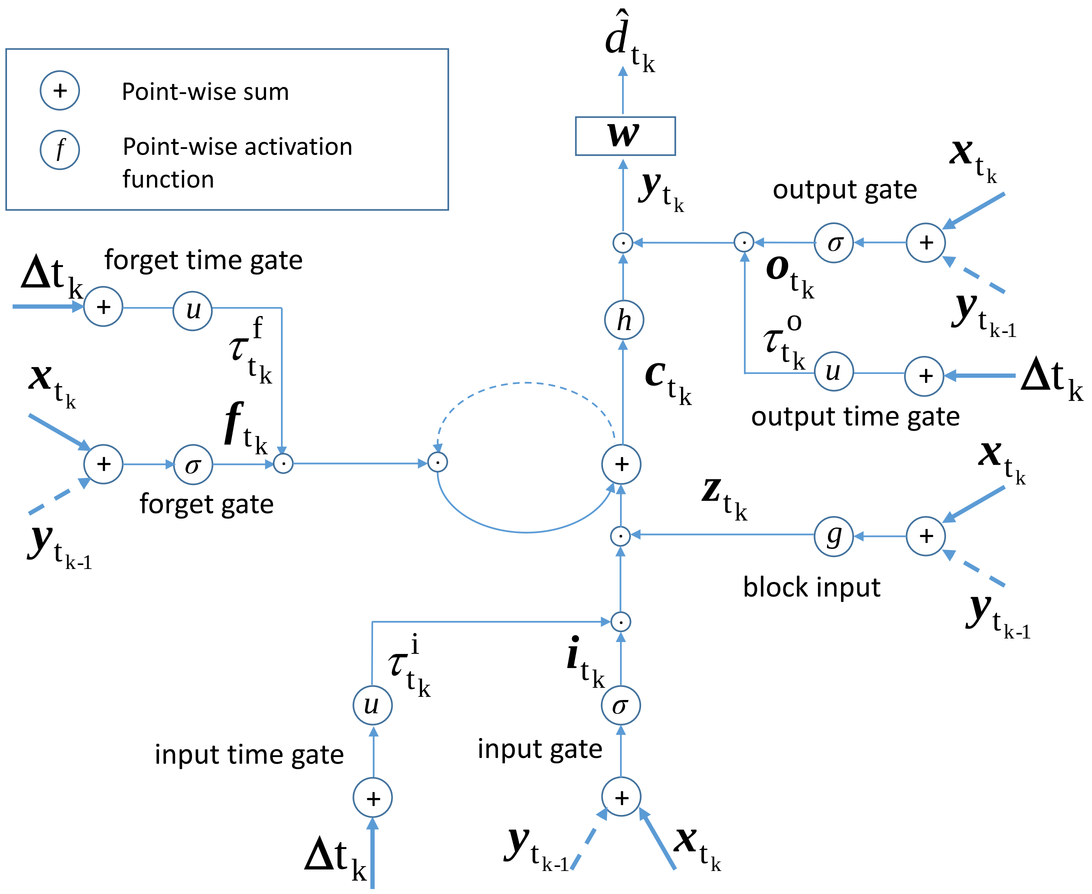


Figure 2.2: Detailed schematic of an LSTM block with additional time gates. Note that $\boldsymbol{x}_{t_k}$, $\boldsymbol{y}_{t_{k-1}}$ and $\boldsymbol{\Delta} t_k$ are multiplied with their weights, $\boldsymbol{W}_{(\cdot)}$ and $\boldsymbol{R}_{(\cdot)}$, according to (3.2)-(3.4) and (2.20)-(2.23). Also corresponding biases $\boldsymbol{b}_{(\cdot)}$ is added.

time intervals makes only an additive contribution to the $f_{\boldsymbol{x},\boldsymbol{h}}(\boldsymbol{x}_{t_k}, \boldsymbol{h}_{t_{k-1}})$ term, which is insufficient to model the effect of $f_\Delta(\Delta t_k)$. To circumvent this issue, we introduce a new RNN structure, particularly, an LSTM architecture, which includes the effect of $f_\Delta(\Delta t_k)$. The new LSTM architecture is explained in the next subsection.

2.2.3 Time Gated - LSTM Architecture

We present a novel LSTM architecture to incorporate the time information into our estimation function as a nonlinear scaling factor, i.e., it learns the time dependent scaling function $f_\Delta(\cdot)$. In the classical LSTM architecture, $f_{\boldsymbol{x},\boldsymbol{h}}(\boldsymbol{x}_{t_k}, \boldsymbol{h}_{t_{k-1}})$

is already modelled as $\sigma(\boldsymbol{W}\boldsymbol{x}_{t_k} + \boldsymbol{R}\boldsymbol{y}_{t_{k-1}})$ in the specialized gate structures as in (3.2)-(3.5). Therefore, we focus on modeling $f_{\Delta}(\cdot)$. In accordance with (2.18), we can straightforwardly incorporate the time information into the LSTM architecture by altering (2.8) as

$$\boldsymbol{y}_{t_k} = \boldsymbol{o}_{t_k} \odot h(\boldsymbol{c}_{t_k}) \odot f_{\Delta}(\Delta t_k). \tag{2.19}$$

Here, we incorporate $f_{\Delta}(\Delta t_k)$ to the LSTM architecture as a scaling factor only on the output gate. Since the gate structures in the LSTM architecture are specialized for different tasks, such as forgetting the last state, their responses to the time intervals need to be different. For example, when the input $\boldsymbol{x}_{t_k}$ arrives after a long time interval Δt_k, while the forget gate needs to keep a small amount of the past state, the input gate needs to incorporate more from the new input. To this end, we decompose $f_{\Delta}(\Delta t_k)$ into three different functions, $f_{\Delta}^{(i)}(\Delta t_k)$, $f_{\Delta}^{(f)}(\Delta t_k)$ and $f_{\Delta}^{(o)}(\Delta t_k)$, and use these functions on the conventional gates in order to allow them to give different responses depending on the time intervals. In particular, we introduce new time gates to the LSTM network in order to model the scaling effect of $f_{\Delta}(\cdot)$. This LSTM architecture is named as Time Gated LSTM (TG-LSTM) in this paper.

We introduce three different time gates, which use sampling intervals, Δt_k's, as their inputs as shown in Fig. 3.2. The first time gate is the input time gate and denoted by $\boldsymbol{\tau}_{t_k}^{i}$, the second time gate is the forget time gate and represented by $\boldsymbol{\tau}_{t_k}^{f}$. Similarly, the third time gate is the output time gate and denoted by $\boldsymbol{\tau}_{t_k}^{o}$. Note that there is no time gate $\boldsymbol{\tau}_{t_k}^{z}$, since $\boldsymbol{i}_{t_k}$ and $\boldsymbol{z}_{t_k}$ participate to the network as multiplied with each other and only one time gate $\boldsymbol{\tau}_{t_k}^{i}$ is sufficient to scale both. The input gate $\boldsymbol{i}_{t_k}$, forget gate $\boldsymbol{f}_{t_k}$ and output gate $\boldsymbol{o}_{t_k}$ are multiplied by $\boldsymbol{\tau}_{t_k}^{i}$, $\boldsymbol{\tau}_{t_k}^{f}$, $\boldsymbol{\tau}_{t_k}^{o}$ respectively as shown in Fig. 3.2. In addition to (3)-(8), the forward-pass process of the new LSTM architecture in Fig. 3.2 is modelled by the following set of equations:

$$
\begin{aligned}
\boldsymbol{\tau}^i_{t_k} &= u(\boldsymbol{W}_{\tau^i}\boldsymbol{\Delta}t_k) && (2.20)\\
\boldsymbol{\tau}^f_{t_k} &= u(\boldsymbol{W}_{\tau^f}\boldsymbol{\Delta}t_k) && (2.21)\\
\boldsymbol{c}_{t_k} &= \boldsymbol{i}_{t_k} \odot \boldsymbol{z}_{t_k} \odot \boldsymbol{\tau}^i_{t_k} + \boldsymbol{f}_{t_k} \odot \boldsymbol{c}_{t_{k-1}} \odot \boldsymbol{\tau}^f_{t_k} && (2.22)\\
\boldsymbol{\tau}^o_{t_k} &= u(\boldsymbol{W}_{\tau^o}\boldsymbol{\Delta}t_k) && (2.23)\\
\boldsymbol{y}_{t_k} &= \boldsymbol{o}_{t_k} \odot \boldsymbol{\tau}^o_{t_k} \odot h(\boldsymbol{c}_{t_k}), && (2.24)
\end{aligned}
$$

where $\boldsymbol{W}_{\tau^i}$, $\boldsymbol{W}_{\tau^f}$ and $\boldsymbol{W}_{\tau^o} \in \mathbb{R}^{q\times n_\tau}$ are the weight matrices of the time gates $\boldsymbol{\tau}^i$, $\boldsymbol{\tau}^f$ and $\boldsymbol{\tau}^o$, respectively. $u(\cdot)$ is the point-wise nonlinearity, which is set to $\sigma(\cdot)$. $\boldsymbol{\Delta}t_k \in \mathbb{R}^{n_\tau}$ is the input for the time gates and one can append different functions of Δt_k such as $(\Delta t_k)^2$ and $\frac{1}{\Delta t_k}$ in addition to Δt_k. Here, (2.20), (2.21) and (2.23) are added to the set of forward-pass equations of the classical LSTM architecture, (2.22) and (2.24) are replaced with (3.6) and (2.8), respectively.

2.2.4 Training of the New Model

For the training of the TG-LSTM architecture, we employ the back-propagation through time (BPTT) algorithm to update the weight matrices of our LSTM network, i.e., the input weight matrices $\boldsymbol{W}_z, \boldsymbol{W}_i, \boldsymbol{W}_f, \boldsymbol{W}_o, \boldsymbol{W}_{\tau^i}, \boldsymbol{W}_{\tau^f}, \boldsymbol{W}_{\tau^o}$, and the recurrent weight matrices $\boldsymbol{R}_z, \boldsymbol{R}_i, \boldsymbol{R}_f, \boldsymbol{R}_o$. To write the update equations in a notationally simplified form, we first define a new notation for the gates before the nonlinearity is applied, e.g.,

$$
\begin{aligned}
\bar{\boldsymbol{i}}_{t_k} &= \boldsymbol{W}_i\boldsymbol{x}_{t_k} + \boldsymbol{R}_i\boldsymbol{y}_{t_{k-1}}\\
\bar{\boldsymbol{\tau}}^i_{t_k} &= \boldsymbol{W}_{\tau^i}\boldsymbol{x}_{t_k},
\end{aligned}
$$

where $\bar{\boldsymbol{i}}_{t_k} \in \mathbb{R}^q$ and $\bar{\boldsymbol{\tau}}^i_{t_k} \in \mathbb{R}^q$ are the sum terms before the nonlinearity for the input gate and input time gate, respectively. The terms for the other gates $\bar{\boldsymbol{z}}_{t_k}$, $\bar{\boldsymbol{f}}_{t_k}$, $\bar{\boldsymbol{o}}_{t_k}$, $\bar{\boldsymbol{\tau}}^f_{t_k}$, $\bar{\boldsymbol{\tau}}^o_{t_k} \in \mathbb{R}^q$ have similar formulations. Then, we first calculate the local gradients as follows:

Architecture	Computational Load
LSTM-1	$4q^2 + 4qm + 3q$
LSTM-2	$4q^2 + 4qm + 7q$
PLSTM	$4q^2 + 4qm + 3q$
TG-LSTM	$4q^2 + 4qm + 6q$

Table 2.1: The number of multiplication operations in the forward pass of the TG-LSTM, PLSTM and the classical LSTM architectures for one time step. LSTM-1 is the network that time intervals are not used. LSTM-2 represents the LSTM network, where the time intervals are added to the input vector as another feature.

$$
\begin{aligned}
\delta \boldsymbol{y}_{t_k} &= \frac{\partial L}{\partial y_{t_k}} + \boldsymbol{R}_z^T \delta \boldsymbol{z}_{t_{k+1}} + \boldsymbol{R}_i^T \delta \boldsymbol{i}_{t_{k+1}} \\
&\quad + \boldsymbol{R}_f^T \delta \boldsymbol{f}_{t_{k+1}} + \boldsymbol{R}_o^T \delta \boldsymbol{o}_{t_{k+1}} \\
\delta \boldsymbol{o}_{t_k} &= \delta \boldsymbol{y}_{t_k} \odot h(\boldsymbol{c}_{t_k}) \odot \boldsymbol{\tau}^o_{t_k} \odot \sigma'(\bar{\boldsymbol{o}}_{t_k}) \\
\delta \boldsymbol{\tau}^o_{t_k} &= \delta \boldsymbol{y}_{t_k} \odot h(\boldsymbol{c}_{t_k}) \odot \boldsymbol{o}_{t_k} \odot u'(\bar{\boldsymbol{\tau}}^o{}_{t_k}) \\
\delta \boldsymbol{c}_{t_k} &= \delta \boldsymbol{y}_{t_k} \odot \boldsymbol{o}_{t_k} \odot \boldsymbol{\tau}^o_{t_k} \odot h'(\boldsymbol{c}_{t_k}) + \boldsymbol{f}_{t_{k+1}} \odot \delta \boldsymbol{c}_{t_{k+1}} \\
\delta \boldsymbol{f}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{c}_{t_{k-1}} \odot \boldsymbol{\tau}^f_{t_k} \odot \sigma'(\bar{\boldsymbol{f}}_{t_k}) \\
\delta \boldsymbol{\tau}^f_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{c}_{t_{k-1}} \odot \boldsymbol{f}_{t_k} \odot \sigma'(\bar{\boldsymbol{\tau}}^f{}_{t_k}) \\
\delta \boldsymbol{i}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{z}_{t_k} \odot \boldsymbol{\tau}^i_{t_k} \odot \sigma'(\bar{\boldsymbol{i}}_{t_k}) \\
\delta \boldsymbol{z}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{i}_{t_k} \odot \boldsymbol{\tau}^i_{t_k} \odot g'(\bar{\boldsymbol{z}}_{t_k}) \\
\delta \boldsymbol{\tau}^i_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{i}_{t_k} \odot \boldsymbol{z}_{t_k} \odot u'(\bar{\boldsymbol{\tau}}^i_{t_k}),
\end{aligned}
$$

where $\delta \boldsymbol{y}_{t_k}$, $\delta \boldsymbol{o}_{t_k}$, $\delta \boldsymbol{\tau}^o_{t_k}$, $\delta \boldsymbol{c}_{t_k}$, $\delta \boldsymbol{f}_{t_k}$, $\delta \boldsymbol{\tau}^f_{t_k}$ $\delta \boldsymbol{c}_{t_k}$, $\delta \boldsymbol{i}_{t_k}$, $\delta \boldsymbol{z}_{t_k}$, $\delta \boldsymbol{\tau}^i_{t_k} \in \mathbb{R}^q$ are the local gradients for corresponding nodes. The gradients for the input and the recurrent weight matrices are calculated by

$$
\begin{aligned}
\delta \boldsymbol{W}_\theta &= \sum_{k=0}^{n} \langle \delta \boldsymbol{\theta}_{t_k}, \boldsymbol{x}_{t_k} \rangle \\
\delta \boldsymbol{R}_\theta &= \sum_{k=0}^{n-1} \langle \delta \boldsymbol{\theta}_{t_{k+1}}, \boldsymbol{y}_{t_k} \rangle,
\end{aligned}
$$

where $\theta \in \{z, i, f, o\}$, and the gradient for weights of the time gates are calculated by

$$\delta \boldsymbol{W}_{\tau^*} = \sum_{k=0}^{n} \langle \delta \boldsymbol{\tau}^*_{t_k}, \boldsymbol{\Delta} t_k \rangle,$$

where $* \in \{i, f, o\}$ and $\boldsymbol{\Delta} t_k = [\Delta t_k; 1]$. $\langle \cdot, \cdot \rangle$ represents the outer product of two vectors, i.e., $\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle = \boldsymbol{x}_1 \boldsymbol{x}_2^T$.

Remark 1 *Our TG-LSTM architecture has additional time gates on top of the vanilla LSTM architecture. One can remove any time gate by setting its all elements to 1, for example, to close input time gate, $\boldsymbol{\tau}^i = \mathbf{1}_q$. In the worst case, the time intervals have no correlation with the underlying model, all time gates converge to $\mathbf{1}_q$ vector and our TG-LSTM architecture simplifies to the classical LSTM architecture.*

Remark 2 *The complexity of the new architecture is in the same order of the complexity of the classical LSTM architecture. In Table 3.1, we provide the computational loads in terms of the number of required multiplication operations in the forward pass for the classical LSTM, PLSTM and TG-LSTM architectures. In the table, LSTM-1 represents the LSTM network in which the time intervals are not incorporated to the input vector, i.e., the input vector is merely $\boldsymbol{x}_{t_k}$. LSTM-2 is the network in which the input vectors are extended with the time intervals between the samples as another feature, i.e., $\tilde{\boldsymbol{x}}_{t_k} = [\boldsymbol{x}_{t_k}; \Delta t_k]$. Four matrix vector multiplications for the input, i.e., $\boldsymbol{W}\boldsymbol{x}_{t_k}$, four matrix vector multiplications for the last hidden state, i.e., $\boldsymbol{R}\boldsymbol{h}_{t_{k-1}}$, and three vector-vector multiplications between gates, i.e., (3.6) and (2.8) in the response of the previous comment, are included in the basic LSTM architecture, which need $4q^2+4qm+3q$ multiplication operations. Since the LSTM-2 architecture has an extended input vector, it has $4q(m+1)$ multiplications instead of $4qm$ from the $\boldsymbol{W}\boldsymbol{x}_{t_k}$ terms. The PLSTM architecture has additional scalar operations for the sampler functions, however, since we include only vectorial multiplications, it has $4q^2 + 4qm + 3q$ multiplication operations in one time step. The TG-LSTM architecture has additional $3q$ multiplications due to the multiplications of time gates with the conventional gates.*

2.3 Simulations

In this section, we illustrate the performance of the proposed LSTM architecture under different scenarios with respect to the state of the art methods through several experiments. In the first part, we focus on the regression problem for various real life datasets such as kinematic [17], bank [18] and pumadyn [19]. In the second part, we compare our method with the LSTM structures on several different classification tasks over real life datasets such as Pen-Based Recognition of Handwritten Digits [20] and UJI Pen (Version 2) [20] datasets.

2.3.1 Regression Task

In this subsection, we evaluate the performances of the TG-LSTM and the vanilla LSTM architectures for the regression problem. The classical LSTM architecture uses the time intervals as another feature in the input vectors, i.e., the LSTM-2 architecture defined in 2.2.4. Therefore, for a dataset with the input size m, the classical LSTM architecture has the input size $m+1$. LSTM-WA represents the classical LSTM architecture in [4], [13], which uses windowing and averaging operations on the data before entering the LSTM network. We train the networks with Stochastic Gradient Descent (SGD) algorithm using the constant learning rate.

We first consider a sine wave with frequency 10 Hz and length $n = 1000$ for training and $n = 500$ for testing. The sampling intervals Δt_k are drawn uniformly from the range $[2, 10]$, $[5, 20]$ and $[20, 50]$ ms for S1, S2 and S3 simulations, respectively. Our aim is to predict the next sample $x_{t_{k+1}}$. For this data, the input is scalar $x_{t_k} \in \mathbb{R}$, i.e., the input size $m = 1$, and the output $d_{t_k} \in \mathbb{R}$, where $d_{t_k} = x_{t_{k+1}}$. For the parameter selection, we perform a grid search on the number of hidden neurons and learning rate in the intervals $q = [3, 20]$ and $\eta = [10^{-3}, 10^{-6}]$, respectively. For the window size of the classical LSTM architecture with preprocessing method, we search on the interval $[\frac{\Delta_{max}}{2}, \Delta_{max}]$, where Δ_{max} equals to 10, 20 and 50 ms, respectively. We choose the parameters with

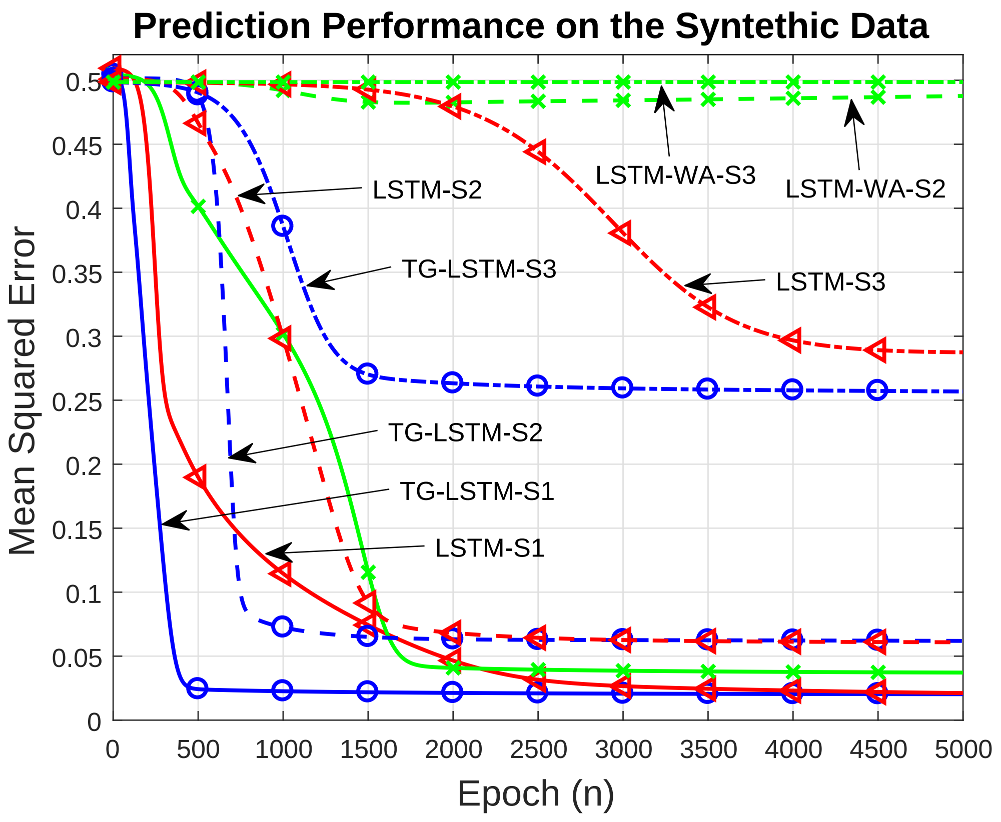


Figure 2.3: Regression performance of the TG-LSTM and the LSTM networks on the syntehtic sine dataset with different sampling intervals. The sampling intervals Δt_k are drawn uniformly from the range $[2, 10]$, $[5, 20]$ and $[20, 50]$ ms for S1, S2 and S3 simulations, respectively. LSTM represents the classical LSTM architecture in [5], which uses the time intervals as another feature. LSTM-WA is the classical LSTM architecture in [4], [13], which uses windowing and averaging operations on the data before entering the LSTM network.

5-fold cross validation, however, we only use the first and last folds for validation to keep the sequential pattern of the data. Otherwise, the sequence is corrupted, e.g., the last sample of the first fold is followed by the first sample of the third fold instead of the second fold. We choose the learning rate as $\eta = 10^{-4}$ for S1 and S2 simulations and $\eta = 2 \times 10^{-5}$ for S3 simulations. The number of hidden neurons are chosen as $q = 20$ for all simulations. The window sizes for the method using windowing and averaging technique are 5, 20 and 50 ms for S1, S2 and S3, respectively. We initiate the weights of the time gates of the TG-LSTM architecture from the distribution $\mathcal{N}(\frac{1}{\mathbb{E}[\Delta t_k]}, 0.01)$ to start the time gates in the smooth area of the sigmoid activation function and prevent the gradients from diminishing due to multiplication. Other weights are initiated from the distribution $\mathcal{N}(0, 0.01)$.

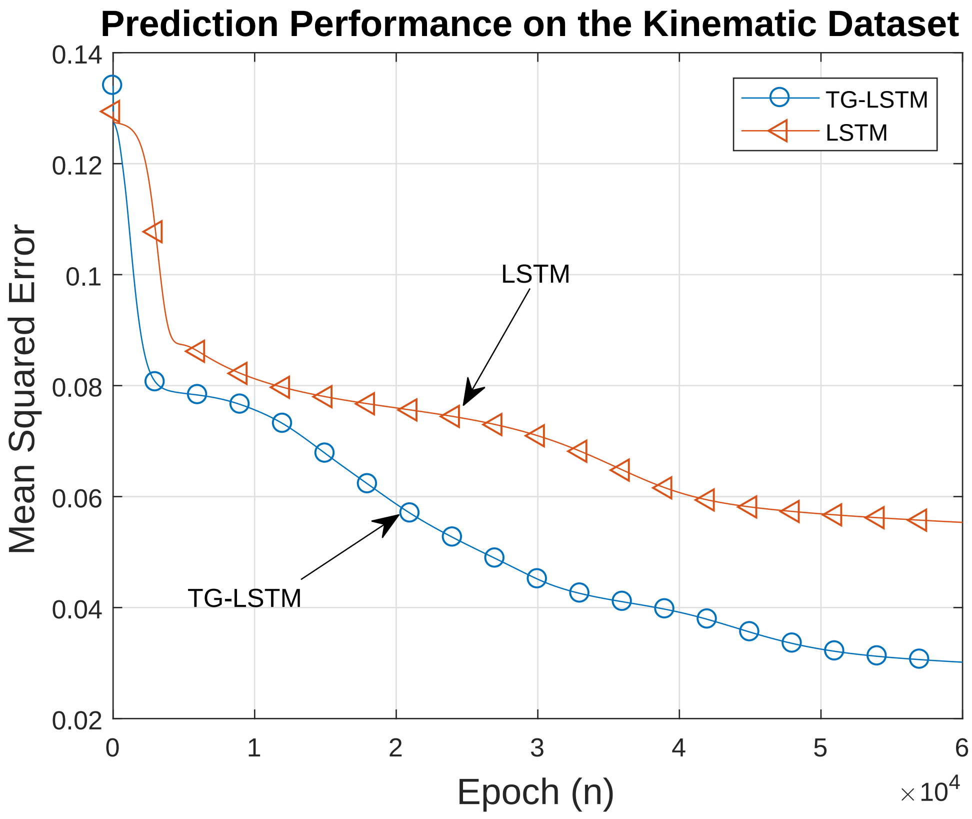


Figure 2.4: Regression performance of the TG-LSTM and LSTM networks on the Kinematic dataset.

In Fig. 2.3, we demonstrate the regression performance of the algorithms under different sampling interval ranges in terms of the mean squared error on the test set per epoch. LSTM represents the classical LSTM architecture in [5], which uses the time intervals as another feature. LSTM-WA is the classical LSTM architecture in [4], [13], which uses windowing and averaging operations on the data before entering the LSTM network. In Fig. 2.3, one can see that the performance improvement by the TG-LSTM architecture becomes more evident for the larger time intervals. While all three architectures achieve similar results in terms of the steady-state error in S1 simulations, the performance difference between the TG-LSTM architecture and the classical LSTM architecture using preprocessing technique significantly increases in S2 simulations. Furthermore, in S3 simulations, we observe a higher performance difference between TG-LSTM and the classical LSTM architectures. Moreover, the TG-LSTM architecture outperforms the other architectures in terms of the convergence rate in all cases.

Other than the sine wave, we compare the TG-LSTM and the classical LSTM architectures on kinematic [17], bank [18] and pumadyn [19] datasets. The results for the LSTM-WA method are not included due to comparatively much better performances of the other methods. Each dataset contains an input vector sequence and the corresponding desired signals for each time step. These datasets do not have separate training and test sets, therefore, we split the sequences in each dataset such that first 60% of the sequence is used for the training and the following 40% is used for the test. Since the datasets contain uniformly sampled sequences, we first need to convert them to the non-uniformly sampled sequences. For this purpose, we sequentially under-sample the sequences based on a probabilistic model. Assume that we have the uniformly sampled input sequence $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l]$. If we receive $\boldsymbol{x}_j$ from the original sequence as $\boldsymbol{x}_{t_k}$, the next sample $\boldsymbol{x}_{t_{k+1}}$ is chosen from the remaining sequence $[\boldsymbol{x}_{j+1}, \ldots, \boldsymbol{x}_l]$ according to the probabilistic model. In our simulations, we use

$$\mathrm{p}(\Delta t_k) = \begin{cases} 0.4, & \text{if } \Delta t_k = 1 \\ 0.4, & \text{if } \Delta t_k = 2 \\ 0.2, & \text{if } \Delta t_k = 3 \\ 0 & \text{otherwise} \end{cases}, \tag{2.25}$$

where $\mathrm{p}(\Delta t_k)$ is the probability mass function for the time difference $\Delta t_k = t_{k+1} - t_k$, e.g., $P(\boldsymbol{x}_{t_{k+1}} = \boldsymbol{x}_{j+1} | \boldsymbol{x}_{t_k} = \boldsymbol{x}_j) = 0.4$ and $P(\boldsymbol{x}_{t_{k+1}} = \boldsymbol{x}_{j+3} | \boldsymbol{x}_{t_k} = \boldsymbol{x}_j) = 0.2$. Using (2.25), we generate the non-uniformly sampled sequence $\boldsymbol{X}_{nu} = [\boldsymbol{x}_{t_1}, \ldots, \boldsymbol{x}_{t_n}]$, $n < l$, and use this sequence in our simulations. Note that, there is no fine tuning on the under-sampling function. We observed similar results with the probabilistic models using different probability mass functions. For each simulation, we used the same number of hidden neurons for both LSTM architectures and set q to the original input size of the dataset, m. Note that, input size for the classical LSTM becomes $m+1$ since we extend the input vector with the time differences, i.e., LSTM-2 architecture.

- Kinematic dataset is a simulation of 8-link all-revolute robotic arm, where the aim is to predict the distance of the effector from the target. The

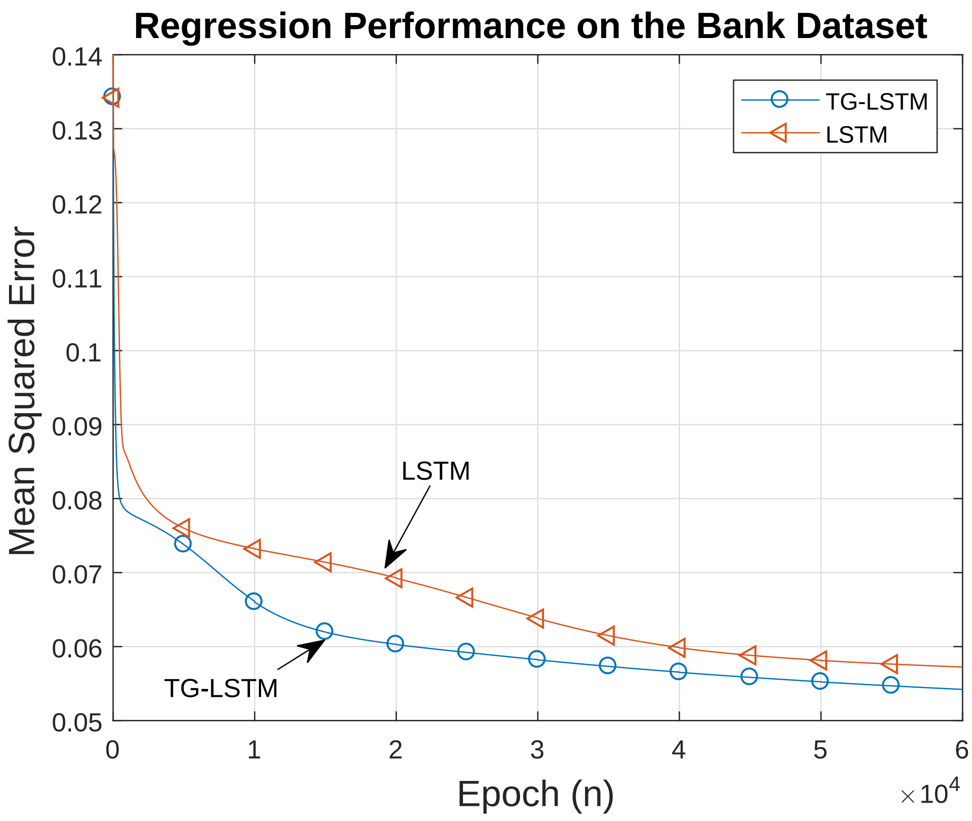


Figure 2.5: Regression performance of the TG-LSTM and LSTM networks on the Bank dataset.

original input vector size $m = 8$ and we set the number of hidden neurons $q = 8$ for both LSTM and TG-LSTM networks. For the SGD algorithm, we select the constant learning rate $\eta = 10^{-5}$ from the interval $[10^{-6}, 10^{-3}]$ using the cross-validation.

- Bank dataset is generated by a simulator, which simulates the queues in banks. Our goal is to predict the fraction of the customers leaving the bank due to long queues. The input vector $\boldsymbol{x}_{t_k} \in \mathbb{R}^{32}$. We set the number of hidden neurons $q = 32$, and the constant learning rate $\eta = 10^{-5}$ from the interval $[10^{-6}, 10^{-3}]$.

- Pumadyn dataset is obtained from a simulation of Unimation Puma 560 robotic arm. Our goal is to predict the angular acceleration of one of the arms. For this dataset, the input vector size $m = 32$ and we set the number of hidden neurons $q = 32$ for both TG-LSTM and LSTM networks. The constant learning rate is set as $\eta = 10^{-5}$ from the interval $[10^{-6}, 10^{-3}]$.

In Fig. 2.4, Fig. 2.5 and Fig.2.6, we illustrate the regression performances of the TG-LSTM and the classical LSTM architectures in terms of mean squared error per epoch for the kinematic, bank and pumadyn datasets, respectively. The TG-LSTM architecture has an outstandingly faster convergence rate compared to the classical LSTM architecture. In addition, the TG-LSTM architecture significantly outperforms the classical LSTM architecture in terms of the steady-state performance. These results show that the time gates, which incorporate the time differences as a nonlinear scaling factor, successfully model the effect of the non-uniform sampling. Both faster convergence and better steady-state performance are achieved by the TG-LSTM architecture. In these simulations no decaying factor is used for the learning rate of SGD algorithm since the architectures are able to converge. In the tasks, which requires a decaying factor for the convergence, the performance difference of the TG-LSTM and the classical LSTM architectures significantly increases since our algorithm has a faster convergence rate.

2.3.2 Classification Task

In this subsection, we evaluate the performances of the TG-LSTM, PLSTM [5] and the classical LSTM architectures for the classification tasks. For this task, we used the real life datasets Pen-Based Recognition of Handwritten Digits [20] and UJI Pen (Version 2) [20]. For the SGD algorithm, we use Adam optimizer [21] with the initial learning rate $\eta = 10^{-3}$.

In the first experiment, we demonstrate the classification performance of the LSTM architrectures on the Pen-Based Recognition of Handwritten Digits [20] dataset. This dataset contains handwritten digits from 44 different writers, where each writer draws 250 digits. These digits are drawn on a 500x500 tablet and uniformly sampled with 100 milliseconds. We non-uniformly under-sample these uniform samples by using (2.25). The input vector $\boldsymbol{x}_{t_k} = [x, y]^T$, where x and y

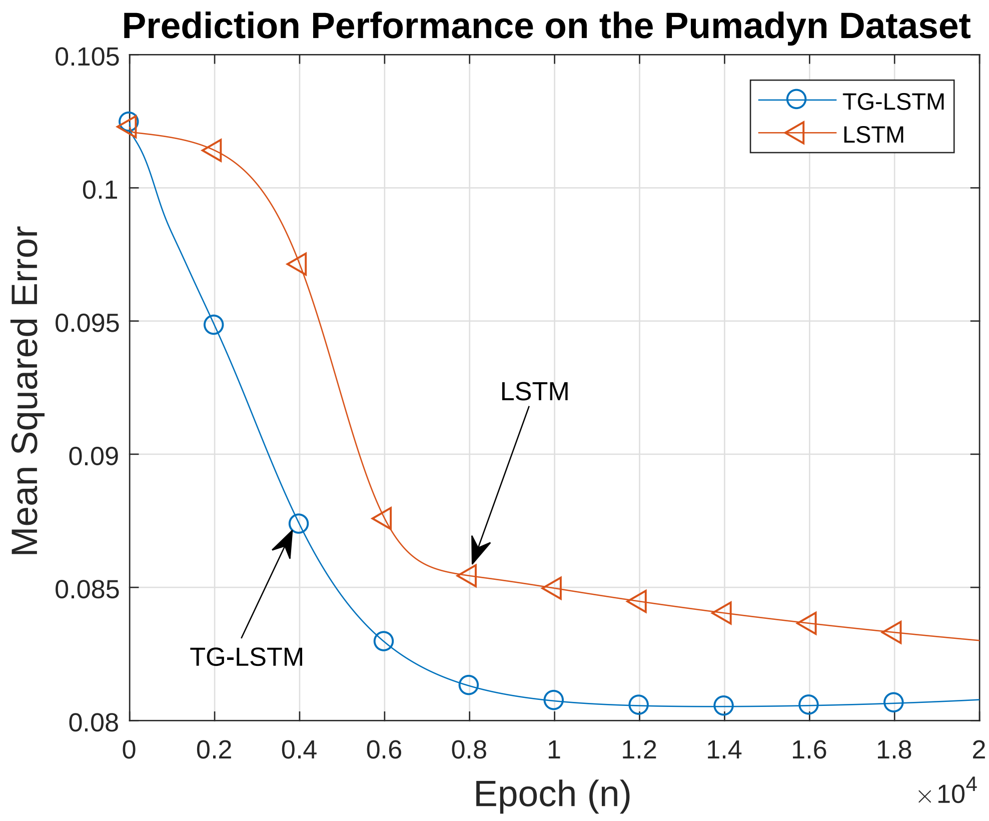
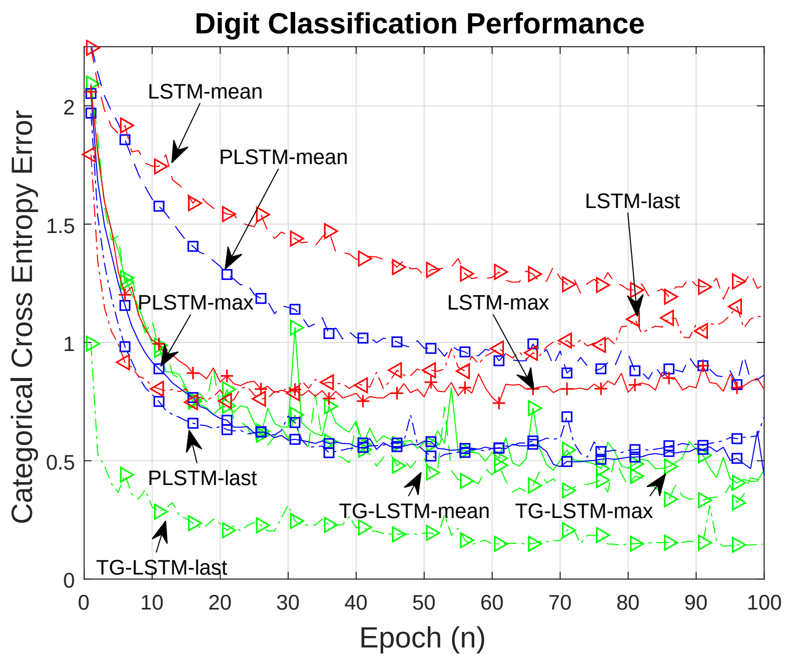


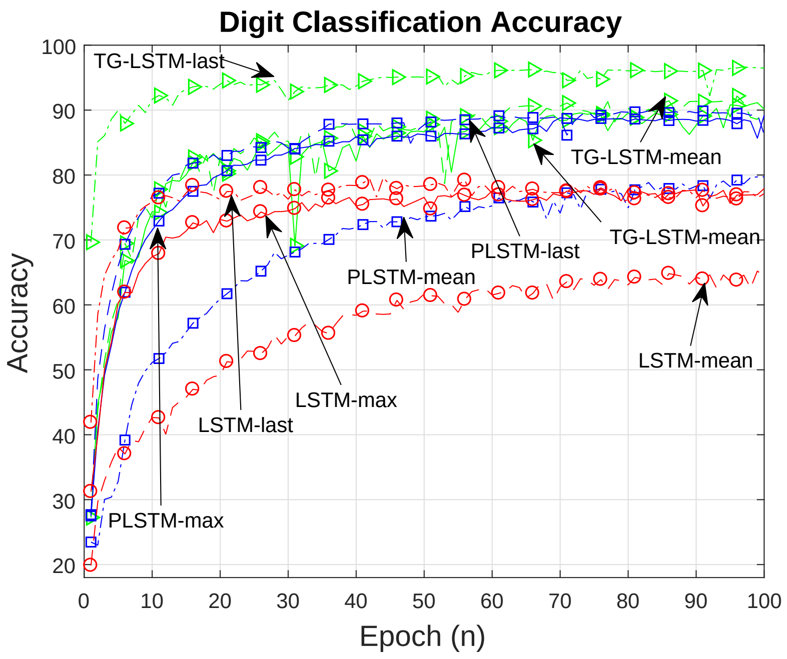
Figure 2.6: Regression performance of the TG-LSTM and LSTM networks on the Pumadyn dataset.

are the coordinates, and the desired signal $d_{t_k} \in \{0, \ldots, 9\}$. For the parameter selection, we use 5-fold cross validation, and set the number of the hidden neurons $q = 100$, which is selected from the set $\{10, 25, 50, 100\}$. For the PLSTM architecture, all three parameters, i.e., the period, shift and on-ratio, are set as trainable to employ the network with full capacity. In Fig. 2.7, we illustrate the cross-entropy loss and the accuracy plots for the architectures with three different pooling methods. We observe from these figures that the TG-LSTM architecture outperforms both the PLSTM and the classical LSTM architectures. In particular, the TG-LSTM architecture using last pooling method significantly improves the performance for both convergence rate and the steady-state accuracy, which shows that the time gates in our method successfully model the the effect of the non-uniform sampling.

We also compare the performance of the architectures on the relatively more



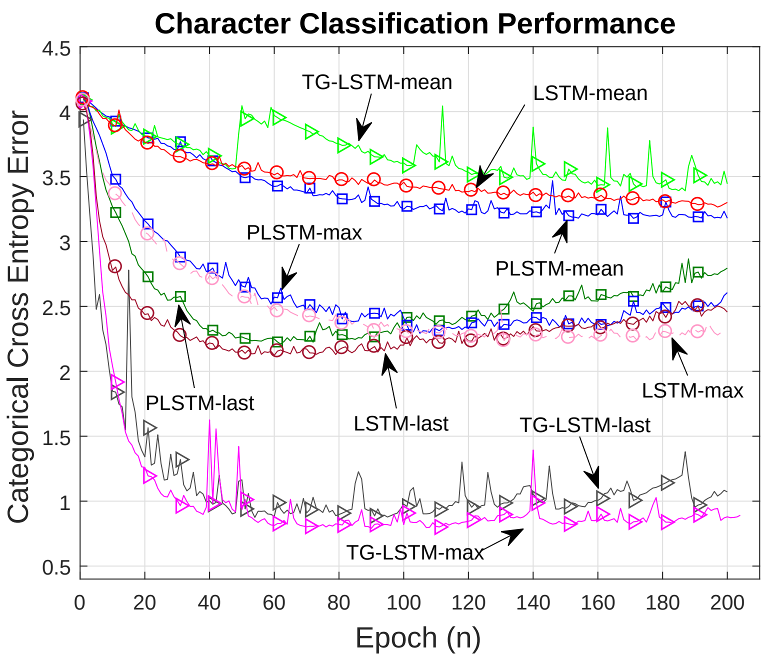
(a)



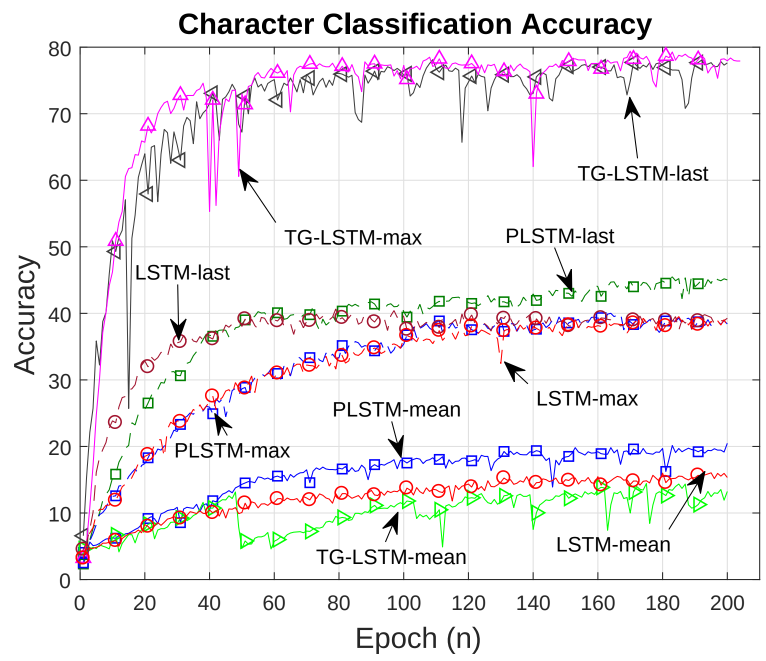
(b)

Figure 2.7: Classification performances based on (a) the categorical cross entropy error (b) the accuracy on the Pen-Based Recognition of Handwritten Digits [20] dataset.

difficult dataset, UJI Pen (Version 2) [20]. This dataset is created by the same method with Pen-Based Recognition of Handwritten Digits [20] dataset. Although there are many other characters in the dataset, we used only upper-case and lower-case letters in the English alphabet, and the digits. The input vector $\boldsymbol{x}_{t_k} = [x, y]^T$, where x and y are the coordinates, and the desired signal $d_{t_k} \in \{a, \dots, z, A, \dots, Z, 1, \dots, 9\}$, where we consider the digit "0" and the upper-case letter "O" as the same label. In this setup, we have 61 different labels, therefore, this a relatively difficult dataset. For all architectures, we set the number of the hidden neurons $q = 100$, which is selected from the set $\{10, 25, 50, 100\}$ by 5-fold cross validation. For the PLSTM architecture, all three parameters are trainable as in the first experiment. In Fig. 2.8, we illustrate the performance of the architectures in terms of the categorical cross entropy error and accuracy, respectively. We observe that the TG-LSTM architecture with max and last pooling methods significantly improve the performance. Since the dataset is more difficult with 61 different classes, the performance increase is more observable in this simulation.



(a)



(b)

Figure 2.8: Classification performances based on (a) the categorical cross entropy error (b) the accuracy on the UJI Pen (Version 2) [20] dataset.

Chapter 3

A Tree Architecture of LSTM Networks for Sequential Regression with Missing Data

We sequentially observe variable length vector sequences $\boldsymbol{X} = [\boldsymbol{x}_{t_1}, ..., \boldsymbol{x}_{t_n}] \in \mathcal{X}$, where $\boldsymbol{x}_{t_k} \in \mathbb{R}^m$ is the regression vector and n is the length of the sequence $\boldsymbol{X}$. Here, the vector sequence $\boldsymbol{X}$ is coming at a constant rate, however, $\boldsymbol{X}$ contains missing samples, i.e., certain regression vectors, $\boldsymbol{x}_{t_k}$, are missing from the data sequence. Note that $\boldsymbol{x}_{t_k}$ is either completely received or completely missing, i.e., we do not consider the case only certain entries of $\boldsymbol{x}_{t_k}$ are missing. The desired output for the regression vector $\boldsymbol{x}_{t_k}$ is given by $d_{t_k} \in \mathbb{R}$ and our goal is to estimate d_{t_k} by

$$\hat{d}_{t_k} = f_{t_k}(\boldsymbol{x}_{t_k}, \ldots, \boldsymbol{x}_{t_1}, d_{t_{k-1}}, \ldots, d_{t_1}),$$

where $f_{t_k}(\cdot)$ is a possibly time varying and adaptive nonlinear regression function at time step t_k. The estimate $\hat{d}_{t_k}$ is a function of the current and past observations. For the input vector $\boldsymbol{x}_{t_k}$, the incurred loss is $l(d_{t_k}, \hat{d}_{t_k})$ and for the whole vector sequence $\boldsymbol{X}$, we suffer $E = \sum_{k=1}^{n} l(d_{t_k}, \hat{d}_{t_k})$.

Since the data has missing samples, the arrival times of the regression vectors

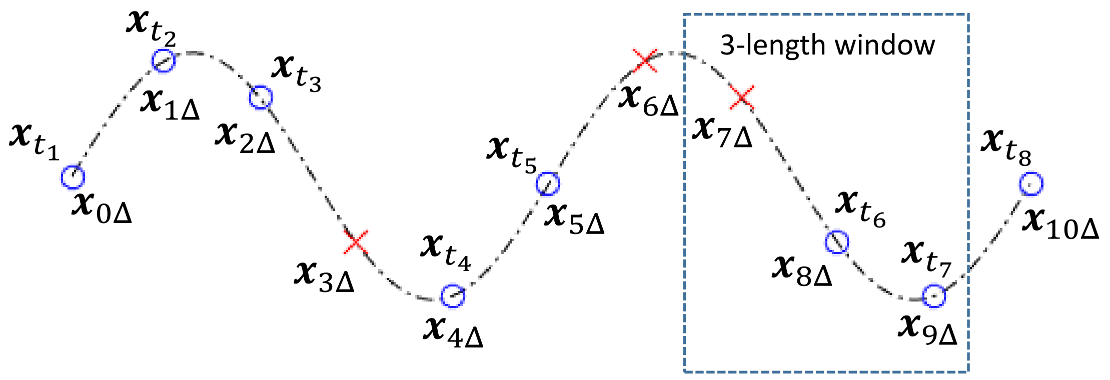


Figure 3.1: An example sinusoidal data sequence with missing samples.

are not regular, i.e., the time intervals between the consecutive regression vectors $\boldsymbol{x}_{t_k}$ and $\boldsymbol{x}_{t_{k+1}}$ may vary and we denote these arrival intervals by Δt_k's,

$$\Delta t_k \triangleq t_k - t_{k-1}.$$

To clarify the framework, in Fig. 3.1, we illustrate an example data sequence $\boldsymbol{X} = [\boldsymbol{x}_{t_1}, ..., \boldsymbol{x}_{t_8}]$ with constant time intervals Δ. However, the data sequence has missing samples, i.e., $\boldsymbol{x}_{3\Delta}$, $\boldsymbol{x}_{6\Delta}$ and $\boldsymbol{x}_{7\Delta}$. Here, $\boldsymbol{x}_{t_k}$ represents the samples in a received order. $\boldsymbol{x}_{m\Delta}$ is the data at time $m\Delta$, which we may receive, e.g., $\boldsymbol{x}_{2\Delta}$, or may not receive, e.g., $\boldsymbol{x}_{3\Delta}$. Hence, for this sequence $\boldsymbol{x}_{t_1} = \boldsymbol{x}_{0\Delta}, \boldsymbol{x}_{t_2} = x_{1\Delta}, \boldsymbol{x}_{t_3} = \boldsymbol{x}_{2\Delta}, \boldsymbol{x}_{t_4} = \boldsymbol{x}_{4\Delta}, \boldsymbol{x}_{t_5} = \boldsymbol{x}_{5\Delta}, \boldsymbol{x}_{t_6} = \boldsymbol{x}_{8\Delta}, \boldsymbol{x}_{t_7} = \boldsymbol{x}_{9\Delta}$ and $\boldsymbol{x}_{t_8} = \boldsymbol{x}_{10\Delta}$. Since the time intervals between the consecutive regression vectors are not regular, the regression function should adapt different cases to predict the desired signal. As an example, let us consider one step ahead prediction regression task for this input sequence. Then, $\boldsymbol{x}_{t_3} = \boldsymbol{x}_{2\Delta}$ should be predicted using the $\boldsymbol{x}_{0\Delta}$ and $\boldsymbol{x}_{1\Delta}$. However, $\boldsymbol{x}_{t_4} = \boldsymbol{x}_{4\Delta}$ should be predicted using the $\boldsymbol{x}_{2\Delta}$, $\boldsymbol{x}_{1\Delta}$ and $\boldsymbol{x}_{0\Delta}$ since $\boldsymbol{x}_{3\Delta}$ is missing.

Here, we use recurrent neural networks to generate the sequential estimates $\hat{d}_{t_k}$. A basic RNN structure is given by [15]

$$\begin{aligned} \boldsymbol{h}_{t_k} &= f(\boldsymbol{W}_h \boldsymbol{x}_{t_k} + \boldsymbol{R}_h \boldsymbol{h}_{t_{k-1}}) \\ \boldsymbol{y}_{t_k} &= g(\boldsymbol{R}_y \boldsymbol{h}_{t_k}), \end{aligned} \tag{3.1}$$

where $\boldsymbol{x}_{t_k} \in \mathbb{R}^m$ is the regression vector, $\boldsymbol{h}_{t_k} \in \mathbb{R}^q$ is the state vector and $\boldsymbol{y}_{t_k} \in \mathbb{R}^q$ is the output at time t_k. $\boldsymbol{W}_h \in \mathbb{R}^{q \times m}$, $\boldsymbol{R}_h \in \mathbb{R}^{q \times q}$ represent the input weight matrices, $\boldsymbol{R}_y \in \mathbb{R}^{q \times q}$ is the output weight matrix. $f(\cdot)$ and $g(\cdot)$ are the nonlinear functions and apply point-wise operations.

As a special case of the RNNs, we focus on the LSTM networks. Among many different variants of the LSTM architecture, we use the most widely used variant, i.e., the LSTM architecture without peephole connections illustrated in Fig. 3.2. The LSTM architecture is given by the following set of equations:

$$\boldsymbol{z}_{t_k} = g(\boldsymbol{W}_z \boldsymbol{x}_{t_k} + \boldsymbol{R}_z \boldsymbol{h}_{t_{k-1}}) \tag{3.2}$$
$$\boldsymbol{i}_{t_k} = \sigma(\boldsymbol{W}_i \boldsymbol{x}_{t_k} + \boldsymbol{R}_i \boldsymbol{h}_{t_{k-1}}) \tag{3.3}$$
$$\boldsymbol{f}_{t_k} = \sigma(\boldsymbol{W}_f \boldsymbol{x}_{t_k} + \boldsymbol{R}_f \boldsymbol{h}_{t_{k-1}}) \tag{3.4}$$
$$\boldsymbol{o}_{t_k} = \sigma(\boldsymbol{W}_o \boldsymbol{x}_{t_k} + \boldsymbol{R}_o \boldsymbol{h}_{t_{k-1}}) \tag{3.5}$$
$$\boldsymbol{c}_{t_k} = \boldsymbol{i}_{t_k} \odot \boldsymbol{z}_{t_k} + \boldsymbol{f}_t \odot \boldsymbol{c}_{t_{k-1}} \tag{3.6}$$
$$\boldsymbol{h}_{t_k} = \boldsymbol{o}_{t_k} \odot g(\boldsymbol{c}_{t_k}), \tag{3.7}$$

where $\boldsymbol{x}_{t_k} \in \mathbb{R}^m$ is the input vector, $\boldsymbol{c}_{t_k} \in \mathbb{R}^q$ is the state vector and $\boldsymbol{h}_{t_k} \in \mathbb{R}^q$ is the output vector of the LSTM network at time t_k. $\boldsymbol{z}_{t_k}$ is the block input, $\boldsymbol{i}_{t_k}$, $\boldsymbol{f}_{t_k}$, $\boldsymbol{o}_{t_k}$ represent the input, forget and output gates at time t_k, respectively. $\boldsymbol{W}_z$, $\boldsymbol{W}_i$, $\boldsymbol{W}_f$, $\boldsymbol{W}_o \in \mathbb{R}^{q \times m}$ are the input weight matrices and $\boldsymbol{R}_z$, $\boldsymbol{R}_i$, $\boldsymbol{R}_f$, $\boldsymbol{R}_o \in \mathbb{R}^{q \times q}$ are the recurrent input weight matrices. $g(\cdot)$ and $\sigma(\cdot)$ are the point-wise nonlinear activation functions. $g(\cdot)$ is commonly set to the tangent hyperbolic function, i.e., $\tanh(\cdot)$ and $\sigma(\cdot)$ is the sigmoid function. With the abuse of notation, we incorporate the bias weights, $\boldsymbol{b}_z$, $\boldsymbol{b}_i$, $\boldsymbol{b}_f$, $\boldsymbol{b}_o \in \mathbb{R}^q$, into the input weight matrices and denote them by $\boldsymbol{W}_\theta = [\boldsymbol{W}_\theta; \boldsymbol{b}_\theta]$, $\theta \in \{z, i, f, o\}$, where $\boldsymbol{x}_t = [\boldsymbol{x}_t; 1]$.

As described in the following section, we estimate the desired signal d_{t_k} by

$$\hat{d}_{t_k} = \hat{\boldsymbol{w}}_{t_k}^T \hat{\boldsymbol{h}}_{t_k}, \tag{3.8}$$

where $\hat{\boldsymbol{w}}_{t_k} \in \mathbb{R}^q$ is the regression coefficients. To obtain $\hat{\boldsymbol{h}}_{t_k}$, we adaptively combine the outputs of the different LSTM networks in our architecture by

$$\hat{\boldsymbol{h}}_{t_k} = \sum_{i=1}^{K_{t_k}} \alpha_{t_k}^{(i)} \boldsymbol{h}_{t_k}^{(i)}, \tag{3.9}$$

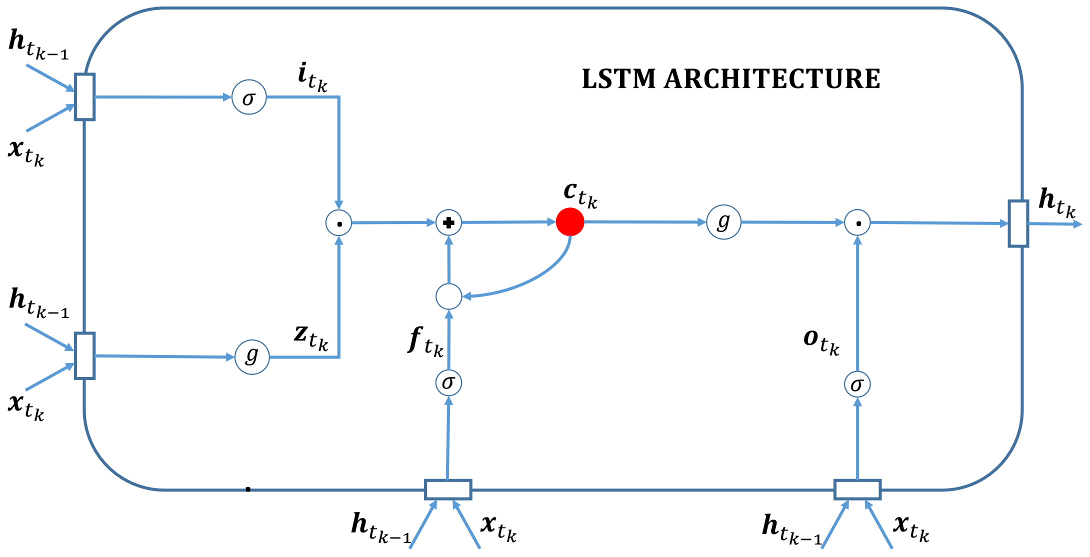


Figure 3.2: Detailed schematic of the LSTM architecture.

where $\boldsymbol{h}_{t_k}^{(i)}$ is the output of the i^{th} LSTM network, K_{t_k} is the number of the total LSTM networks to be combined at time t_k and $\hat{\boldsymbol{h}}_{t_k}$ is the linear combination of these outputs.

In the following, we introduce a tree architecture based on the LSTM networks working on the sequential data with missing samples, and also provide its forward-pass and backward-pass update formulas.

3.1 LSTM Network Based Tree Architecture

Since the time intervals between the regression vectors are not regular in the case of missing data, the regression function should adapt to different scenarios to estimate the desired signal. Hence, we directly incorporate the missingness information into our nonlinear regression function $f_{t_k}(\cdot)$ to model the effect of the missing data in our sequence. In our algorithm, we consider the missing input values in a particular window, which shifts at each time step.

For this purpose, we first partition the regression function $f_{t_k}(\cdot)$ into two parts

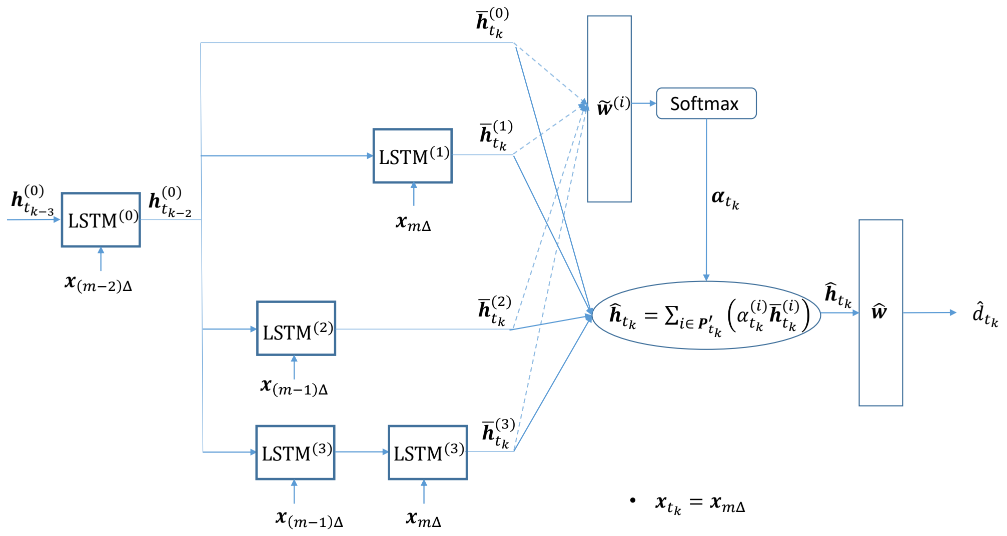


Figure 3.3: Detailed schematic of the Tree-LSTM architecture, where the tree depth $L = 2$. Note that $\boldsymbol{x}_{t_k} = \boldsymbol{x}_{m\Delta}$.

as follows

$$\hat{d}_{t_k} = \theta^{\mathrm{M}}_{t_k} f^{\mathrm{M}}_{t_k}(\cdot) + \theta^{\mathrm{W}}_{t_k} f^{\mathrm{W}}_{t_k}(\cdot), \tag{3.10}$$

where $f^{\mathrm{W}}_{t_k}(\cdot)$ processes the input sequence in a particular window, e.g., the length-3 window in Fig. 3.1, and $f^{\mathrm{M}}_{t_k}(\cdot)$ is the main regression function using the whole input sequence except the samples inside the window. The purpose of two distinct functions will be clear in the following. Specifically, for a length-L window, $f^{\mathrm{M}}_{t_k}(\cdot)$ and $f^{\mathrm{W}}_{t_k}(\cdot)$ process the inputs $[\boldsymbol{x}_{0\Delta}, \ldots, \boldsymbol{x}_{(m-L)\Delta}]$ and $[\boldsymbol{x}_{(m-L+1)\Delta}, \ldots, \boldsymbol{x}_{m\Delta}]$, where $t_k = m\Delta$, respectively. Here, $f^{\mathrm{M}}_{t_k}(\cdot)$ captures the general pattern of the data, while $f^{\mathrm{W}}_{t_k}(\cdot)$ provides more elaborate decisions on the inputs inside the window. Next, we incorporate the missingness information into $f^{\mathrm{W}}_{t_k}(\cdot)$, i.e., $f^{\mathrm{W}}_{t_k}(\cdot, \boldsymbol{p}^{(L)}_{t_k})$. We define presence-pattern, i.e., the pattern of the present input samples, $\boldsymbol{p}^{(L)}_{t_k} = [p^{(L)}_{t_k,1}, \ldots, p^{(L)}_{t_k,L}] \in \{0,1\}^L$, which holds the missingness information in its most explicit form, i.e., whether the inputs $[\boldsymbol{x}_{(m-L+1)\Delta}, \ldots, \boldsymbol{x}_{m\Delta}]$ exist or not, where $\boldsymbol{x}_{t_k} = \boldsymbol{x}_{m\Delta}$. For example, a length-3 presence-pattern $\boldsymbol{p}^{(3)}_{t_k} = [1, 0, 1]$ indicates that $\boldsymbol{x}_{m\Delta}$ and $\boldsymbol{x}_{(m-2)\Delta}$ are received, however, $\boldsymbol{x}_{(m-1)\Delta}$ is missing from the input sequence, where $t_k = m\Delta$. The presence-pattern always has the same length with the window, hence, we drop the length L to simplify the notation, i.e., $\boldsymbol{p}_{t_k}$.

Note that we explicitly incorporate the missingness information into $f_{t_k}^{\mathrm{W}}(\cdot, \boldsymbol{p}_{t_k})$, besides, $f_{t_k}^{\mathrm{M}}(\cdot)$ implicitly carries this information from the past inputs thanks to the memory in the LSTM architecture since we sequentially process the data.

There exist L inputs inside the window, which corresponds to 2^L possible unique presence-patterns since each input has two options, i.e., may or may not exist. Since all-zero presence-pattern indicates all of the inputs inside the window are missing, we have $2^L - 1$ presence-patterns containing inputs to be processed. In our algorithm, we assign a unique regression function to process each of these patterns, while $f_{t_k}^{\mathrm{M}}(\cdot)$ corresponds to all-zero pattern since it only uses the inputs outside the window, which is the main reason using two functions in (3.10). For this purpose, we divide $f_{t_k}^{\mathrm{W}}(\cdot, \boldsymbol{p}_{t_k})$ into $2^L - 1$ components as follows

$$\hat{d}_{t_k} = \theta_{t_k}^{\mathrm{M}} f_{t_k}^{\mathrm{M}}(\cdot) + \theta_{t_k}^{\mathrm{W}} \sum_{i=1}^{2^L-1} \beta_{t_k}^{(i)} f_{t_k}^{\mathrm{W}_i}(\cdot, \boldsymbol{p}_{t_k}), \tag{3.11}$$

where $\theta_{t_k}^{\mathrm{M}}$, $\theta_{t_k}^{\mathrm{W}}$ and $\beta_{t_k}^{(i)} \in \mathbb{R}$. Each $f_{t_k}^{\mathrm{W}_i}(\cdot)$ is the regression function specifically assigned to process the input sequence with a unique presence-pattern $\boldsymbol{p}^{(i)}$. As an example, $f_{t_k}^{\mathrm{W}_1}(\cdot)$, $f_{t_k}^{\mathrm{W}_5}(\cdot)$ and $f_{t_k}^{\mathrm{W}_7}(\cdot)$ process the inputs for length-3 presence-patterns $\boldsymbol{p}^{(1)} = [0, 0, 1]$, $\boldsymbol{p}^{(5)} = [1, 0, 1]$ and $\boldsymbol{p}^{(7)} = [1, 1, 1]$, respectively, which is the binary representation of the number i using L bits. We can also consider $f_{t_k}^{\mathrm{M}}(\cdot)$ is the regression function for the all-zero presence-pattern $\boldsymbol{p}^{(0)} = [0, 0, 0]$. In its most extensive form, our model contains 2^L unique regression functions, the computational loads and the methods for reducing the number of regression functions will be explained in the following.

One can directly use $f_{t_k}^{\mathrm{W}_i}(\cdot)$ to process the inputs inside the window, when $\boldsymbol{p}^{(i)} = \boldsymbol{p}_{t_k}$. However, note that certain presence-patterns inherently contain the other presence-patterns, therefore, we can use multiple regression functions to improve the estimate $\hat{d}_{t_k}$ for the same $\boldsymbol{p}_{t_k}$. For example, when $\boldsymbol{p}_{t_k} = [0, 1, 1]$ is received, we also obtain the patterns $[0, 0, 0]$, $[0, 0, 1]$ and $[0, 1, 0]$. Therefore, we have sufficient information to use and train $f_{t_k}^{\mathrm{M}}(\cdot)$, $f_{t_k}^{\mathrm{W}_1}(\cdot)$ and $f_{t_k}^{\mathrm{W}_2}(\cdot)$ in addition to $f_{t_k}^{\mathrm{W}_3}(\cdot)$. To represent this relation, we define presence subpattern $\bar{\boldsymbol{p}}_{t_k}$ such that if $\bar{\boldsymbol{p}}_{t_k,j} \leq \boldsymbol{p}_{t_k,j}, \forall j$ excluding $\bar{\boldsymbol{p}}_{t_k} = \boldsymbol{p}_{t_k}$, then $\bar{\boldsymbol{p}}_{t_k}$ is a subpattern of $\boldsymbol{p}_{t_k}$. Next, we define the set $\boldsymbol{P}_{t_k}$ as the active set of the presence-pattern $\boldsymbol{p}_{t_k}$ such that $\boldsymbol{P}_{t_k}$ contains all possible subbatterns of $\boldsymbol{p}_{t_k}$ in addition to $\boldsymbol{p}_{t_k}$ itself. To simplify the notation, we also define the set $\boldsymbol{P}'_{t_k}$ such that $\boldsymbol{P}'_{t_k}$ contains the decimal representations of the presence-patterns included in the set $\boldsymbol{P}_{t_k}$. As an example, for $\boldsymbol{p}_{t_k} = [1, 0, 1]$, the active set $\boldsymbol{P}_{t_k} = \{[0, 0, 0], [0, 0, 1], [1, 0, 0], [1, 0, 1]\}$ and $\boldsymbol{P}'_{t_k} = \{0, 1, 4, 5\}$. Note that all-zero pattern, i.e., $[0, 0, 0]$ for $L = 3$, is always included in the active set.

In our architecture, we use a separate LSTM network to model each regression function in (3.11). In particular, we employ one main LSTM network modelling $f^{\mathrm{M}}_{t_k}(\cdot)$ and also many different leaf LSTM networks, which model the regression functions $f^{\mathrm{W}_i}_{t_k}(\cdot)$ in (3.11). While the main LSTM network captures the general pattern of the data, the leaf LSTM networks provide more precise outputs based on the presence-pattern inside the window. We emphasize that the leaf LSTM networks use the information not only from the inputs inside the window, but also history of the sequence, since they receive the state and output of the main LSTM network as their initial state and recurrent input. Since these leaf LSTM networks receive the state and output of the main LSTM network as their initial state and recurrent input, they also have information on history of the sequence. We then combine the outputs of these LSTM networks to generate our final output.

In our algorithm, each leaf LSTM network is assigned to a particular presence-pattern. If an input is missing in length-L window, the LSTM networks containing this input in their assigned input sequence do not generate their outputs. Therefore, only the certain leaf LSTM networks contribute to the final output in the case of missing data based on the existence of the inputs inside the window. By this way, we directly incorporate the missingness information by selecting the particular leaf LSTM networks instead of artificially inserting it into the input vectors as done in literature [4], [22]. Due to this hierarchical nature we name our architecture as the Tree-LSTM architecture.

To clarify the algorithm, let us say we receive a sequence with missing samples as illustrated in Fig. 3.1 and the aim is to predict the next sample, i.e., $\boldsymbol{d}_{t_k} = \boldsymbol{x}_{t_{k+1}}$. For example, to estimate $\boldsymbol{x}_{t_8} = \boldsymbol{x}_{10\Delta}$ in Fig. 3.1, the length-3 window encapsulates the inputs $[\boldsymbol{x}_{7\Delta}, \boldsymbol{x}_{8\Delta}, \boldsymbol{x}_{9\Delta}]$. The main LSTM network processes the existing inputs before this window, i.e., $[\boldsymbol{x}_{t_1}, \ldots, \boldsymbol{x}_{t_5}] = [\boldsymbol{x}_{0\Delta}, \ldots, \boldsymbol{x}_{5\Delta}]$ and generates its state and output vectors. Since $\boldsymbol{x}_{7\Delta}$ is missing from our sequence, only the leaf LSTM networks, which do not contain $\boldsymbol{x}_{7\Delta}$ in their input sequences, are able to generate their outputs. Next, we combine the outputs of different LSTM networks and obtain our final estimate.

In Section 3.1.1, we first provide our architecture with a specific depth to clarify the framework. In particular, we select the depth as $L = 2$ to provide a clear representation of the algorithm with a small number of LSTM networks. We then extend this architecture to the generic case in 3.1.2.

3.1.1 A Specific Tree-LSTM Architecture

Suppose the depth of the Tree-LSTM network is $L = 2$ and we estimate the generic desired signal as in Fig. 3.3. The architecture contains $2^2 = 4$ different LSTM networks. For each LSTM network, there exists a weight vector, $\tilde{\boldsymbol{w}}^{(0)}, \tilde{\boldsymbol{w}}^{(1)}, \tilde{\boldsymbol{w}}^{(2)}, \tilde{\boldsymbol{w}}^{(3)} \in \mathbb{R}^{4+q}$ to combine the outputs of these LSTM networks. $\hat{\boldsymbol{w}} \in \mathbb{R}^{q+1}$ is the final regression coefficients. $\boldsymbol{W}_z^{(j)}$, $\boldsymbol{W}_i^{(j)}$, $\boldsymbol{W}_f^{(j)}$, $\boldsymbol{W}_o^{(j)} \in \mathbb{R}^{q \times m}$ are the input weight matrices and $\boldsymbol{R}_z^{(j)}$, $\boldsymbol{R}_i^{(j)}$, $\boldsymbol{R}_f^{(j)}$, $\boldsymbol{R}_o^{(j)} \in \mathbb{R}^{q \times q}$ are the recurrent weight matrices of the j^{th} LSTM network, i.e., $\text{LSTM}^{(j)}$ in Fig. 3.3.

This architecture as shown in Fig. 3.3 contains four different LSTM networks, i.e., $\text{LSTM}^{(0)}$, $\text{LSTM}^{(1)}$, $\text{LSTM}^{(2)}$ and $\text{LSTM}^{(3)}$, where each LSTM network is responsible for processing the data sequence with a particular presence-pattern. In particular, $\text{LSTM}^{(0)}$, $\text{LSTM}^{(1)}$, $\text{LSTM}^{(2)}$ and $\text{LSTM}^{(3)}$ are assigned to the presence-patterns $[0, 0]$, $[0, 1]$, $[1, 0]$ and $[1, 1]$, respectively. Here, we have one main LSTM network, i.e., $\text{LSTM}^{(0)}$, to identify the general pattern and propagate the essential state information contained in the state and output vectors $\boldsymbol{c}_{t_k}^{(0)}$ and $\boldsymbol{h}_{t_k}^{(0)}$. The other three LSTM networks, i.e., $\text{LSTM}^{(1)}$, $\text{LSTM}^{(2)}$ and $\text{LSTM}^{(3)}$, are the leaf LSTM networks. They receive $\boldsymbol{c}_{t_k}^{(0)}$ and $\boldsymbol{h}_{t_k}^{(0)}$ as their initial states and process their input sequences inside the length-2 window. Note that while the $\text{LSTM}^{(0)}$ network runs over the whole sequence, the other three LSTM networks process only the data sequence corresponding to their presence-patterns in this window.

When a regression vector $\boldsymbol{x}_{t_k}$ with a particular presence-pattern $\boldsymbol{p}_{t_k}$ is received, only the LSTM networks included in the active set of $\boldsymbol{p}_{t_k}$, i.e., $\boldsymbol{P}'_{t_k}$, process their corresponding input sequences and generate their outputs. For example, when $\boldsymbol{p}_{t_k} = [1, 1]$, all of the four LSTM networks generate output since $\boldsymbol{P}' = \{0, 1, 2, 3\}$. To generate these outputs, firstly, the main LSTM network processes the input $\boldsymbol{x}_{t_{k-2}}$ and generates its state and output vectors, $\boldsymbol{c}^{(0)}_{t_{k-2}}$ and $\boldsymbol{h}^{(0)}_{t_{k-2}}$, respectively. As shown in Fig. 3.3, these vectors are passed to the leaf LSTM networks as their initial states and recurrent inputs. Then, each leaf LSTM network processes its corresponding input sequence, i.e., the input is merely $\boldsymbol{x}_{t_k}$ for LSTM$^{(1)}$, similarly, merely $\boldsymbol{x}_{t_{k-1}}$ for LSTM$^{(2)}$ and $[\boldsymbol{x}_{t_{k-1}}, \boldsymbol{x}_{t_k}]$ for LSTM$^{(3)}$. Although, all four LSTM networks are active for this presence-pattern $\boldsymbol{p}_{t_k} = [1, 1]$, this is not the case for the other presence-patterns. As an example, for the presence-pattern $\boldsymbol{p}_{t_k} = [1, 0]$, only LSTM$^{(0)}$ and LSTM$^{(2)}$ generate output since $P'_{t_k} = \{0, 2\}$. Note that LSTM$^{(0)}$ generates output at each time step since presence-pattern $[0, 0]$ is included in the active set for any pattern, i.e., $[0, 0] \in \forall \boldsymbol{P}_{t_k}$.

We combine the outputs the active LSTM networks as

$$\hat{\boldsymbol{h}}_{t_k} = \sum_{i=1}^{4} \alpha^{(i)}_{t_k} \bar{\boldsymbol{h}}^{(i)}_{t_k}, \tag{3.12}$$

where $\alpha^{(i)}_{t_k}$ is the weight for the output of the ith LSTM network, i.e., LSTM$^{(i)}$, as learned in the following. $\bar{\boldsymbol{h}}^{(i)}_{t_k}$ is defined for the consistency in the time subscripts and represents the last output of the LSTM$^{(i)}$ network to be combined at time t_k. For example, in Fig. 3.3, $\bar{\boldsymbol{h}}^{(0)}_{t_k} = \boldsymbol{h}^{(0)}_{t_{k-2}}$, $\bar{\boldsymbol{h}}^{(1)}_{t_k}$ and $\bar{\boldsymbol{h}}^{(3)}_{t_k}$ are the outputs of the LSTM$^{(1)}$ and LSTM$^{(3)}$ networks generated by using the input $\boldsymbol{x}_{m\Delta}$. Similarly, $\bar{\boldsymbol{h}}^{(2)}_{t_k}$ is the output of the LSTM$^{(2)}$ network generated with the input $\boldsymbol{x}_{(m-1)\Delta}$ since the last input for the LSTM$^{(2)}$ network is $\boldsymbol{x}_{(m-1)\Delta}$. $\hat{\boldsymbol{h}}_{t_k}$ is the linear combination of these outputs.

Since the number of the LSTM outputs to be combined varies with respect to the presence-pattern at each time step, we need an adaptive algorithm to determine weights $\alpha^{(i)}_{t_k}$. Otherwise, we may obtain the combined outputs from very distant ranges, e.g., too small or too high values of the combined outputs for $\boldsymbol{p}_{t_k} = [0,0]$ and $\boldsymbol{p}_{t_k} = [1,1]$, respectively. To address this problem, we use softmax$(\cdot)$ function to determine the combination weights $\alpha_{t_k}^{(i)}$, hence, the sum of the combination weights is always set to 1, i.e., $\sum_{i=1}^{4} \alpha_{t_k}^{(i)} = 1$. The combination weights are calculated as

$$\alpha_{t_k}^{(i)} = \begin{cases} \frac{\exp\left(\tilde{\boldsymbol{w}}^{(i)T} \tilde{\boldsymbol{h}}_{t_k}^{(i)}\right)}{\sum_{j \in \boldsymbol{P}'_{t_k}} \exp\left(\tilde{\boldsymbol{w}}^{(j)T} \tilde{\boldsymbol{h}}_{t_k}^{(j)}\right)} & \text{if } i \in \boldsymbol{P}'_{t_k} \\ 0 & \text{otherwise} \end{cases}, \tag{3.13}$$

where $\tilde{\boldsymbol{h}}_{t_k}^{(i)} \in \mathbb{R}^{4+q}$ is defined as $\tilde{\boldsymbol{h}}_{t_k}^{(i)} = [\boldsymbol{p}_{t_k}; \boldsymbol{p}^{(i)}; \bar{\boldsymbol{h}}_{t_k}^{(i)}]$, i.e., we incorporate the missingness information in length-2 window to the weight calculations by appending the presence-patterns of the current input and the LSTM$^{(i)}$ network to the LSTM network outputs. Note that we consider only the outputs of the LSTM networks in the active set $\boldsymbol{P}'_{t_k}$ to calculate combined output $\hat{\boldsymbol{h}}_{t_k}$. The final estimate of the desired signal is calculated by

$$\hat{d}_{t_k} = \hat{\boldsymbol{w}}_{t_k}^T \hat{\boldsymbol{h}}_{t_k}. \tag{3.14}$$

In Section 3.1.2, we explain the proposed architecture for the generic case, i.e., length-L window.

3.1.2 Generic Tree-LSTM Architecture

In this subsection, we consider the Tree-LSTM architecture for the generic case, i.e., the depth of the Tree-LSTM is L. The architecture contains 2^L different LSTM networks, where each LSTM network specializes in estimation of desired signal d_{t_k} from an input sequence $[\boldsymbol{x}_{t_k}, \ldots, \boldsymbol{x}_{t_1}]$ with a particular presence-pattern $\boldsymbol{p}_{t_k} \in \mathbb{R}^L$. For each LSTM network, there exist weight vectors $\tilde{\boldsymbol{w}}^{(j)} \in \mathbb{R}^{2\times L+q}$ to adaptively generate the combination weights of the outputs of these LSTM networks. $\tilde{\boldsymbol{w}}^{(i)} \in \mathbb{R}^{q+1}$ is the regression weights to generate the final estimate. $\boldsymbol{W}_z^{(j)}$, $\boldsymbol{W}_i^{(j)}$, $\boldsymbol{W}_f^{(j)}$, $\boldsymbol{W}_o^{(j)} \in \mathbb{R}^{q\times m}$ are the input weight matrices and $\boldsymbol{R}_z^{(j)}$, $\boldsymbol{R}_i^{(j)}$, $\boldsymbol{R}_f^{(j)}$, $\boldsymbol{R}_o^{(j)} \in \mathbb{R}^{q\times q}$ are the recurrent weight matrices of the LSTM$^{(j)}$ network.

Algorithm The Tree-LSTM Network Regressor

```
1: $k = 0$
2: for m = 1 to $L$ do
3:     $\boldsymbol{x}_{t_k} = \boldsymbol{x}_{m\Delta}$
4: end for
5: for $m = L + 1$ to $N$ do
6:     if $\boldsymbol{x}_{(m)\Delta}$ exists then
7:         $k = k + 1$
8:         $\boldsymbol{x}_{t_k} = \boldsymbol{x}_{m\Delta}$
9:     end if
10:    if $\boldsymbol{x}_{(m-L)\Delta}$ exists then
11:        $\boldsymbol{h}^{(0)}_{t_{k-L}}, \boldsymbol{c}^{(0)}_{t_{k-L}} \Leftarrow \text{LSTM}^{(0)}(\boldsymbol{x}_{t_{k-L}})$
12:    else
13:        $\boldsymbol{h}^{(0)}_{t_{k-L}} = \boldsymbol{h}^{(0)}_{t_{k-L-1}}$
14:        $\boldsymbol{c}^{(0)}_{t_{k-L}} = \boldsymbol{c}^{(0)}_{t_{k-L-1}}$
15:    end if
16:    $\bar{\boldsymbol{h}}^{(0)}_{t_k} = \boldsymbol{h}^{(0)}_{t_{k-L}}$
17:    for all $i \in \boldsymbol{S}'_{t_k}$ do
18:        $\boldsymbol{H}^{(i)}_{t_k,1} = \boldsymbol{h}^{(0)}_{t_{k-L-1}}$
19:        $\boldsymbol{C}^{(i)}_{t_k,1} = \boldsymbol{c}^{(0)}_{t_{k-L-1}}$
20:        for $j = 1$ to $||\boldsymbol{p}^{(i)}||_1$ do
21:            $\boldsymbol{H}^{(i)}_{t_k,j}, \boldsymbol{C}^{(i)}_{t_k,j} \Leftarrow \text{LSTM}^{(i)}(\boldsymbol{X}^{(i)}_{t_k,j})$
22:        end for
23:        $\bar{\boldsymbol{h}}^{(i)}_{t_k} = \boldsymbol{H}^{(i)}_{t_k,||\boldsymbol{p}^{(i)}||_1}$
24:    end for
25:    for all $i \in \boldsymbol{P}'_{t_k}$ do
26:        $\tilde{\boldsymbol{h}}^{(i)}_{t_k} = [\boldsymbol{p}_{t_k}; \boldsymbol{p}^{(i)}; \bar{\boldsymbol{h}}^{(i)}_{t_k}]$
27:        $\alpha^{(i)} = \text{softmax}(\tilde{\boldsymbol{w}}^{(i)T} \tilde{\boldsymbol{h}}^{(i)}_{t_k})$
28:    end for
29:    $\hat{\boldsymbol{h}}_{\boldsymbol{t_k}} = \sum_{i \in \boldsymbol{P}'} \alpha^{(i)} \bar{\boldsymbol{h}}^{(i)}_{t_k}$
30:    $d_{t_k} = \hat{\boldsymbol{w}}^T \hat{\boldsymbol{h}}_{\boldsymbol{t_k}}$
31:    $e_{t_k} = \frac{1}{2}(d_t - \hat{d}_t)^2$
32: end for
```

To generate the estimate of the desired signal d_{t_k}, we first create the presence-pattern $\boldsymbol{p}_{t_k}$ and its corresponding sets $\boldsymbol{P}_{t_k}$, $\boldsymbol{P}'_{t_k}$ by considering the existence of the last L input vectors, i.e., $[\boldsymbol{x}_{(m-L+1)\Delta}, \ldots, \boldsymbol{x}_{m\Delta}]$, where $m\Delta = t_k$. Based on this presence-pattern, we choose $2^{||\boldsymbol{p}_{t_k}||_1}$ LSTM networks, i.e., LSTM$^{(j)}$, where $j \in \boldsymbol{P}'$, among the total 2^L LSTM networks in the architecture. As described in Algorithm, firstly, the main LSTM network, LSTM$^{(0)}$, generates its state and output vectors, $\boldsymbol{h}^{(0)}_{t_{k-L}}, \boldsymbol{c}^{(0)}_{t_{k-L}}$, by processing the input $\boldsymbol{x}_{(m-L)\Delta}$, if it exists. Otherwise, we directly use the previous state and output vectors of the main LSTM network. We then pass these state and output vectors of the main LSTM network, i.e., $\boldsymbol{c}^{(0)}_{t_{k-L}}$ and $\boldsymbol{h}^{(0)}_{t_{k-L}}$, to the leaf LSTM networks, i.e., LSTM$^{(j)}$, where $j \in \boldsymbol{P}'_{t_k}$, as their initial state and recurrent input vectors. Each active leaf LSTM network processes its corresponding length-$||\boldsymbol{p}^{(i)}||_1$ input sequence $\boldsymbol{X}^{(i)}_{t_k}$ and generates its output vector. Note that in Algorithm, $\boldsymbol{H}^{(i)}_{t_k}$, $\boldsymbol{C}^{(i)}_{t_k} \in \mathbb{R}^{q \times ||\boldsymbol{p}^{(i)}||_1}$ are the matrices storing the state and output vectors of the LSTM$^{(i)}$ in their columns, respectively. To simplify the notation, we denote the last output vector of each LSTM network by $\bar{\boldsymbol{h}}^{(i)}_{t_k}$. We then create $\tilde{\boldsymbol{h}}^{(i)}_{t_k}$ vectors for our combination algorithm by appending $\boldsymbol{p}_{t_k}$ and $\boldsymbol{p}^{(i)}$, which represents the presence-pattern of the LSTM$^{(i)}$ network, to these output vectors, i.e., $[\boldsymbol{p}_{t_k}; \boldsymbol{p}^{(i)}; \bar{\boldsymbol{h}}^{(i)}_{t_k}]$. Here, each combination weight is conditioned on the input presence-pattern and the assigned presence- pattern of the LSTM network. We generate the combination weights as

$$\alpha^{(i)}_{t_k} = \begin{cases} \dfrac{\exp\left(\tilde{\boldsymbol{w}}^{(i)^T} \tilde{\boldsymbol{h}}^{(i)}_{t_k}\right)}{\sum_{j \in \boldsymbol{P}'_{t_k}} \exp\left(\tilde{\boldsymbol{w}}^{(j)^T} \tilde{\boldsymbol{h}}^{(j)}_{t_k}\right)} & \text{if } i \in \boldsymbol{P}'_{t_k} \\ 0 & \text{otherwise} \end{cases}, \tag{3.15}$$

where $\tilde{\boldsymbol{w}}^{(i)} \in \mathbb{R}^{q+2L}$. We use $\alpha^{(i)}_{t_k} \in \mathbb{R}$ to linearly combine the outputs of the LSTM networks as

$$\hat{\boldsymbol{h}}_{t_k} = \sum_{i \in \boldsymbol{P}'} \alpha^{(i)}_{t_k} \bar{\boldsymbol{h}}^{(i)}_{t_k}, \tag{3.16}$$

where $\hat{\boldsymbol{h}}_{t_k} \in \mathbb{R}^q$ is the final output vector of the architecture. Finally, we generate the estimate of the desired signal by

$$\hat{d}_{t_k} = \hat{\boldsymbol{w}}^T_{t_k} \hat{\boldsymbol{h}}_{t_k}, \tag{3.17}$$

where, $\hat{\boldsymbol{w}}_{t_k} \in \mathbb{R}^{q+1}$ is the final regression weights.

Remark 3 *We introduce the most extensive variant of the Tree-LSTM architecture, i.e., all of the 2^L LSTM networks are included. Since we provide an adaptive combination algorithm working on any number of LSTM networks, one can use only a set of desired LSTM networks by exclusively altering the set $\boldsymbol{P}_{t_k}$. As an example, for the length-3 presence-pattern $\boldsymbol{p}_{t_k} = [1,1,1]^T$, one can employ only the LSTM networks with the presence-patterns $[1,0,0]^T$, $[1,1,0]^T$ and $[1,1,1]^T$ instead of the $2^3 = 8$ LSTM networks, which corresponds to the combination of $1-, 2-$ and $3-$step ahead predictors. Therefore, the number of LSTM networks can be reduced to avoid overfitting issues and accelerate the training of the architecture thanks to our adaptive combination algorithm. In addition, one can also use a common weight vector $\tilde{\boldsymbol{w}}$ to calculate $\alpha_{t_k}^{(i)}$'s for all of the LSTM networks instead of assigning a unique weight vector $\tilde{\boldsymbol{w}}^{(i)}$ for each of them.*

3.1.3 Training of the Tree-LSTM Architecture

For the training of the Tree-LSTM architecture, we employ the back-propagation through time algorithm (BPTT) [15] to update the parameters in the architecture, i.e., the combination weights $\tilde{\boldsymbol{w}}^{(i)}$, the final regression coefficients $\hat{\boldsymbol{w}}$, the input weight matrices $\boldsymbol{W}_z^{(j)}$, $\boldsymbol{W}_i^{(j)}$, $\boldsymbol{W}_f^{(j)}$, $\boldsymbol{W}_o^{(j)}$ and the recurrent weight matrices $\boldsymbol{R}_z^{(j)}$, $\boldsymbol{R}_i^{(j)}$, $\boldsymbol{R}_f^{(j)}$, $\boldsymbol{R}_o^{(j)}$ of the LSTM networks.

To write the update equations for the weights of the LSTM networks in a notationally simplified form, we define a new notation for the gates before the nonlinearity is applied, e.g.,

$$\bar{\boldsymbol{o}}_{t_k} = \boldsymbol{W}_o \boldsymbol{x}_{t_k} + \boldsymbol{R}_o \boldsymbol{h}_{t_{k-1}},$$

where $\bar{\boldsymbol{o}}_{t_k} \in \mathbb{R}^q$ is the sum terms before the nonlinearity for the output gate. The terms for the other gates $\bar{\boldsymbol{z}}_{t_k}$, $\bar{\boldsymbol{i}}_{t_k}$, $\bar{\boldsymbol{f}}_{t_k} \in \mathbb{R}^q$, have similar formulations. Then, we first calculate the local gradients as follows:

Architecture	Computational Load
LSTM-ZI	$N(4q^2 + 4qm + 3q)$
LSTM-FI	$N(4q^2 + 4qm + 7q)$
Tree-LSTM (max)	$(N - M)(1 + 2^{L-1}L)(4q^2 + 4qm + 3q)$
Tree-LSTM (min)	$(N - M)(1 + \frac{2^{L(1-r)}L(1-r)}{2})(4q^2 + 4qm + 3q)$

Table 3.1: The number of multiplication operations in the forward pass of the LSTM-ZI, LSTM-FI and the Tree-LSTM architectures to process a sequence with length-N, where M is the number of missing inputs. LSTM-ZI is the network that imputes all-zero vectors for the missing inputs. LSTM-FI represents the LSTM network using forward-filling method and a binary missingness indicator as another feature. Tree-LSTM (max) and Tree-LSTM (min) are the maximum and the minimum computational loads for our architecture.

$$
\begin{aligned}
\delta \boldsymbol{h}_{t_k} &= \frac{\partial E}{\partial \boldsymbol{h}_{t_k}} + \boldsymbol{R}_z^T \delta \boldsymbol{z}_{t_{k+1}} + \boldsymbol{R}_i^T \delta \boldsymbol{i}_{t_{k+1}} \\
&\quad + \boldsymbol{R}_f^T \delta \boldsymbol{f}_{t_{k+1}} + \boldsymbol{R}_o^T \delta \boldsymbol{o}_{t_{k+1}} \\
\delta \boldsymbol{o}_{t_k} &= \delta \boldsymbol{h}_{t_k} \odot g(\boldsymbol{c}_{t_k}) \odot \sigma'(\bar{\boldsymbol{o}}_{t_k}) \\
\delta \boldsymbol{c}_{t_k} &= \delta \boldsymbol{h}_{t_k} \odot \boldsymbol{o}_{t_k} \odot g'(\boldsymbol{c}_{t_k}) + \boldsymbol{f}_{t_{k+1}} \odot \delta \boldsymbol{c}_{t_{k+1}} \\
\delta \boldsymbol{f}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{c}_{t_{k-1}} \odot \sigma'(\bar{\boldsymbol{f}}_{t_k}) \\
\delta \boldsymbol{i}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{z}_{t_k} \odot \sigma'(\bar{\boldsymbol{i}}_{t_k}) \\
\delta \boldsymbol{z}_{t_k} &= \delta \boldsymbol{c}_{t_k} \odot \boldsymbol{i}_{t_k} \odot g'(\bar{\boldsymbol{z}}_{t_k}),
\end{aligned}
$$

where $\delta \boldsymbol{h}_{t_k}$, $\delta \boldsymbol{o}_{t_k}$, $\delta \boldsymbol{c}_{t_k}$, $\delta \boldsymbol{f}_{t_k}$, $\delta \boldsymbol{c}_{t_k}$, $\delta \boldsymbol{i}_{t_k}$, $\delta \boldsymbol{z}_{t_k} \in \mathbb{R}^q$ are the local gradients for corresponding nodes. Based on these local gradients, we calculate the gradients for the input and the recurrent weight matrices of the LSTM networks as

$$
\begin{aligned}
\delta \boldsymbol{W}_\phi &= \sum_{k=0}^{n} \langle \delta \boldsymbol{\phi}_{t_k}, \boldsymbol{x}_{t_k} \rangle \\
\delta \boldsymbol{R}_\phi &= \sum_{k=0}^{n-1} \langle \delta \boldsymbol{\phi}_{t_{k+1}}, \boldsymbol{h}_{t_k} \rangle,
\end{aligned}
$$

where $\phi \in \{z, i, f, o\}$. $\langle \cdot, \cdot \rangle$ is the outer product of two vectors, i.e., $\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle = \boldsymbol{x}_1 \boldsymbol{x}_2^T$.

Remark 4 *The complexity of the new architecture is in the same order of the complexity of the conventional LSTM architectures in terms of the sequence* *length, i.e., N. In Table 3.1, we provide the computational loads in terms of the number of required multiplication operations to process a length-N sequence containing M missing samples for the Tree-LSTM architecture and the conventional algorithms. In Table 3.1, LSTM-ZI is the network that imputes all-zero vectors for the missing inputs. LSTM-FI represents the LSTM network using forward-filling imputation technique together with a binary missingness indicator as another feature. In the vanilla LSTM architecture, there exist four matrix-vector multiplications for the input, i.e., $\boldsymbol{W}\boldsymbol{x}_{t_k}$, four matrix-vector multiplications for the recurrent input, i.e., $\boldsymbol{R}\boldsymbol{h}_{t_{k-1}}$, and three vector-vector multiplications between the gates, i.e.,* (3.5) *and* (3.6)*, which correspond to $4q^2 + 4qm + 3q$ multiplication operations in total. Since the LSTM-FI algorithm extends the feature vector with a binary missingness indicator, the input size is $m + 1$ for this algorithm, which requires* 4 *additional multiplication operations, i.e., $4q^2 + 4qm + 7q$. Since the LSTM-ZI and LSTM-FI algorithms impute the missing inputs with the all-zero vectors and the previous existing input vectors, respectively, these algorithms require N LSTM operations to process a length-N sequence. On the other hand, the Tree-LSTM architecture processes only the existing inputs in the sequence, which requires $N - M$ Tree-LSTM network operations to process the same sequence. For each step of the Tree-LSTM architecture, the main LSTM network processes the each existing input once, however, the number of active leaf LSTM alters based on the presence-pattern of the input. For a length-L window, there exist 2^L different leaf LSTM networks, which require $2^L \times \frac{L}{2}$ LSTM operations in total if all L inputs in the window exist. If one input is missing, the total number of LSTM operations for this window decreases to $2^{L-1} \times \frac{L-1}{2}$. Similarly, the total number of LSTM operations for this window is $2^{L-2} \times \frac{L-2}{2}$ for the case two inputs are missing. Note that the computational load of the Tree-LSTM architecture depends on the distribution of the missing inputs in addition to number of missing inputs. In the worst (also unrealistic) case, the missing and existing inputs are completely separated, the total computational load for our architecture is $(N-M)(1+2^{L-1}L)(4q^2+4qm+3q)$. However, while the distribution of the missing inputs goes to the uniform distribution, the computational load our algorithm rapidly decreases and converges to $(N - M)(1 + \frac{2^{L(1-r)}L(1-r)}{2})(4q^2 + 4qm + 3q)$, where $r = \frac{M}{N}$ is the missingness ratio. Note that the computational load in terms* *of the number of required multiplication operations for our algorithm decreases as the ratio of missing inputs increases. In particular, our architecture is more efficient than the conventional architectures for high values of missingness ratio, r, and small values of the window length L. For example, the computational load for the Tree-LSTM architecture is less than the computational load for the LSTM-ZI architecture when i) $r > 0.5$ if $L = 2$, ii) $r > 0.60$ if $L = 3$ and iii) $r > 0.65$ if $L = 4$ in the optimal case, i.e., the missing inputs are far from each other as much as possible. Since the computational load for the LSTM-FI architecture is higher than the computational load for the LSTM-ZI architecture, our algorithm is also more efficient than LSTM-FI architecture for these parameters.*

3.2 Simulations

In this section, we illustrate the regression performance of the proposed Tree-LSTM architecture under different scenarios with respect to the state-of-the-art algorithms in various real-life datasets. In the first part, we focus on the next value prediction problem over various financial datasets such as the New York stock exchange (NYSE) [23] and the Bitcoin [24]. In the second part, we compare our algorithm with the other architectures on the several real life datasets such as kinematics [18] and California housing [17]. We also illustrate the performance of our architecture in underfitting and overfitting (in terms of the depth of the tree) scenarios.

Throughout this section, "TL" represents the Tree-LSTM architecture. In the first two part, we use the Tree-LSTM architecture with the depth $L = 3$ to have sufficient length input sequences for the leaf LSTM networks by keeping the number of the LSTM networks in a certain limit. The single LSTM network using *i*) the zero imputation and *ii*) the forward-filling imputation with a missingness indicator algorithms are denoted by "ZI" and "FI", respectively.

Since the datasets used in our simulations do not have separate training and test sets, we split the sequences in each dataset such that the first %60 of the

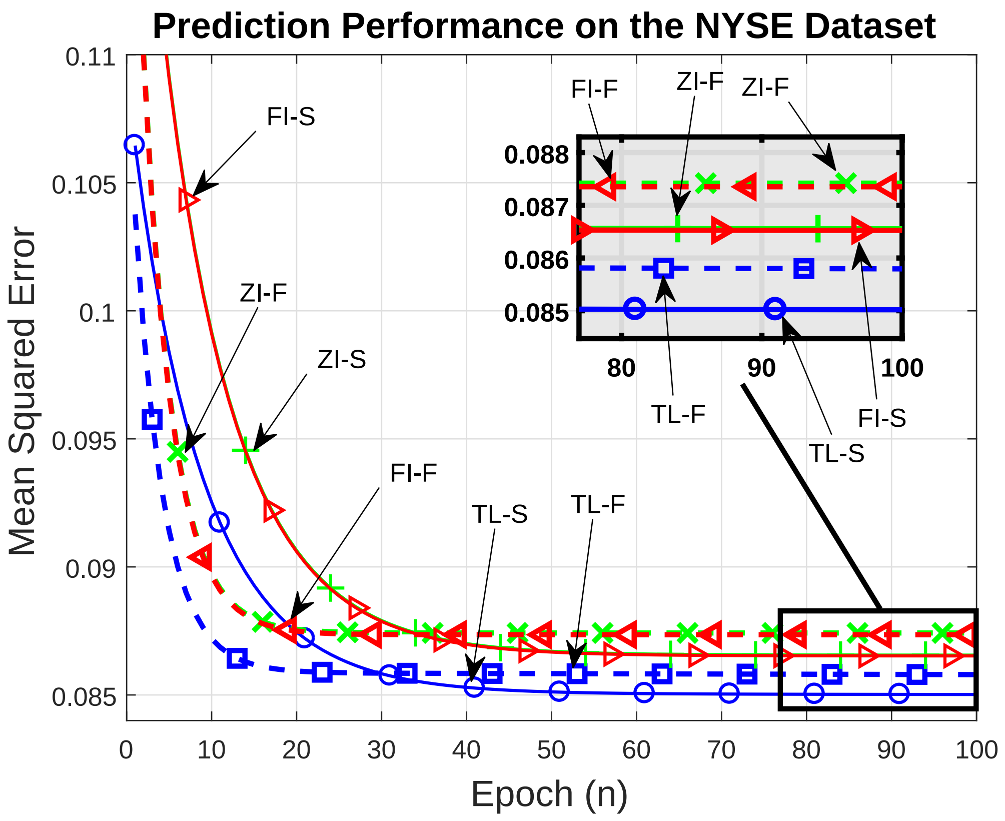


Figure 3.4: Prediction performances for the New York Stock Exchange dataset.

sequence is used for training and the remaining %40 is for test. We also insert missingness to these sets by randomly deleting certain inputs according to a probabilistic model. To evaluate the performance of the algorithms with respect to the different missingness ratios we generate two different sets from each dataset such that randomly selected %30 and %70 of the sequences are missing. In the simulations, these datasets with %30 and %70 missingness ratios are represented by ”-F” (frequent) and ”-S” (sparse), respectively. For the training of the networks, we employ Stochastic Gradient Descent (SGD) algorithm [15] with a constant learning rate. We use 5-fold cross validation for the parameter selection.

3.2.1 Financial Datasets

In this subsection, we evaluate the performances of the Tree-LSTM architectures and the single LSTM architectures employing the zero imputation and the forward-filling techniques. The LSTM network with the forward-filling algorithm uses the existence of the inputs as another feature in the input vectors. Therefore, for a dataset with the input size m, this algorithm has the input size $m + 1$.

We first evaluate the performances of the algorithms on NYSE dataset. The dataset contains the stock prices of 36 different companies over 5651 days (22 years), where we randomly select the third company, i.e., Amer-Brands, for the simulations. For this data, the input is scalar $x_{t_k} \in \mathbb{R}$, i.e., the input size $m = 1$, and the desired output $d_{t_k} \in \mathbb{R}$, where $d_{t_k} = x_{t_{k+1}}$. For the parameter selection, we make a grid search on the number of hidden neurons and the learning rate in the intervals $q = [3, 10]$ and $\eta = [10^{-1}, 10^{-5}]$, respectively. We choose the number of hidden neurons as $q = 10$ and the learning rate as 10^{-3}. All of the weights in the networks are initiated from the Gaussian distribution $\mathcal{N}(0, 10^{-2})$.

In Fig. 3.4, we illustrate the prediction performance of the algorithms in terms of the mean squared error on the test set per epoch. For both experiments conducted with different missing rates, the Tree-LSTM architecture significantly outperforms the other two architectures in terms of the steady-state performance, thanks to its structure providing a unique response to each presence-pattern. The forward-filling imputation with an existence indicator algorithm is slightly better than the zero imputation algorithm in terms of the steady-state error. The Tree-LSTM architecture has also a faster convergence rate compared to the other LSTM architectures. These results show that assigning a unique LSTM network for each presence-pattern and then combining their outputs successfully models the effect of missing data. Our Tree-LSTM architecture outperforms the state-of-the-art architectures in terms of both convergence rate and the steady-state performance in this one-step ahead estimation task.

We also test our algorithm in Bitcoin [24] dataset, which is a more challenging and unstable data compared to NYSE. The dataset contains the price of Bitcoin

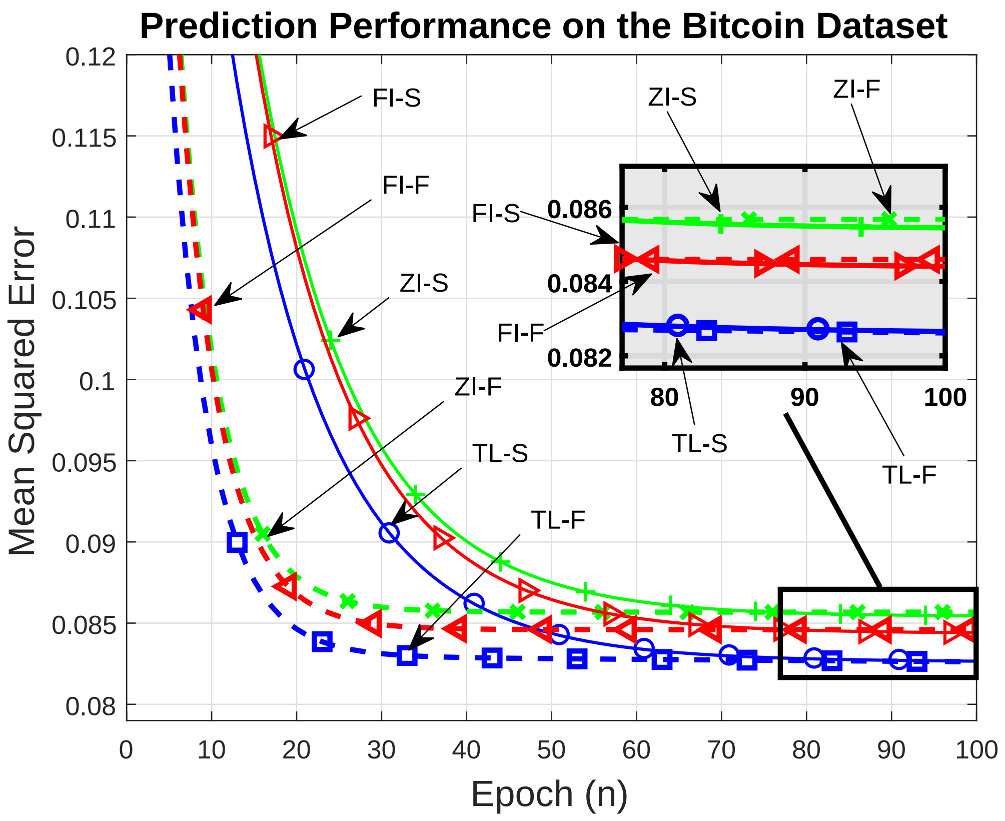


Figure 3.5: Prediction performances for the Bitcoin dataset.

in terms of USD. Similar to NYSE dataset, the input is scalar $x_{t_k} \in \mathbb{R}$, i.e., the input size $m = 1$, and the desired output $d_{t_k} \in \mathbb{R}$, where $d_{t_k} = x_{t_{k+1}}$. For the parameter selection, we make a grid search on the number of hidden neurons and the learning rate in the intervals $q = [3, 10]$ and $\eta = [10^{-1}, 10^{-5}]$, respectively. We choose the number of hidden neurons as $q = 10$ and the learning rate as 10^{-4}. We initiate the weights from the distribution $\mathcal{N}(0, 10^{-2})$.

In Fig. 3.5, we demonstrate the prediction performance of the algorithms in terms of the mean squared error on the test set per epoch. For both missing rates, our architecture has a better steady-state performance compared to the other two algorithms. In terms of the convergence rate, all of the architectures have similar performances in this dataset. The results show that the Tree-LSTM architecture significantly outperforms the other algorithms thanks to its novel structure, which separately processes each pattern and adaptively combines them by incorporating the missingness information.

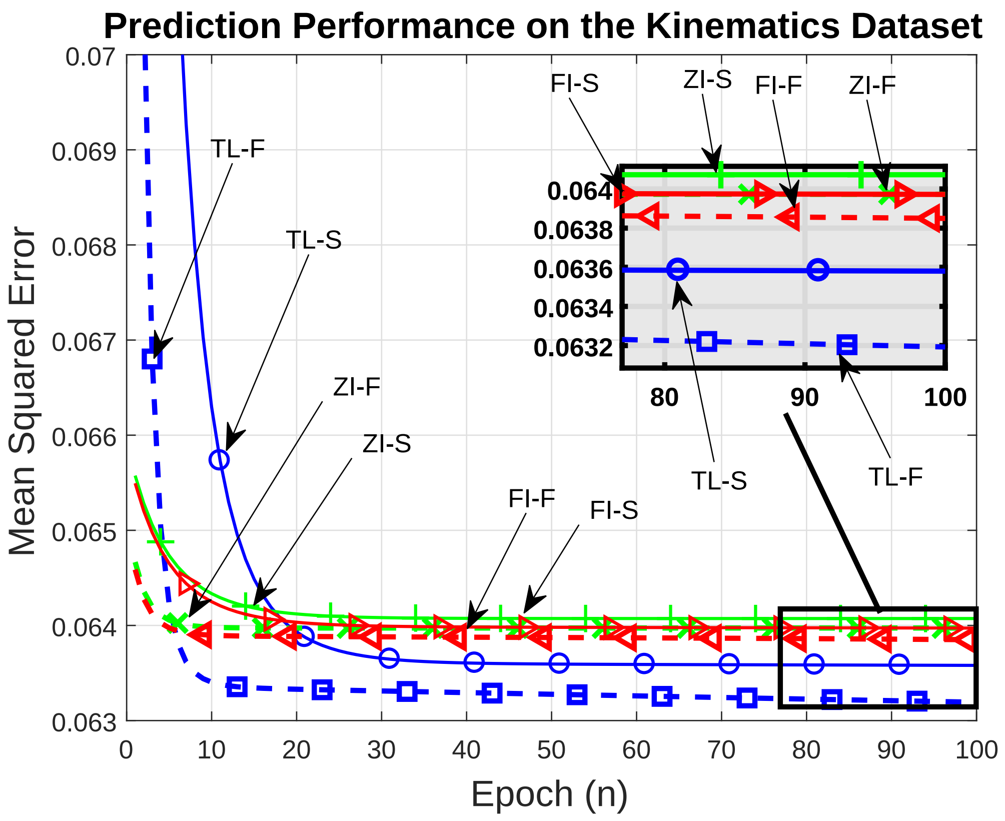


Figure 3.6: Regression performances for the Kinematics dataset.

3.2.2 Real Life Datasets

In this subsection, we compare the performances of the algorithms over several real life datasets under different missing rates. The LSTM network with the forward-filling algorithm uses the existence of the inputs as another feature in the input vectors. Therefore, for a dataset with the input size m, this algorithm has the input size $m+1$. We test our algorithms on kinematics [18] and California housing [17] datasets. These datasets contain an input vector sequence and the corresponding desired signal for each time step.

- Kinematic dataset is a simulation of 8-link all-revolute robotic arm, where the aim is to predict the distance of the effector from the target. The original input vector size $m = 8$ and we set the number of hidden neurons $q = 8$ for both LSTM and TG-LSTM networks. For the SGD algorithm, we select the constant learning rate $\eta = 10^{-4}$ from the interval $[10^{-5}, 10^{-2}]$ using the cross-validation.

- California Housing dataset contains the house prices in the California area and the aim is to estimate the median of these house prices. The input vector $\boldsymbol{x}_{t_k} \in \mathbb{R}^8$. We set the number of hidden neurons $q = 8$, and the constant learning rate $\eta = 10^{-4}$ from the interval $[10^{-5}, 10^{-2}]$.

In Fig. 3.6 and Fig. 3.7, we illustrate the regression performance of the algorithms in terms of the mean squared error per epoch for kinematics and California housing datasets, respectively. In these simulations, the Tree-LSTM architecture captures the sequential pattern of the data with a faster convergence rate compared to the other two algorithms. The LSTM architecture using forward-filling imputation method is slightly better than the LSTM architecture using the zero imputation technique in terms of the steady-state error. However, our architecture significantly outperforms the other two methods in terms of the steady-state error. These results show that our algorithm successfully handles the effect of missing samples and models the underlying structure by using only the existing data compared to the other methods.

In Fig. 3.8, we illustrate the effect of the parameter tree depth L on the performance of our architecture over the California housing dataset. For this simulation, we use $\{1, 2, 3, 4\}$ as the depth of the tree, the number of hidden neurons $q = 8$, and the learning rate $\eta = 10^{-4}$. When we compare the performance of the algorithm with the same depth for different missing rates, the steady-state performance on the frequent data is significantly higher than the performance on the sparse data. For the same missing rates and different depth of the tree, the Tree-LSTM architecture with the depths $L = 2$ and $L = 3$ achieve the highest performances in terms of the steady-state performance. These two networks outperform even the Tree-LSTM architecture with the depth $L = 4$ since the size of the data is not sufficient for learning to combine the outputs of 16 LSTM networks. Here, we observe modelling capacity vs. trainability trade-off [25] of the recurrent neural networks, i.e., while the number of the parameters of an RNN

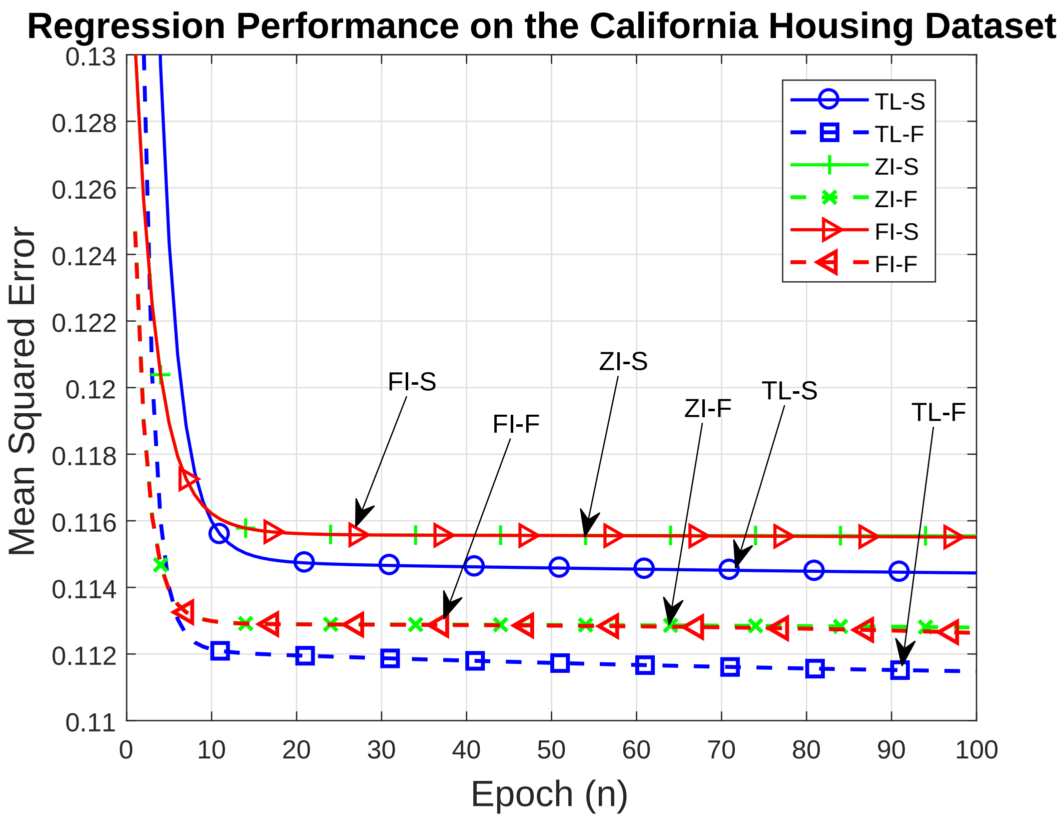


Figure 3.7: Regression performances for the California housing dataset.

increases, its potential modelling capability increases as well, however, the training process becomes more difficult and the RNN may not achieve its potential steady-state performance. Therefore, $L = 2$ and $L = 3$ are the optimal choices for this dataset in terms of the steady-state performance.

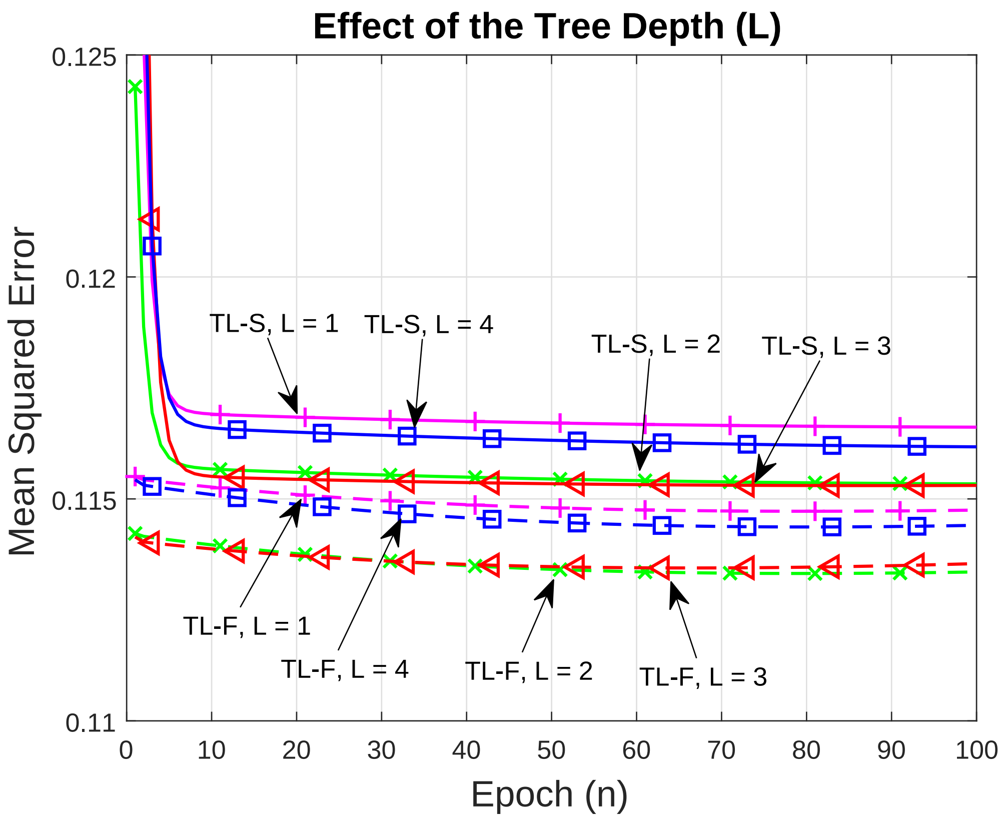


Figure 3.8: The effect of the tree depth on the regression performance the Tree-LSTM architecture for the California housing dataset.

Chapter 4

Conclusion

We studied nonlinear classification and regression problems on variable length non-uniformly sampled sequential data in a supervised framework. In chapter 2, we introduced a novel LSTM architecture, namely, TG-LSTM. In the TG-LSTM architecture, we incorporate the sampling time information to enhance the performance for applications involving non-uniformly sampled sequential data. In particular, the input, forget and output gates of the LSTM architecture are scaled by these time gates using the sampling intervals. When the time intervals do not convey information related to the underlying task, our architecture simplifies to the vanilla LSTM architecture. The TG-LSTM architecture has a wide range of application areas since it can generate output at each time step as well as at the end of the input sequence unlike the other state of the art methods. We achieve significant performance gains in various applications, while our approach has nearly the same computational complexity with the classical LSTM architecture. In our simulations, covering several different classification and regression tasks, we demonstrate significant performance gains achieved by the introduced LSTM architecture with respect to the conventional LSTM architectures [11], [5] over several synthetic and real life datasets. In chapter 3, we introduce a novel LSTM network based on a tree architecture, which combines the outputs of a variable number of LSTM networks for sequential regression tasks. Our architecture assigns a unique LSTM network based on the missingness information. Particularly, our architecture grants each LSTM network the capability of modelling an input sequence with a specific missingness pattern and learns to combine the outputs of these LSTM networks. By this way, the proposed algorithm incorporates the missingness information by selecting the particular LSTM networks based on the existence of the certain input patterns. Therefore, it uses both the input signal itself as well as the missingness pattern to circumvent the effects of the missing samples without making any statistical or artificial assumptions on the underlying data. In addition, our architecture keeps the computational load in terms of number of multiplication operations less than the computational load of the conventional algorithms especially when the number of missing samples is high. Through an extensive set of experiments, we illustrate significant performance gains compared to the state-of-the-art methods in several regression tasks.

Bibliography

[1] J. J. Benedetto and H. C. Wu, "Nonuniform sampling and spiral MRI reconstruction," in *Wavelet Applications in Signal and Image Processing VIII*, vol. 4119, pp. 130–142, International Society for Optics and Photonics, 2000.

[2] R. Vio, T. Strohmer, and W. Wamsteker, "On the reconstruction of irregularly sampled time series," *Publications of the Astronomical Society of the Pacific*, vol. 112, no. 767, p. 74, 2000.

[3] F. Eng and F. Gustafsson, "Algorithms for downsampling non-uniformly sampled data," in *Signal Processing Conference, 2007 15th European*, pp. 1965–1969, IEEE, 2007.

[4] Z. C. Lipton, D. C. Kale, C. Elkan, and R. Wetzell, "Learning to diagnose with LSTM recurrent neural networks," *arXiv preprint arXiv:1511.03677*, 2015.

[5] D. Neil, M. Pfeiffer, and S.-C. Liu, "Phased lstm: Accelerating recurrent network training for long or event-based sequences," in *Advances in Neural Information Processing Systems*, pp. 3882–3890, 2016.

[6] J. L. Rojo-Álvarez, C. Figuera-Pozuelo, C. E. Martínez-Cruz, G. Camps-Valls, F. Alonso-Atienza, and M. Martínez-Ramón, "Nonuniform interpolation of noisy signals using support vector machines," *IEEE Transactions on Signal Processing*, vol. 55, no. 8, pp. 4116–4126, 2007.

[7] L. Xie, Y. Liu, H. Yang, and F. Ding, "Modelling and identification for non-uniformly periodically sampled-data systems," *IET Control Theory & Applications*, vol. 4, no. 5, pp. 784–794, 2010.

[8] J. Sjölund, A. Eklund, E. Özarslan, and H. Knutsson, "Gaussian process regression can turn non-uniform and undersampled diffusion MRI data into diffusion spectrum imaging," in *Biomedical Imaging (ISBI 2017), 2017 IEEE 14th International Symposium on*, pp. 778–782, IEEE, 2017.

[9] A. C. Singer, G. W. Wornell, and A. V. Oppenheim, "Nonlinear autoregressive modeling and estimation in the presence of noise," *Digital signal processing*, vol. 4, no. 4, pp. 207–221, 1994.

[10] N. Cesa-Bianchi and G. Lugosi, *Prediction, Learning, and Games.* Cambridge University Press, 2006.

[11] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, 2017.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] Z. C. Lipton, D. C. Kale, and R. Wetzel, "Modeling missing data in clinical time series with RNNs," *Machine Learning for Healthcare*, 2016.

[14] J. Pan and B. Hu, "Robust occlusion handling in object tracking," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pp. 1–8, IEEE, 2007.

[15] H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, vol. 5. GMD-Forschungszentrum Informationstechnik, 2002.

[16] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[17] C. E. Rasmussen, R. M. Neal, G. Hinton, D. Camp, M. Revow, Z. Ghahramani, R. Kustra, and R. Tibshirani, "Delve data sets." `http://www.cs.toronto.edu/~delve/data/datasets.html`. Accessed: 2018-03-7.

[18] L. Torgo, “Regression data sets.” `http://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html`. Accessed: 2018-03-8.

[19] J. Alcala-Fdez, A. Fernandez, J. Luengo, J. Derrac, S. García, L. Sánchez, and F. Herrera, “KEEL data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework,” *Journal of Multiple-Valued Logic and Soft Computing*, vol. 17, no. 2-3, pp. 255–287, 2011.

[20] M. Lichman, “UCI machine learning repository.” `http://archive.ics.uci.edu/ml`, 2013. Accessed: 2018-03-5.

[21] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[22] Z. C. Lipton, D. Kale, and R. Wetzel, “Directly modeling missing data in sequences with RNNs: Improved classification of clinical time series,” in *Machine Learning for Healthcare Conference*, pp. 253–270, 2016.

[23] “New york stock exchange dataset.” `http://www.cs.technion.ac.il/~rani/portfolios/NYSE_Dataset.htm`. Accessed: 2018-03-8.

[24] “Public rest api for binance.” `https://github.com/binance-exchange/binance-official-api-docs/blob/master/rest-api.md`. Accessed: 2018-03-15.

[25] J. Collins, J. Sohl-Dickstein, and D. Sussillo, “Capacity and trainability in recurrent neural networks,” *arXiv preprint arXiv:1611.09913*, 2016.