UNIVERSIDAD MICHOACANA
DE SAN NICOLÁS DE HIDALGO

FACULTAD DE INGENIERÍA ELÉCTRICA
DIVISIÓN DE ESTUDIOS DE POSGRADO

Cloud-based Implementation of Distributed Machine Learning Algorithms for Time Series Forecasting

TESIS

Que para obtener el grado de

DOCTOR EN CIENCIAS EN INGENIERÍA ELÉCTRICA

Presenta

José Luis GARCÍA NAVA

Director de Tesis
Dr. Juan José FLORES ROMERO

Co-Director de Tesis
Dr. Félix CALDERÓN SOLORIO

Morelia, Michoacán. Febrero de 2023

CLOUD-BASED IMPLEMENTATION OF DISTRIBUTED MACHINE LEARNING ALGORITHMS FOR TIME SERIES FORECASTING

Los Miembros del Jurado de Examen de Grado aprueban la **Tesis de Doctorado en Ciencias en Ingeniería Eléctrica, Opción en Sistemas de Computacionales** de **José Luis García Nava.**

Dr. José Antonio Camarena Ibarrola
Presidente del Jurado

Dr. Juan José Flores Romero
Director de Tesis

Dr. Félix Calderón Solorio
Co-director

Dr. José Ortíz Bejar
Vocal

Dr. Carlos Alberto Lara Álvarez
Revisor Externo (CIMAT Zacatecas)

Dr. J. Aurelio Medina Rios
Jefe de la División de Estudios de Posgrado de la Facultad de Ingeniería Eléctrica. UMSNH (Por reconocimiento de firmas)

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO
Febrero 2023

Abstract

Time series forecasting is essential to decision-making in crucial domains and machine learning algorithms have consolidated as a predominant approach to it. When developing machine-learning solutions, the capability to produce fast results is crucial to progressively refine a model within adequate time frames. Having to deal with Big Data makes this situation more challenging as massive data demands equally massive computing power. In this context, cloud computing has been globally used to accelerate the development of forecasting solutions based on machine learning. The aim of this Thesis is to achieve state-of-the-art results in time series forecasting (in terms of accuracy) with remarkably increased speed in model training by leveraging proven machine learning algorithms and advanced cloud computing resources. The main contributions of this Thesis are as follows. (1) A cloud-based implementation of the three machine learning architectures that account for most of the recent international research on time series forecasting. (2) The first TPU-based implementation of a canonical Transformer architecture for time series forecasting reported in the literature. (3) This architecture reports the lowest training times found in the literature on two predominant, standardized datasets for global, multi-horizon time series forecasting, while achieving good predictive performance on those datasets when compared to state-of-the-art models.

Keywords: deep learning, Long Short-Term Memory network, Encoder-Decoder with Attention network, Transformer network, Tensor Processing Unit.

Resumen

El pronóstico de series de tiempo es esencial para importantes áreas de conocimiento y los algoritmos de aprendizaje automático se han consolidado como un método predominante para realizarlo. Al desarrollar soluciones de aprendizaje automático, la capacidad para producir resultados rápidos es crucial para el refinamiento progresivo de modelos dentro de horizontes de tiempo adecuados. El trabajo con datos masivos complica aún más esta situación ya que exige el uso de recursos de cómputo igualmente masivos. En este contexto, la computación en la nube ha sido utilizada globalmente para acelerar el desarrollo de soluciones de pronóstico basadas en aprendizaje automático. El objetivo de esta Tesis es lograr resultados al nivel del estado del arte en pronóstico de series de tiempo (en términos de exactitud) con gran velocidad en el entrenamiento de modelos mediante el uso de algoritmos de aprendizaje automático comprobados y avanzados recursos de computación en la nube. Las principales contribuciones de esta Tesis son tres. (1) La implementación en la nube de las tres arquitecturas de aprendizaje automático más utilizadas recientemente en la investigación internacional de pronóstico de series de tiempo. (2) La primera implementación basada en Unidad de Procesamiento Tensorial de una arquitectura Transformer canónica para pronóstico de series de tiempo reportada en la literatura. (3) Esta arquitectura reporta los tiempos de entrenamiento más bajos en la literatura para dos conjuntos de datos que son utilizados de manera predominante para el pronóstico global y multi-horizonte de series de tiempo. Simultáneamente, esta arquitectura logra un buen desempeño predictivo en comparación con el estado del arte.

Palabras clave: aprendizaje profundo, red con memoria larga de corto plazo, red codificador-decodificador con atención, red Transformer, Unidad de Procesamiento Tensorial.

Agradecimientos

Agradezco en primer término al Dr. Juan José Flores Romero por haber sido profesor y guía a lo largo de mis estudios de posgrado. Su apoyo continuo y riguroso a mi formación sólo es comparable en importancia a la confianza que siempre ha brindado a las ideas provenientes del lado derecho de mi cerebro. Me siento muy afortunado por haberme beneficiado de esa combinación que considero poco común en una escuela de ingeniería. Aunque no tuve la fortuna de ser su alumno, agradezco al Dr. Félix Calderón Solorio por brindarme su apoyo en todos los momentos en que lo necesite. Muchas ocasiones llegaba al salón de clases apenas terminada la clase del Dr. Calderón y encontraba una extraña paz, que me ayudaba a comenzar el día, al contemplar el orden impecable de sus apuntes en el pizarrón. Agradezco al Dr. José Antonio Camarena Ibarrola no sólo por sus clases y sus consejos sino de manera muy especial por aquella ocasión en que una tarea de su clase de Matemáticas Discretas me hizo entender que debía elegir entre reorganizar completamente mi pensamiento o abandonar el curso propedeútico, y con ello el posgrado. Afortunadamente logré definirme por la primera alternativa. Agradezco al Dr. Jaime Cerda Jacobo por ser también mi profesor durante el doctorado. Agradezco al Dr. José Ortiz Bejar y al Dr. Carlos Alberto Lara Álvarez por su invaluable apoyo como miembros de la mesa sinodal de mi examen de grado.

Un agradecimiento especial lo merecen todos los integrantes de la comunidad de la DEPFIE que hicieron muy agradables los años que he pasado en esta escuela: los profesores e investigadores, mis compañeros alumnos, la Sra. Lupita y todo el personal administrativo y de apoyo, de los cuales siempre recibí ayuda y amabilidad.

Agradezco especialmente a la División de Distribución Centro-Occidente de la Comisión Federal de Electricidad y a sus profesionales con los que tuve la oportunidad de colaborar por su generosidad al permitirme utilizar sus

conjuntos de datos en la elaboración de mis experimentos.

De la manera más especial agradezco a mi esposa María del Rosario y a mis hijas María Paz y María de la Luz por haber sido increíblemente pacientes y alentadoras durante mis estudios de posgrado. Igualmente agradezco a mi madre Rosa María y a mis hermanos Juan y María Guadalupe por haber estado siempre presentes, incluso en la distancia física.

Esta investigación contó con el apoyo del Consejo Nacional de Ciencia y Tecnología a través de una Beca Nacional otorgada a José Luis García Nava bajo el CVU número 737505.

Publications

Part of the research produced during the elaboration of this Thesis has appeared previously in the following publications.

Journal Articles

- **García-Nava J-Luis**, Flores JJ, Tellez VM, Calderon F. *Fast Training of a Transformer for Global Multi-horizon Time Series Forecasting on Tensor Processing Units.* **The Journal of Supercomputing** 2022; published online. https://doi.org/10.1007/s11227-022-05009-x

- Flores JJ, **García-Nava J-Luis**, Cedeno Gonzalez JR, Tellez VM, Calderon F, Medrano A. *A Machine-Learning Pipeline for Large-Scale Power-Quality Forecasting in the Mexican Distribution Grid.* **Applied Sciences** 2022; 12(17):8423. https://doi.org/10.3390/app12178423

Contents

Abstract iii

Resumen v

Agradecimientos viii

Publications ix

List of Figures xv

List of Tables xvii

Acronyms xix

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Hypothesis . . . 5
- 1.3 Objectives . . . 5
- 1.4 Overview of Thesis . . . 6

2 Machine Learning for Time Series Forecasting 9
- 2.1 Time Series Forecasting Fundamentals . . . 9
  - 2.1.1 Problem definition . . . 9
  - 2.1.2 One- vs. Multiple-step ahead Forecasting . . . 10
  - 2.1.3 Local vs. Global Forecasting . . . 13
  - 2.1.4 Point vs. Probabilistic Forecasting . . . 15
- 2.2 Time Series Forecasting Methods . . . 15
  - 2.2.1 Exponential Smoothing Models . . . 16
  - 2.2.2 ARIMA Models . . . 16

2.2.3 Tree-based Models 17
2.2.4 Convolutional Neural Networks 17
2.2.5 Recurrent Neural Networks 19
2.2.6 Attention-based Models 19
2.2.7 Hybrid Models 21
2.2.8 Other Forecasting Models 22
2.3 Neural Networks Implemented in Thesis 22
2.3.1 Long Short-Term Memory Network 22
2.3.2 Sequence-to-Sequence with Attention Network 24
2.3.3 Transformer Network 27

3 Distributed Machine Learning on Cloud Computing 33
3.1 DML Fundamentals 33
3.1.1 Background 34
3.1.2 DML Features 36
3.1.3 DML Frameworks 42
3.1.4 Additional Research Directions 44
3.2 Cloud Computing for DML 45
3.2.1 Cloud Computing Fundamentals 45
3.2.2 DML on the IaaS Service Model 49
3.2.3 DML on the PaaS Service Model 51
3.2.4 DML on the SaaS Service Model 52
3.3 Tensor Processing Units 52
3.3.1 Background 53
3.3.2 TPU Basic Characteristics 53
3.3.3 TPU-based Cloud Computing Pattern 56

4 Deep Multi-Sequence LSTM Architecture 59
4.1 Related Work 59
4.2 Experimental Study 63
4.2.1 Dataset 63
4.2.2 Architecture Description 67
4.2.3 Model Parametrization 68
4.3 Results and Discussion 71
4.3.1 Baseline 72
4.3.2 Metrics 73
4.3.3 Forecasting Accuracy 74
4.3.4 Computation Time 78

4.3.5 Predictive Performance Degradation 79

5 Sequence-to-Sequence with Attention Architecture 87
5.1 Related Work 87
5.2 Experimental Study 89
5.2.1 Dataset 89
5.2.2 Architecture Description 89
5.2.3 Model Parametrization 92
5.3 Results and Discussion 95
5.3.1 Baseline 95
5.3.2 Metrics 95
5.3.3 Forecasting Accuracy 96
5.3.4 Computation Time 100
5.3.5 Predictive Performance Degradation 101

6 Transformer Architecture for Local Forecasting 107
6.1 Related Work 107
6.2 Experimental Study 109
6.2.1 Dataset 109
6.2.2 Architecture Description 109
6.2.3 Model Parametrization 112
6.3 Results and Discussion 115
6.3.1 Baseline 116
6.3.2 Metrics 116
6.3.3 Forecasting Accuracy 116
6.3.4 Computation Time 120
6.3.5 Predictive Performance Degradation 121

7 Transformer Architecture for Global Forecasting 127
7.1 Related Work 127
7.2 Experimental Study 128
7.2.1 Datasets and Data Preprocessing 128
7.2.2 Architecture Description 133
7.2.3 Model Parametrization 133
7.2.4 Inference 138
7.3 Results and Discussion 139
7.3.1 Benchmark Models 139
7.3.2 Forecasting Accuracy 140

7.3.3 Computation Time 142

8 Conclusions and Future Work **145**

8.1 Conclusions 145

8.2 Contributions 146

8.3 Future Work 148

References **151**

List of Figures

2.1 One-step ahead forecasting as a generic regression problem. . . 11
2.2 Iterative strategy for multiple-step ahead forecasting. 12
2.3 Global, multi-step time series forecasting. 13
2.4 CNN-based temporal encoder architecture. 18
2.5 RNN-based temporal encoder architecture. 20
2.6 Attention-based temporal encoder architecture. 20
2.7 Detailed schematic of the LSTM block. 23
2.8 Sequence-to-Sequence with Attention Network. 25
2.9 Transformer architecture. 28
2.10 Multi-head Attention. 29
2.11 Scaled Dot-product Attention. 29

3.1 Cloud computing characteristics and models. 46
3.2 Systolic data flow of the MXU. 54
3.3 Comparison of the float32, bfloat16, and float16 formats. . . . 55
3.4 Cloud-based TPU Computing Pattern. 56

4.1 Active power for an urban substation meter in southeast Morelia. 65
4.2 Deep Multi-Sequence LSTM Network Architecture. 66
4.3 DMSLSTM Learning Rate Schedule. 70
4.4 DMSLSTM Training Loss. 70
4.5 DMSLSTM vs. ARIMA vs. Real for Day 1. 74
4.6 DMSLSTM vs. ARIMA vs. Real for Day 2. 75
4.7 DMSLSTM vs. ARIMA vs. Real for Day 3. 75
4.8 DMSLSTM vs. ARIMA vs. Real for Day 4. 75
4.9 DMSLSTM vs. ARIMA vs. Real for Day 5. 76
4.10 DMSLSTM vs. ARIMA vs. Real for Day 6. 76
4.11 DMSLSTM vs. ARIMA vs. Real for Day 7. 76

4.12 DMSLSTM metrics for 28 rolling-day predictions. 82
4.13 DMSLSTM metrics for 672 rolling-hour predictions. 83

5.1 Encoder-Decoder with attention Stacked LSTM Network Architecture. 90
5.2 EDSLSTM Learning Rate Schedule. 94
5.3 EDSLSTM Training Loss. 94
5.4 EDSLSTM vs. ARIMA vs. Real for Day 1. 96
5.5 EDSLSTM vs. ARIMA vs. Real for Day 2. 97
5.6 EDSLSTM vs. ARIMA vs. Real for Day 3. 97
5.7 EDSLSTM vs. ARIMA vs. Real for Day 4. 97
5.8 EDSLSTM vs. ARIMA vs. Real for Day 5. 98
5.9 EDSLSTM vs. ARIMA vs. Real for Day 6. 98
5.10 EDSLSTM vs. ARIMA vs. Real for Day 7. 98
5.11 EDSLSTM metrics for 28 rolling-day predictions. 102
5.12 EDSLSTM metrics for 672 rolling-hour predictions. 103

6.1 BSCTRFM Network Architecture for Local Forecasting. 110
6.2 BSCTRFM Learning Rate Schedule. 114
6.3 BSCTRFM Training Loss. 114
6.4 BSCTRFM vs. ARIMA vs. Real for Day 1. 116
6.5 BSCTRFM vs. ARIMA vs. Real for Day 2. 117
6.6 BSCTRFM vs. ARIMA vs. Real for Day 3. 117
6.7 BSCTRFM vs. ARIMA vs. Real for Day 4. 117
6.8 BSCTRFM vs. ARIMA vs. Real for Day 5. 118
6.9 BSCTRFM vs. ARIMA vs. Real for Day 6. 118
6.10 BSCTRFM vs. ARIMA vs. Real for Day 7. 118
6.11 BSCTRFM metrics for 28 rolling-day predictions. 122
6.12 BSCTRFM metrics for 672 rolling-hour predictions. 123

7.1 Three time series from the `electricity` dataset. 129
7.2 Three time series from the `traffic` dataset. 130
7.3 Data setup for BSCTRFM at training/inference time. 132
7.4 BSCTRFM Network Architecture for Global Forecasting. . . . 134
7.5 BSCTRFM learning rate schedule/training loss on `electricity`. 136
7.6 BSCTRFM learning rate schedule/training loss on `traffic`. . 137

List of Tables

4.1 Dataset statistics. 64
4.2 DMSLSTM parameters. 69
4.3 ARIMA parametrization on BigQuery ML. 72
4.4 DMSLSTM Vs. ARIMA Comparison. 77
4.5 DMSLSTM Vs. ARIMA Computation Time Comparison. . . . 79
4.6 DMSLSTM Summarized Results. 84

5.1 EDSLSTM parameters. 93
5.2 EDSLSTM Vs. ARIMA Comparison. 99
5.3 EDSLSTM Vs. ARIMA Computation Time Comparison. . . . 100
5.4 EDSLSTM Summarized Results. 104

6.1 BSCTRFM parameters. 113
6.2 BSCTRFM Vs. ARIMA Comparison. 119
6.3 BSCTRFM Vs. ARIMA Computation Time Comparison. . . . 121
6.4 BSCTRFM Summarized Results. 125

7.1 Dataset statistics and BSCTRFM parameters. 135
7.2 Forecasting accuracy of BSCTRFM vs. state-of-the-art. 141
7.3 Computation time comparison. 143

Acronyms

API Application Programming Interface. 50, 56, 57, 147, 149

ARCH Autoregressive Conditional Heteroskedasticity. 22

ARIMA Auto Regressive Integrated Moving Average. xvii, 16, 17, 72–74, 78, 79, 84, 95, 96, 105, 116, 120, 126, 145

ASIC Application-Specific Integrated Circuit. 35, 50, 52, 53

AWS Amazon Web Services. 50, 52

BSCTRFM Basic Transformer. xvii, 105, 107, 109, 111, 112, 115, 116, 120, 121, 124, 126–128, 131, 133, 135, 138–140, 142–146, 148, 149

CC Cloud Computing. 33, 45, 53, 57

CNN Convolutional Neural Network. 3, 17–19, 21, 44, 61, 88

CPU Central Processing Unit. 35, 37, 50, 53, 55, 57, 71, 85, 95, 100, 105, 115, 120, 126, 138

CUDA Compute Unified Device Architecture. 50

CV Computer Vision. 53

DL Deep Learning. 3, 5, 6, 14, 15, 35, 36, 50, 53, 55–58, 84, 87, 100, 107, 126, 145

DML Distributed Machine Learning. 6, 33–39, 41–45, 50, 52, 57, 146, 148

DMSLSTM Deep Multi-Sequence Long Short-Term Memory. 59, 63, 67, 68, 71–74, 78–81, 84, 85, 89, 95, 101, 105, 109, 116, 121, 126, 145, 146, 148

DNN Deep Neural Network. 53

EDSLSTM Encoder-Decoder with attention Stacked Long Short-Term Memory. 85, 87, 89, 91, 92, 95, 96, 100, 101, 104, 105, 109, 116, 120, 121, 126, 145, 146, 148

FPGA Field-Programmable Gate Array. 37, 44, 50, 52

GAN Generative Adversarial Network. 108

GARCH Generalized Autoregressive Conditional Heteroskedasticity. 22

GBDT Gradient Boosted Decision Tree. 17

GC Google Cloud. 36, 50, 52–54, 56, 72, 84, 105, 126

GPT Generative Pre-trained Transformer. 36

GPU Graphics Processing Unit. 34–37, 50, 52, 53, 55

gRPC Google Remote Procedural Calls. 56

GRU Gated Recurrent Unit. 88

IaaS Infrastructure-as-a-Service. 48–50, 53

IoT Internet of Things. 45

IT Information Technology. 47

JSON JavaScrip Object Notation. 50

LSTM Long Short-Term Memory. 3, 6, 21–24, 27, 44, 58–62, 67, 68, 78, 84, 87, 88, 91, 105, 108, 148

MAE Mean Absolute Error. 73, 74, 81, 84, 95, 96, 101, 104, 116, 120, 121, 124, 125

MAPE Mean Absolute Percentage Error. 73

MHA Multi-Head Attention. 30, 31, 108, 111, 112, 126

ML Machine Learning. 3–5, 9, 21, 22, 34, 35, 37, 38, 43–45, 50–53, 57, 59, 60, 79, 80, 145, 147–150

MLaaS Machine Learning-as-a-Service. 51, 52

MLP Multi-Layer Perceptron. 21, 22

MPI Message Passing Interface. 50

MSE Mean Squared Error. 69, 71, 92, 95, 115, 138

MXU Matrix Multiplier Unit. 53–55

NCCL NVIDIA Collective Communication Library. 44, 50, 52

NIST National Institute of Standards and Technology. 45

NLP Natural Language Processing. 36, 53, 87, 107

PaaS Platform-as-a-Service. 48, 50, 51

RMSE Root Mean Squared Error. 71, 73, 74, 81, 84, 92, 95, 96, 101, 104, 115, 116, 120, 121, 124, 125, 138

RNN Recurrent Neural Network. 3, 19, 22, 24, 26, 44, 59–62, 88, 140

SaaS Software-as-a-Service. 48, 52, 72

Seq2Seq Sequence to Sequence. 6, 22, 24, 26, 27, 61, 85, 87, 88, 91, 92, 107, 108, 140

SGD Stochastic Gradient Descent. 35, 38

SLDB Supervised-Learning DataBase. 67, 89, 109, 111

SMAPE Symmetric Mean Absolute Percentage Error. 73, 74, 81, 84, 95, 96, 101, 104, 116, 120, 121, 124, 125

SSH Secure Shell. 56

TFX TensorFlow Extended. 149

TPU Tensor Processing Unit. 6, 33, 35–37, 50, 52–58, 71, 78, 84, 85, 95, 100, 104, 105, 115, 120, 125, 126, 138, 142, 143, 145–147, 149

TSF Time Series Forecasting. 3–7, 9, 10, 13, 15–17, 19, 21, 22, 24, 26, 27, 31, 53, 59, 60, 62, 85, 87, 88, 105, 107, 108, 112, 127, 128, 139, 140, 143–145, 147–149

VM Virtual Machine. 50, 51, 54–57

VT Vanilla Transformer. 27, 30

XLA Accelerated Linear Algebra. 55

Chapter 1

Introduction

1.1 Motivation

Time Series Forecasting (TSF) is essential to decision-making in crucial domains such as science, engineering, business, and economics. Over the last two decades, Machine Learning (ML) algorithms consolidated as a practical approach to TSF. Two factors contributed to this situation. First, the increasing size, quality, and availability of time series historical datasets in many research fields. Second, the sustained development of powerful ML-oriented computing frameworks. Additionally, progressive advances in Deep Learning (DL) provided neural network architectures able to produce state-of-the-art results in TSF, including Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM) networks, attention-based mechanisms and, among the latter category, the novel neural network referred to as Transformer.

We lived immersed in Big Data and the advent of cloud computing is in part responsible for the drastic increment in information we experience every day: the immense amount of interactions on multiple social networks, the growing number of services based on mobile applications, and the continuous data-gathering performed by millions of Internet-of-Things devices are only part of an incessant data stream that challenges any analytics or forecasting system. In a parallel direction, cloud computing also promotes the development and utilization of important resources that allow us to efficiently process information at a very large scale, such as distributed computing, high-performance databases, high-speed networking, and DL-specific

computing frameworks. However, in order to make appropriate use of such resources, a robust research work aimed to design, implement, and test integrated software and hardware architectures as cloud-native solutions must be conducted inside academic institutions, as it has been done for many years in private IT companies.

When it comes to developing an ML-based forecasting solution, the capability to produce fast results is crucial to progressively refine a model within adequate time frames. Then, serious efforts must be conducted to accelerate the ML life-cycle, especially at the model training stage. Having to deal with Big Data makes this situation even more challenging as massive data demands equally massive computing power. Consequently, the benefits associated with ML operations come to any contemporary data environment only at the price of deploying computing resources powerful enough to allow applications to produce solutions that are not only accurate but also extremely fast.

This Thesis proposes an implementation of ML-based algorithms for TSF that leverages cloud-based resources like production-level computing frameworks, distributed computing tools, and high-performance data management to significantly increase the speed of state-of-the-art forecasting methods. It is worth mentioning that some research lines derived from this primary intention may seem unorthodox from a traditional perspective. For instance, in diverse stages of this project, it was more important to setup a novel computing infrastructure and use it to verify algorithms that are already proven in the literature rather than propose a new algorithm and extensively test its performance. In a similar direction, the proportion of this research work that was directed toward the orchestration of multiple layers of data and software components in the cloud (data engineering) is as important (maybe even more important at certain stages) as the work that was directed toward data processing, analysis, and interpretation of results (data science). This unconventional order of priority was, sometimes, unavoidable as most of the algorithms I implemented have proven, in the course of the investigation, to be impossible to execute on computing infrastructures that lack rigorously designed hardware acceleration. Fortunately, research work is the ideal spot to break conventions when they hinder knowledge acquisition.

Consequently, this Thesis emphasizes the fact that machine learning cannot be assumed anymore as a research field where successful contributions exclusively depend on the selection of the right algorithm. Now, a multi-level choice for the integration of an algorithm, a software stack, and a specific

hardware is required to produce results that advance the state-of-the-art [Hooker, 2021]. I started my research from this assumption and progressively adapted to it by carefully balancing the importance given to these three elements during the investigation.

1.2 Hypothesis

It is possible to achieve state-of-the-art results in time series forecasting with significant processing time reductions by implementing specific machine learning algorithms on cloud computing resources, and particularly by leveraging distributed computing tools, hardware-based accelerators, and production-level software frameworks.

1.3 Objectives

The general and specific objectives of this Thesis are as follows.

General Objective

To achieve state-of-the-art results in time series forecasting with significant processing time reductions by implementing a series of proven ML-based algorithms on the basis of specialized cloud-computing resources like distributed computing, hardware-based accelerators, and production-level, DL-specific frameworks.

Specific Objectives

1. To design a cloud-based ecosystem, including the solution architecture, infrastructure, and software stack, for the implementation of contemporary ML-based algorithms for TSF. This ecosystem is required to offer distributed computing functionalities for, at least, a proven DL-specific computing framework.

2. To design, implement, and test a series of relevant ML-based algorithms[1] for TSF on the cloud-based ecosystem mentioned above.

[1] The algorithms originally proposed for this objective were Nearest-Neighbors with

3. To evaluate the performance, in terms of forecasting accuracy and training speed, of the resulting implementations, as well as to compare it with the performance achieved by conventional methods of TSF when executed in competitive environments.

1.4 Overview of Thesis

This Thesis comprises six Chapters that can be grouped into three parts: Chapters 2 and 3 provide background materials. Chapters 4, 5, and 6 are mainly experimental, and present the implementation of the DL architectures that account for most of recent TSF research in the proposed framework. On this basis, Chapter 7 presents my main contribution to the global, multi-horizon TSF problem. I extend this overview in the following paragraphs.

Chapter 2 presents the generic TSF problem as well as a classification of forecasting use cases. It also presents a collection of methods that have received important attention from researchers in the TSF domain. Among those methods, the three approaches that account for most of recent TSF research are discussed in detail.

Chapter 3 presents the basic elements for implementing Distributed Machine Learning (DML) projects on cloud computing. It discusses the design and implementation of DML and presents the most extended computing frameworks used for that goal. It also presents the concept of cloud computing as well as the cloud-specific Tensor Processing Unit (TPU) accelerator.

Chapter 4 presents the implementation of a deep, multi-sequence, LSTM-based architecture for local, multi-horizon TSF. It discusses related work, the experimental study on an extensive power distribution dataset, and the corresponding results.

Chapter 5 presents the implementation of a deep, LSTM-based Sequence to Sequence (Seq2Seq) with attention architecture for local, multi-horizon TSF. It also presents related work, the experimental study on an extensive power distribution dataset, and the corresponding results.

Chapter 6 presents the implementation of a deep Transformer-based architecture for local, multi-horizon TSF. It discusses related work, the experimental study on an extensive power distribution dataset, and the corresponding

---

Differential Evolution and the basic implementation of the Long Short-Term Memory (LSTM) network. As a result of the research process, those algorithms were replaced by the three architectures that account for most of the recent research in the field.

results.

Chapter 7 presents the implementation of a deep Transformer-based architecture for global multi-horizon TSF. It also presents the experimental study on two predominant standard datasets and the corresponding results.

Finally, Chapter 8 presents concluding remarks and directions for future work.

Chapter 2

Machine Learning for Time Series Forecasting

This Chapter presents the basic elements for using Machine Learning (ML) methods in the Time Series Forecasting (TSF) problem. Section 2.1 provides the generic definition of the TSF problem, and departs from that basis to discuss the characteristics that determine common forecasting use cases. Section 2.2 presents a collection of methods that have received important attention from researchers in the TSF domain. On this basis, Section 2.3 presents in detail the three most reported architectures, at the time of this writing, for ML applications in TSF. It is worth highlighting that these three architectures constitute the core of the experimental study of this Thesis.

2.1 Time Series Forecasting Fundamentals

This section presents the generic definition of the TSF problem. On that basis, common forecasting use cases are discussed as particular design choices made on the one- to multiple-step ahead, local to global, and point to probabilistic forecasting alternatives.

2.1.1 Problem definition

Let $\{\mathbf{z}_{i,1:t_0}\}_{i=1}^{N}$ be a collection of N related time series, where the $i-th$ series $\mathbf{z}_{i,1:t_0} = [\mathbf{z}_{i,1}, \mathbf{z}_{i,2}, \ldots, \mathbf{z}_{i,t_0}]$ and $\mathbf{z}_{i,t} \in \mathbb{R}$ is the value of time series i at time t (where time index t is relative, therefore the same t in different time series

may represent different time points). Let $\tau \in \mathbb{N}$ be the *forecast horizon*. I will predict the next τ time steps for all time series, i.e. $\{\mathbf{z}_{i,t_0+1:t_0+\tau}\}_{i=1}^{N}$ Also, let $\{\mathbf{x}_{i,1:t_0+\tau}\}_{i=1}^{N}$ be a set of associated time-based covariate vectors with dimension d_x that are known over the entire time period, e.g. hour-of-the-day or day-of-the-week. Finally, let $\{\mathbf{s}_i\}_{i=1}^{N}$ be a set of associated static covariate vectors of dimension d_s that are known and constant over the entire time period, e.g. a time series identifier.

A prediction function f is modeled as:

$$\hat{\mathbf{z}}_{i,t_0+1:t_0+\tau} = f(\mathbf{z}_{i,1:t_0}, \mathbf{x}_{i,1:t_0+\tau}, \mathbf{s}_i) \tag{2.1}$$

which is the general expression for the global, multi-step TSF problem. In the following subsections, I discuss particular cases that derive from (2.1) as the intersection of three design choices: one-step ahead versus multiple-step ahead forecasting, local versus global forecasting, and point versus probabilistic forecasting.

2.1.2 One- vs. Multiple-step ahead Forecasting

From (2.1), let $i = 1$, then it is obtained the first single time series $\mathbf{z}_{1:t_0}$ to build predictions on. In the absence of other time series, the information provided by static covariates $\mathbf{s}$ becomes irrelevant, then they are neglected. Also, let $\tau = 1$ and the expression for one-step ahead forecasting of the time series results in:

$$\hat{\mathbf{z}}_{t_0+1} = f(\mathbf{z}_{1:t_0}, \mathbf{x}_{1:t_0+1}) \tag{2.2}$$

Figure 2.1 illustrates one-step ahead forecasting as a generic regression problem [Bontempi et al., 2012], which is equivalent to (2.2) when time-based covariates $\mathbf{x}$ are not considered.

Similarly, multiple-step ahead forecasting of time series $\mathbf{z}_{1:t_0}$ with a horizon $\tau > 1$ is given by

$$\hat{\mathbf{z}}_{t_0+1:t_0+\tau} = f(\mathbf{z}_{1:t_0}, \mathbf{x}_{1:t_0+\tau}) \tag{2.3}$$

Different strategies have been developed to deal with the multiple-step ahead, also referred to as *multi-horizon*, TSF problem. As early as 2004, [Kline, 2004] proposed a classification of solutions based on neural networks. On the basis of this work, [Taieb et al., 2012] proposed a more general classification that includes the following categories:

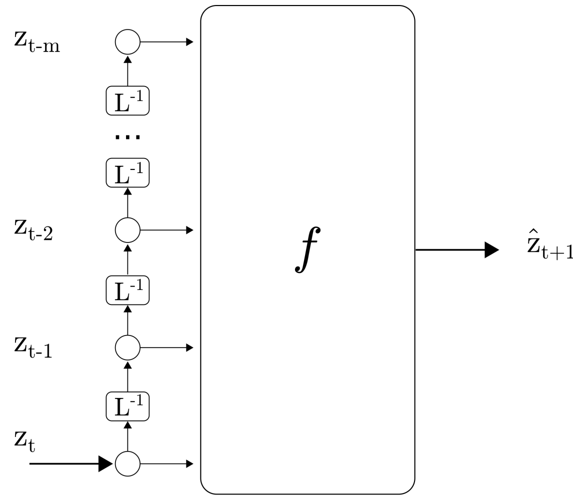


Figure 2.1: One-step ahead forecasting as a generic regression problem. As described in [Bontempi et al., 2012], the prediction function f is an approximator based on the *lag* operator L, which is calculated as $L^k y_t = y_{t+k}$.

- Iterative. Also referred to as *recursive*. It trains a one-step ahead model and then uses it iteratively to build a multi-step prediction (see Figure 2.2). A commonly observed drawback of iterative forecasting is that prediction errors are accumulated over the forecasting horizon.

- Direct. It trains τ independent models f_ν as $\hat{\mathbf{z}}_{t_0+\nu} = f_\nu(\mathbf{z}_{1:t_0}, \mathbf{x}_{1,t_0+\nu})$ for $\nu \in 1, \ldots, \tau$, then returns a multi-step forecast by concatenating the τ predictions. Unlike the iterative case, the direct strategy does not produce error accumulation. However, since the τ models f_ν are learned independently, the likely useful statistical dependencies between predictions in the forecast horizon are not considered.

- Multiple-Input, Multiple-Output (MIMO). It learns a single forecasting model able to produce, in a one-step execution, an output with multiple predictions in the forecast horizon. As such, the MIMO model f is directly derived from (2.3), with $f : \mathbb{R}^{d_{input}} \to \mathbb{R}^{\tau}$, where d_{input} is the dimensionality of the vector that results from the concatenation of the past observations $\mathbf{z}$ and the dynamic covariates $\mathbf{x}$. As a result,

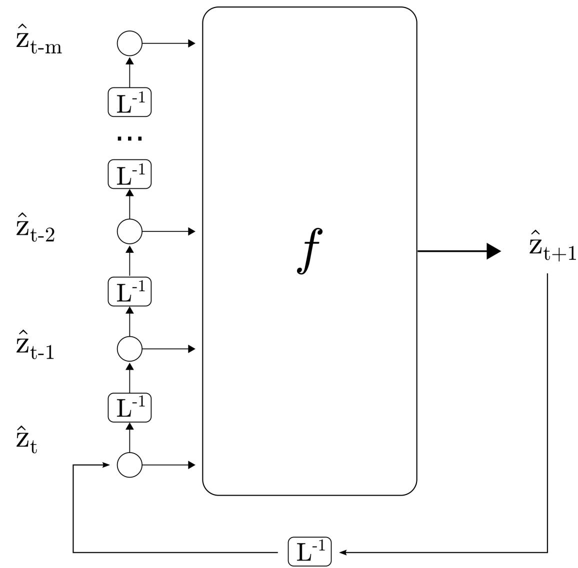


Figure 2.2: Iterative strategy for multiple-step ahead forecasting. As described in [Bontempi et al., 2012], the approximator f returns the prediction at time $t+1$ by iterating the forecast values obtained in the previous steps.

models produced with the MIMO strategy avoid the error accumulation characteristic of the iterative strategy as well as the assumption of conditional independence between future predictions present in the direct strategy.

- Hybrid variants. These are combinations of the direct with the recursive strategies (DirRec) [Sorjamaa and Lendasse, 2006], and the direct with the MIMO cases (DirMO) [Taieb et al., 2009].

2.1.3 Local vs. Global Forecasting

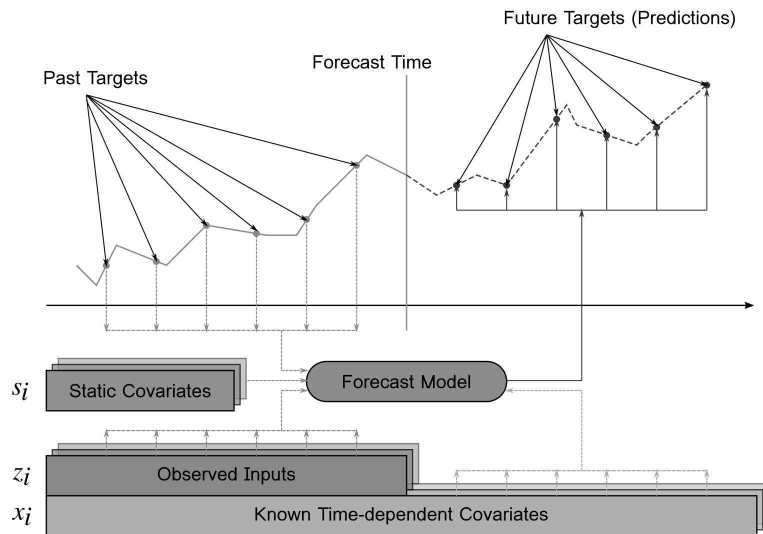


Figure 2.3: Global, multi-step time series forecasting. Figure based on [Lim et al., 2021].

The Big Data era has driven a generalized shift in TSF practice, when dealing with the problem of predicting over large collections of related time series. Two different approaches can be used [Januschowski et al., 2020]: *local* models that estimate parameters independently from each time series, or *global* methods that estimate model parameters jointly from all available time series. Global forecasting models are generally intended for multi-step prediction, therefore are expressed by (2.1). Local forecasting models, as previously stated, are expressed by (2.2), and (2.3), for the one- and multiple-step-ahead cases, respectively.

Figure 2.3 illustrates the general process of global, multi-step forecasting. The horizontal line labeled as *forecast time* separates the time series' history, or *conditioning range*, from the *prediction range*. To the left of the forecast time, for each element i in a collection of related time series $\{\mathbf{z}_{i,1:t_0}\}_{i=1}^{N}$, observed values for the interest variable z_i, associated time-based covariates x_i, and static covariates s_i are passed as inputs to a single forecast model. For each m-sized look-back window, a (past) target value is used as a label for training a supervised learning model. Time-dependent covariates (i.e. hour, weekday, or month-based encodings) generate a temporal alignment, which can be used to find patterns across time series dynamics. Static covariates (i.e. geographical location, customer identity, or item category), in conjunction with proper embeddings, enable the model to group time series into sub-sets with similar behavior. Once trained, the model can predict a given number of time steps into the future of any time series in the collection. To the right of the forecast time (during inference time), only time-based covariates and static covariates are known, then the look-back window remains to the left of the forecast time and the targets that result from the model are predictions over the forecast horizon. [Hewamalage et al., 2021] affirm that any local method applied on a dataset of many series can be approximated by a global model with sufficient complexity, irrespective of the relatedness of the underlying series. Therefore, for a global model to outperform a collection of local models on a given collection of time series, it is required to find the right amount of model complexity. The authors designed an extensive simulation study to investigate the effect that factors such as model complexity, pattern complexity, homogeneity, and size of time series have on the compared performance of global to local forecasting models.

Among a large collection of forecasting techniques, deep neural networks are proven to be particularly suitable for global forecasting approaches. DL-based architectures learn predictive relationships by using a series of non-linear layers to construct intermediate feature representations [Lim and Zohren, 2021]. The generic process consists of encoding relevant information into a latent variable $\mathbf{l}$, then using $\mathbf{l}$ to build the final forecast, as

$$f(\mathbf{z}_{i,t-m:t}, \mathbf{x}_{i,t-m:t}, \mathbf{s}_i) = g_{dec}(\mathbf{l}_t) \tag{2.4}$$

and

$$\mathbf{l}_t = g_{enc}(\mathbf{z}_{i,t-m:t}, \mathbf{x}_{i,t-m:t}, \mathbf{s}_i) \tag{2.5}$$

where m is the size in timesteps of a look-back window that is used to acquire data examples from the time series, $g_{enc}(\cdot)$, $g_{dec}(\cdot)$ are *encoder* and *decoder*

functions, respectively, and the forecast horizon is set to 1, for simplicity. I will return later to this *encoder-decoder* approach for global TSF when discussing the implementation of selected techniques for the experimental study of this Thesis.

2.1.4 Point vs. Probabilistic Forecasting

Assuming a single time series, a TSF model developed for *point forecasting* outputs a fixed target value $\hat{\mathbf{z}}_{t_0+k}$ for each timestep $t_0 + k$, with $k \in \{1, \cdots, \tau\}$, to be predicted. This is equivalent to predicting the conditional mean $\mathbb{E}(\mathbf{z}_{t_0+k}|\mathbf{z}_{1:t_0}, \mathbf{x}_{1:t_0+k})$ of a probability distribution p calculated at each timestep in the forecast horizon [Wen et al., 2017]. In many TSF situations it is important to provide information regarding the uncertainty of the forecast values, therefore a *probabilistic forecasting* model, able to output the full conditional distribution $p(\mathbf{z}_{t_0+k}|\mathbf{z}_{1:t_0}, \mathbf{x}_{1:t_0+k})$ is more appropriate.

A common way to model forecast uncertainties is by assuming a probability distribution for the predictions at each timestep (for instance a Gaussian distribution for continuous targets), then training a DL-based model to generate the parameters of the distribution (in the example, mean and variance) [Lim and Zohren, 2021]. Quantile Regression is another popular approach to model forecast uncertainty. It is used to train models by minimizing the Total Quantile Loss [Wen et al., 2017] and, unlike parametric approaches, it does not have to assume a probability distribution for the prediction values.

2.2 Time Series Forecasting Methods

This section introduces a collection of TSF methods that, in recent years, have reported important results in the research literature. I avoid the commonplace of classifying forecasting methods as being either statistical- or machine learning-based, as I adhere to the work of [Januschowski et al., 2020] which affirms this distinction does not arise from fundamental differences in the methods assigned to either class, and it limits the insights into the appropriateness and effectiveness of different forecasting methods. Instead, I opted for describing the six approaches to TSF that, according to the revision of the state-of-the-art, account for the majority of recently reported research and use cases. This collection comprises the following methods:

- Exponential Smoothing.

- Auto Regressive Integrated Moving Average (ARIMA).
- Tree-based Models.
- Convolutional Neural Networks.
- Recurrent Neural Networks.
- Attention-based Models.

Moreover, I emphasize the increased importance that hybrid models have gained in TSF, and provide some relevant examples. Finally, I mention other important forecasting approaches that have recently shown decreasing interest in the literature.

2.2.1 Exponential Smoothing Models

Exponential smoothing [Brown and Meyer, 1961] is one of the most extensively used forecasting methods. It has served a diversity of key domains of the forecasting environment for decades. The basic principle of exponential smoothing is to produce a prediction on the basis of the aggregation of weighted contributions of past observations. The weights assigned to observations decrease exponentially as one moves from the present to the past. Different exponential smoothing methods can be chosen from a well-defined catalog [Pegels, 1969] to deal with time series of different complexity, that is time series that include different combinations of patterns like level, trend, seasonality, or multiple cycles. The simplest form of exponential smoothing is Simple Exponential Smoothing (SES), which accounts for the *level* pattern only (no clear trend or seasonality is observed in the time series data), and is expressed recursively as $\hat{z}_{t+1} = \alpha z_t + (1 - \alpha)\hat{z}_t$, where $\hat{z}_t$ and z_t are the forecast value and the observation at time t, respectively, and α is the level smoothing parameter. An extensive critical review of exponential smoothing models is provided in a two-part, state-of-the-art study by [Gardner Jr, 1985] and [Gardner Jr, 2006].

2.2.2 ARIMA Models

Like exponential smoothing methods, autoregressive models make predictions of future values of a time series based on functions built as linear

combinations of past observations. Unlike exponential smoothing, autoregressive models assume that successive observations in a time series show serial dependence between the signal and noise. The most popular of autoregressive models, ARIMA [Box et al., 2008] has been extensively and successfully used for local, univariate forecasting of time series with a linear behavior. The ARIMA(p, d, q) model of a time series z is expressed as $(1 - \phi_1 L - \cdots - \phi_p L^p)(1 - L)^d z_t = c + (1 + \theta_1 L + \cdots + \theta_q L^q) + \epsilon_t$ where p, d, and q are the orders of the autoregression, differencing, and moving average components, ϕ_i and θ_i are the i-th coefficients for autoregression and moving average, L is the lag operator, calculated as $L^k z_t = z_{t-k}$, and ϵ_t is a zero-mean process that is not correlated with z. Relevant extensions to the ARIMA method are Seasonal ARIMA (SARIMA) [Williams and Hoel, 2003] and ARIMA with eXogenous variables(ARIMAX) [Williams, 2001].

2.2.3 Tree-based Models

Tree-based forecasting methods have achieved outstanding results in competitions such as M4, M5, and Kaggle's Rossmann Store Sales and Global Energy Forecasting [Januschowski et al., 2021], to mention a few. Two techniques stand out among tree-based forecasting models: Random Forests and Gradient Boosted Decision Tree (GBDT)s. A Random Forest [Breiman, 2001] is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest; where the generalization error for the forest converges to a limit as the number of trees in the forest becomes large. On the other hand, GBDT [Friedman, 2001] is an ensemble model of decision trees, which are trained in sequence. In each iteration, it learns the decision trees by fitting the negative gradients, also known as residual errors [Ke et al., 2017]. XGBoost [Chen and Guestrin, 2016] is a popular implementation of GBDT that provides scalability up to thousands of millions of samples via important optimizations on systems and algorithms. Tree-based models are global models that provide cross-learning, that is sharing parameters between related time series to enhance the resulting predictive performance.

2.2.4 Convolutional Neural Networks

The CNN architecture is widely used as a basic block to build temporal encoder functions for TSF, as defined in 2.5. For this purpose, CNNs are

adapted to fulfill two conditions [Lara-Benítez et al., 2020]: the network's output is set, via zero-padding, to the same length as its input; and standard convolution is replaced with causal convolution to ensure the forecasting is based only on past information. The causal convolution is expressed as [Lim and Zohren, 2021]:

$$\mathbf{h}_t^{j+1} = A((\mathbf{W} * \mathbf{h})(j,t)), \tag{2.6}$$

$$(\mathbf{W} * \mathbf{h})(j,t) = \sum_{i=0}^{m} \mathbf{W}(j,i)\mathbf{h}_{t-i}^{j}, \tag{2.7}$$

where $\mathbf{h}_t^j \in \mathbb{R}^{\mathbb{H}_{input}}$ is the intermediate (hidden) state of the CNN layer j at time t, $*$ is the convolutional operator, $\mathbf{W}(j,i) \in \mathbb{R}^{\mathbb{H}_{output} \times \mathbb{H}_{input}}$ is a fixed filter weight at layer j, and A is a non-linear activation function. Then, for a temporal encoder with a total of J convolutional layers the output of the encoder function is given by $\mathbf{l}_t = \mathbf{h}_t^J$.

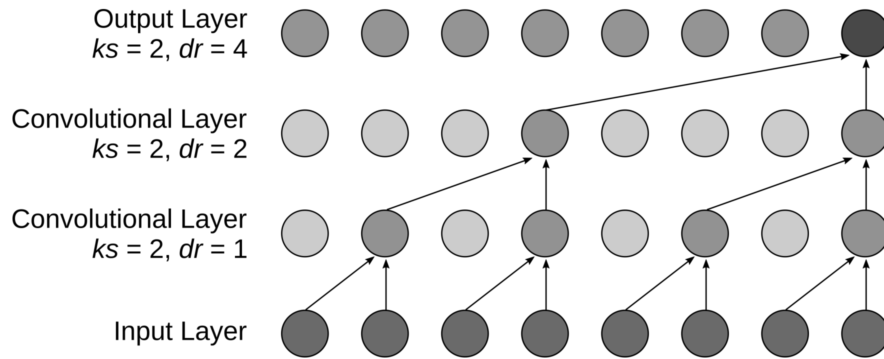


Figure 2.4: CNN-based temporal encoder architecture. Figure based on [Lim and Zohren, 2021]

.

In order to learn long-term dependencies, causal convolutions are extended to *dilated convolutions* by cascading causal layers to down-sampled versions of lower-layer features. The effect of reducing the resolution on lower layers is that more information from the distant past is included. The *kernel size* parameter that defines a standard convolution is complemented with a *dilation rate* parameter to characterize a dilated convolution layer. Figure 2.4 illustrates a temporal encoder based on CNN. A first CNN layer filters the inputs using a kernel size $ks = 2$ and a dilation rate $dr = 1$ (no down-sampling). A second CNN layer is cascaded to the first one with $ks = 2$ and

$dr = 2$. Finally, a third CNN layer is cascaded with $ks = 2$ and $dr = 4$ to produce the output. Architectures based on causal dilated convolutional layers also referred to as Temporal Convolutional Networks, have reported successful results in the literature [Wan et al., 2019], [Hewage et al., 2020].

2.2.5 Recurrent Neural Networks

RNN architectures are based on blocks that implement internal connections to develop a memory of past events. This feature makes RNNs adequate to model temporal dependencies on many types of sequences. Besides TSF, RNNs have been successfully applied to domains like language modeling, audio modeling, speech recognition, and image generation [Bianchi et al., 2017], to mention a few. In forecasting applications, a RNN performs the same operations on every value of the input sequence. At each timestep, the block output is calculated with the current observation and the memory states of previous blocks, which depend on previous observations. The RNN memory state is then updated recursively as:

$$\mathbf{l}_t = \sigma(\mathbf{l}_{t-1}, \mathbf{z}_t, \mathbf{x}_t, \mathbf{s}), \tag{2.8}$$

where $\mathbf{l}_t \in \mathbb{R}^{\mathbb{H}}$ is the hidden state of the network at time t and $\sigma(.)$ is the memory update function, which depends on the RNN type. For instance, the Elman RNN is defined by the following expressions

$$\mathbf{z}_{t+1} = \sigma_z(\mathbf{W}_z \mathbf{l}_t + \mathbf{b}_z), \tag{2.9}$$

$$\mathbf{l}_t = \sigma_l(\mathbf{W}_{l_1} \mathbf{l}_{t-1} + \mathbf{W}_{l_2} \mathbf{z}_t + \mathbf{W}_{l_3} \mathbf{x}_t + \mathbf{W}_{l_4} \mathbf{s} + \mathbf{b}_l), \tag{2.10}$$

where $\mathbf{W}, \mathbf{b}$ are the linear weight and bias matrices of the network and $\sigma_z(.), \sigma_l(.)$ are activation functions.

Figure 2.5 illustrates a deep RNN architecture with two recurrent layers on top of the input sequence. RNNs are commonly stacked to allow each layer to operate at a different time scale [Sagheer and Kotb, 2019] which results in the network learning more complex patterns from the data.

2.2.6 Attention-based Models

Attention mechanisms were introduced to enhance sequence-to-sequence modeling [Graves et al., 2014] by learning separated contributions that each element of the source sequence has in calculating each element of the target

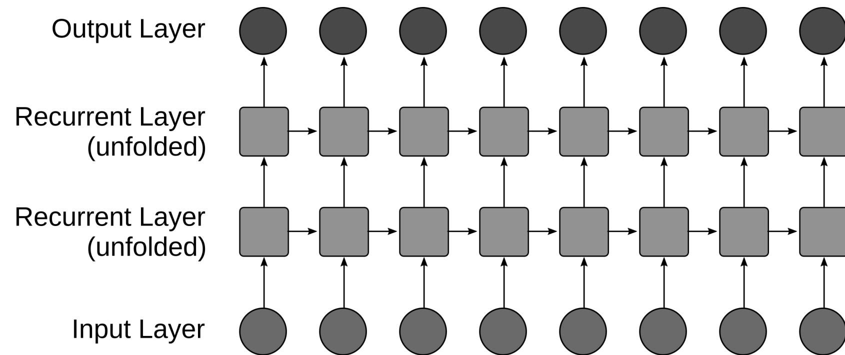


Figure 2.5: RNN-based temporal encoder architecture. Figure based on [Lim and Zohren, 2021]

.

sequence. The attention mechanism is expressed in its most basic form as:

$$\mathbf{h}_t = \sum_{i=0}^{m} \alpha(\mathbf{k}_t, \mathbf{q}_i)\mathbf{v}_{t-i}, \tag{2.11}$$

where the *key* $\mathbf{k}_t$, *query* $\mathbf{q}_i$, and *value* $\mathbf{v}_{t-i}$ are intermediate features produced at different time steps by lower levels of the network, $\alpha(\mathbf{k}_t, \mathbf{q}_i) \in [0, 1]$ is the attention weight the upper layer at timestep t grants to the lower layer at $t - i$, and $\mathbf{h}_t$ is the context vector output of the attention layer [Lim and Zohren, 2021].

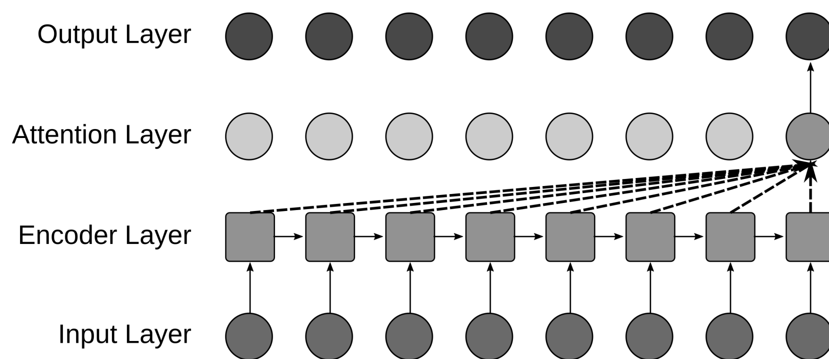


Figure 2.6: Attention-based temporal encoder architecture. Figure based on [Lim and Zohren, 2021]

.

Figure 2.6 illustrates temporal encoding enhanced with an attention layer. The network calculates the rightmost output value using weighted contributions (dotted connections) of each timestep in the encoder layer. Several alignment models have been proposed to evaluate attention. [Bahdanau et al., 2014] calculate the attention weight by using a single-layer MLP to reduce computation costs. [Luong et al., 2015] propose various attention-based models and classify them into two broad categories: *global* models that attend all source sequence positions, and *local* models which attend to a few selected source positions. Later in this Chapter, I will discuss the *transformer* neural network architecture, an attention-based model that has recently advanced the state-of-the-art for the TSF problem.

2.2.7 Hybrid Models

The forecasting models described so far are frequently combined into hybrid configurations that attempt to benefit from the advantages of multiple components. For instance, [Xiao et al., 2021] propose an architecture that uses CNN layers to extract "spatial" features of multi-variate time series, that is the interrelationship of different variables at the same time. The long-term, temporal patterns in the time series are extracted using LSTM layers in conjunction with a dual-stage attention mechanism. [Guo et al., 2019] propose a hybrid architecture to forecast traffic flows. It models dependencies at three different temporal resolutions (recent, daily, and weekly) using three tiers composed of spatial-temporal attention mechanisms, graph-CNN, and standard CNN layers. [Smyl, 2020] proposes the Dynamic Computational Graph Neural Network system, a hybrid architecture that enhances a standard exponential smoothing model with LSTM layers. It features three main elements: de-seasonalization and adaptive normalization, generation of forecasts, and ensembling. This hybridization of statistical and ML techniques is the winner of the M4 international forecasting competition [Makridakis et al., 2020], and has gained extensive interest among researchers and practitioners. [Wang et al., 2022] propose another hybrid approach for TSF. This work uses CNN layers to divide the original time series data into sub-series with different time scales, then feeds the sub-series a to a state-of-the-art transformer-based model referred to as *Informer* [Zhou et al., 2021]. This work also provides a summary of recent studies for wind power forecasting based on hybrid models.

2.2.8 Other Forecasting Models

Important approaches to TSF that in recent years have shown decreasing interest in the literature are State-Space models [Gu and Anderson, 1995], Multi-Layer Perceptron (MLP) [Zhang et al., 2001], k-Nearest Neighbors [Martínez et al., 2019], Autoregressive Conditional Heteroskedasticity (ARCH)/Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models [Lv and Yue, 2011], Support Vector Regression [Lu et al., 2009], and Fuzzy Logic [Cheng et al., 2016], among others. For an introductory review of these methods, I suggest the extensive taxonomy presented by [Petropoulos et al., 2022].

2.3 Neural Networks Implemented in Thesis

This Section presents in detail the three architectures that account for most of the ML applications to TSF reported in the literature, at the time of this writing. These architectures are the Long Short-Term Memory (LSTM) network, the Sequence to Sequence (Seq2Seq) with attention network, and the Transformer network. In practice, LSTM networks allow for enhanced long-term-dependency learning. On the other hand, Seq2Seq with attention architectures extend and stabilize long-term-dependency learning in multi-step forecasting situations. Moreover, Transformer-based models provide the aforementioned advantages with an algorithm that allows for a great extent of parallelization, which results in state-of-the-art applications for global, multi-step, probabilistic forecasting problems. These three architectures also constitute the core of the experimental study of this Thesis.

2.3.1 Long Short-Term Memory Network

The ability of RNNs to develop a memory of past events makes them adequate to model temporal dependencies on time series data. However, early RNN designs suffer from a well-known issue: when trained with conventional gradient-based techniques like Back-Propagation Through Time or Real-Time Recurrent Learning, error signals that flow backward in time tend to either blow up or disappear, as the temporal evolution of the backpropagated error exponentially depends on the size of the weights. LSTM is a special type of RNN that was initially proposed by [Hochreiter and Schmidhuber, 1997] to deal with the *exploding/vanishing gradients* problem. To

accomplish this objective, LSTM networks are provided with a set of special gates that modulate its memory state as follows: an input gate **i** protects the memory contents from perturbation by irrelevant inputs, an output gate **o** protects subsequent LSTM units in the layer from perturbation by currently irrelevant memory contents, and a forget gate **f** (introduced at a later time by [Gers et al., 2000]) enables the LSTM to reset its own state.

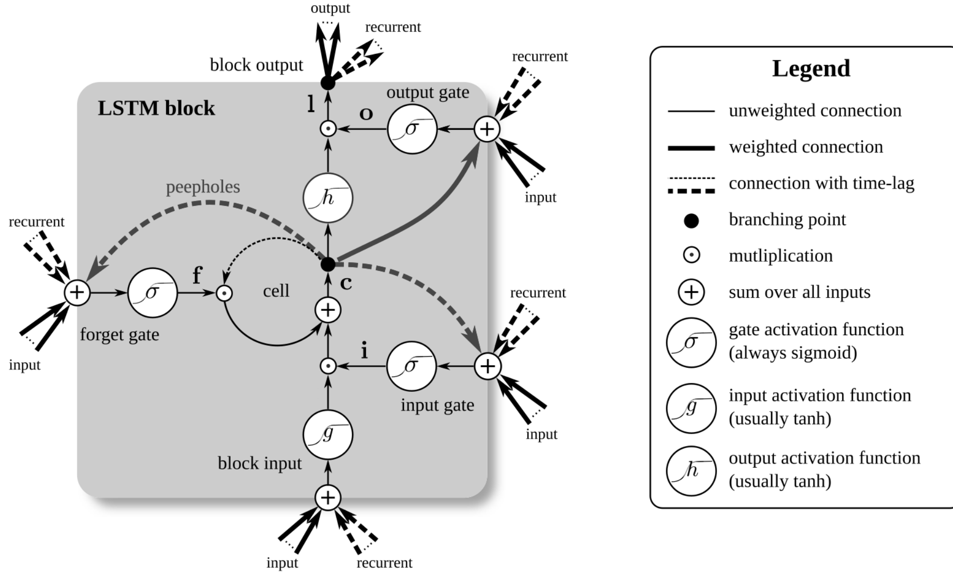


Figure 2.7: Detailed schematic of the LSTM block. Source: [Greff et al., 2016]

.

From (2.8), LSTM internal gates are expressed as:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{i_1}\mathbf{l}_{t-1} + \mathbf{W}_{i_2}\mathbf{z}_t + \mathbf{W}_{i_3}\mathbf{x}_t + \mathbf{W}_{i_4}\mathbf{s} + \mathbf{b}_i), \tag{2.12}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{o_1}\mathbf{l}_{t-1} + \mathbf{W}_{o_2}\mathbf{z}_t + \mathbf{W}_{o_3}\mathbf{x}_t + \mathbf{W}_{o_4}\mathbf{s} + \mathbf{b}_o), \tag{2.13}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{f_1}\mathbf{l}_{t-1} + \mathbf{W}_{f_2}\mathbf{z}_t + \mathbf{W}_{f_3}\mathbf{x}_t + \mathbf{W}_{f_4}\mathbf{s} + \mathbf{b}_f), \tag{2.14}$$

where σ is the *sigmoid* activation function. As illustrated in Figure 2.7, these gates modify the hidden state **l** and the cell state **c** as:

$$\mathbf{l}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t), \tag{2.15}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{c_1}\mathbf{l}_{t-1} + \mathbf{W}_{c_2}\mathbf{z}_t + \mathbf{W}_{c_3}\mathbf{x}_t + \mathbf{W}_{c_4}\mathbf{s} + \mathbf{b}_c), \quad (2.16)$$

where $\odot$ is the element-wise (Hadamard) product, and $\tanh(\cdot)$ is the tanh activation function. LSTM architectures in TSF allow for keeping a stable memory from the early stages of the data dynamics, and to use this memory along with present information to better build predictions. In Chapter 4 I present relevant applications of the LSTM architecture to TSF models, as well as my multi-layered implementation for distributed training on cloud computing.

2.3.2 Sequence-to-Sequence with Attention Network

A diversity of domains demand TSF models to provide multiple-step predictions based on a MIMO strategy. The Seq2Seq neural network architecture, initially developed for automatic translation purposes, was adapted to accomplish this task. To perform statistical machine translation, [Cho et al., 2014] proposed an *encoder-decoder* architecture where a RNN-based encoder maps a variable-length source sequence to a fixed-length vector, then a RNN-based decoder maps this vector representation back to a variable-length target sequence. They trained the two networks jointly to maximize the conditional probability of the target sequence given a source sequence. [Sutskever et al., 2014] also proposed a Seq2Seq model where both the encoder and decoder are based on multi-layer LSTM networks. The authors affirm the translation quality was improved by reversing the rank of the elements in the input sequence. [Wu et al., 2016] extended the use of deep architectures for neural machine translation with a Seq2Seq model that included eight LSTM layers on both the encoder and the decoder, as well as residual connections. This architecture was designed to improve parallelism during training and to accelerate the final translation during inference.

[Bahdanau et al., 2014] and [Luong et al., 2015] improved the Seq2Seq architecture by adding attention mechanisms. Original Seq2Seq models used to encode complete input sequences into a single, fixed-length vector representation. Attention mechanisms add flexibility as they allow the model to encode an input sequence into a sequence of vectors, or *annotations*, then to choose a subset of these vectors adaptively during the decoding stage. Figure 2.8 illustrates the Seq2Seq model with attention as proposed in [Bahdanau et al., 2014], with minor adaptations to comply with the nomenclature

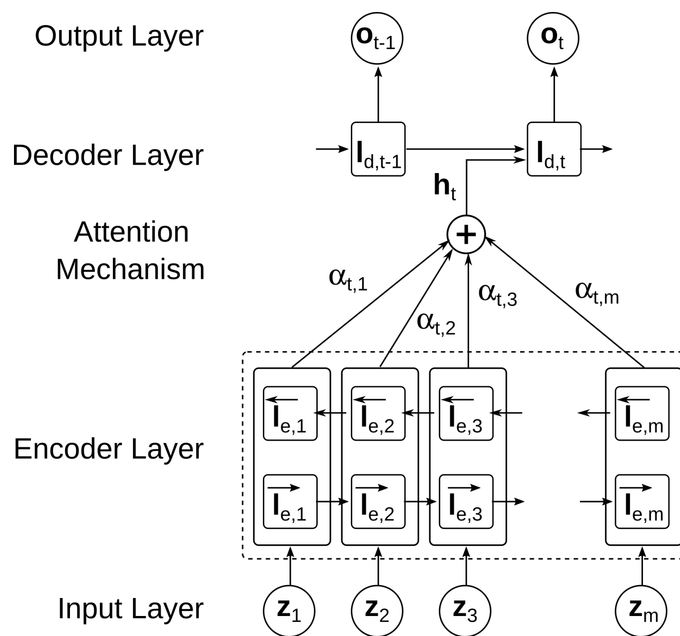


Figure 2.8: Sequence-to-Sequence with Attention Network. Based on [Bahdanau et al., 2014]

.

adopted for the TSF problem. For simplicity, in the discussion of the Seq2Seq with attention architecture, I omit the time-dependent covariates $\mathbf{x}$ and the static covariates $\mathbf{s}$, and restrict the input sequence to the past observations $\mathbf{z}$ only. Then, $(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m)$ is the input sequence (the time series segment in the rolling, look-back window) that is passed to the input layer. The encoder layer is built on the basis of a bi-directional RNN with a forward RNN and a backward RNN. The forward RNN $\overrightarrow{f}$ reads the input sequence as it is ordered, from $\mathbf{z}_1$ to $\mathbf{z}_m$, and calculates a sequence of *forward hidden states* of the encoder $(\overrightarrow{\mathbf{l}_{e,1}}, \overrightarrow{\mathbf{l}_{e,2}}, \ldots, \overrightarrow{\mathbf{l}_{e,m}})$. The backward RNN $\overleftarrow{f}$ reads the input sequence in the reverse order, from $\mathbf{z}_m$ to $\mathbf{z}_1$, and delivers a sequence of *backward hidden states* of the encoder $(\overleftarrow{\mathbf{l}_{e,1}}, \overleftarrow{\mathbf{l}_{e,2}}, \ldots, \overleftarrow{\mathbf{l}_{e,m}})$.

As for the attention mechanism, for each element $\mathbf{z}_i$ in the input sequence, an annotation vector $\mathbf{l}_i$ is obtained by concatenating the forward and backward hidden states along the example-row axis, as $\mathbf{l}_i = [\overrightarrow{\mathbf{l}}_i^T; \overleftarrow{\mathbf{l}}_i^T]$. From (2.11), a context vector $\mathbf{h}_t$ is calculated as the sum of weighted contributions of the annotation vectors as $\mathbf{h}_t = \alpha_{t,1}\mathbf{l}_1 + \alpha_{t,2}\mathbf{l}_2 + \cdots + \alpha_{t,m}\mathbf{l}_m$, where $\alpha_{j,k}$ are learnable attention weights for decoding the jth time step of the output sequence from the annotation of the kth time step of the input sequence. As a result, the decoder layer is trained to generate $\mathbf{o}_t$ given the hidden state of the decoder at time t $\mathbf{l}_{d,t}$ and the context vector $\mathbf{h}_t$. From here, the predictions along the forecast horizon are generated as a time-distributed function

$$\hat{\mathbf{z}}_t = \sigma_z(\mathbf{W}_z\mathbf{o}_t + \mathbf{b}_z), \tag{2.17}$$

where σ_z is a non-linear activation function and $t \in [1, \ldots, f]$ where f is the forecast horizon.

Adding an attention mechanism to the Seq2Seq architecture makes it able to jointly produce attention weights (a functionality referred to as *align* in the original neural machine translation scope) and predict the output sequence (*translate* in the original). The capability of the decoder to decide the parts of the input sequence it attends to when predicting a specific part of the output sequence greatly improves long-term-dependency learning by relieving the encoder from the excessive requirement of encoding all information in the input into a fixed-length vector representation. Chapter 5 presents relevant applications of the Seq2Seq with attention architecture to TSF, as well as the proposed multi-layered implementation for distributed training on cloud computing.

2.3.3 Transformer Network

Like LSTM and Seq2Seq with attention architectures, Transformer-based networks were initially proposed to perform automatic translation tasks. [Vaswani et al., 2017] presented the canonical Transformer, also referred to as Vanilla Transformer (VT) in the literature, achieving superior performance and reduced training times with respect to the state-of-the-art translation models at that time. The main idea behind the Transformer is to build an *encoder-decoder* architecture, like the Seq2Seq network discussed above, that features a fundamental modification: both the encoder and the decoder modules are solely based on attention layers. In other words, the Transformer is able to learn short- and long-term dependencies from very large sets of sequential data using a special kind of attention mechanism that entirely leaves out any recurrent (or convolutional) component. As the Transformer algorithm does not rely on sequential calculations to build and maintain its memory state, it is amenable to parallelization. In consequence, the Transformer can effectively model prediction problems that required prohibitive computation times when approached with other architectures.

Next, I elaborate on the discussion of the Transformer for the global, multi-step, point TSF problem. From (2.1), I intend to model the following conditional distribution [Li et al., 2019b]:

$$p(\mathbf{z}_{i,t_0+1:t_0+\tau} \mid \mathbf{z}_{i,1:t_0}, \mathbf{x}_{i,1:t_0+\tau}, \mathbf{s}_i; \mathbf{\Phi})^1 = \prod_{t=t_0+1}^{t_0+\tau} p(\mathbf{z}_{i,t} \mid \mathbf{z}_{i,1:t-1}, \mathbf{x}_{i,1:t}, \mathbf{s}_i; \mathbf{\Phi}) \tag{2.18}$$

where t_0 is the last timestep of the input sequence (in the rolling, look-back window) and $\mathbf{\Phi}$ denotes the learnable parameters shared by all the time series in the collection. The problem defined in (2.18) is reduced to learning a one-step-ahead prediction model $p(\mathbf{z}_{i,t} \mid \mathbf{z}_{i,1:t-1}, \mathbf{x}_{i,1:t}, \mathbf{s}_i; \mathbf{\Phi})$, as multiple steps ahead can be iteratively calculated by adding previously predicted values as new inputs. To fully use observations and covariates, they are concatenated into an augmented matrix as follows:

[1] For simplicity, we omit the fact that this is the conditional distribution of the prediction range of time series i given the conditioning range of time series i, as well as the conditioning and prediction ranges of all other time series in the collection. I consider this situation implicit since the parameters $\mathbf{\Phi}$ are learned jointly from all the time series in the collection.

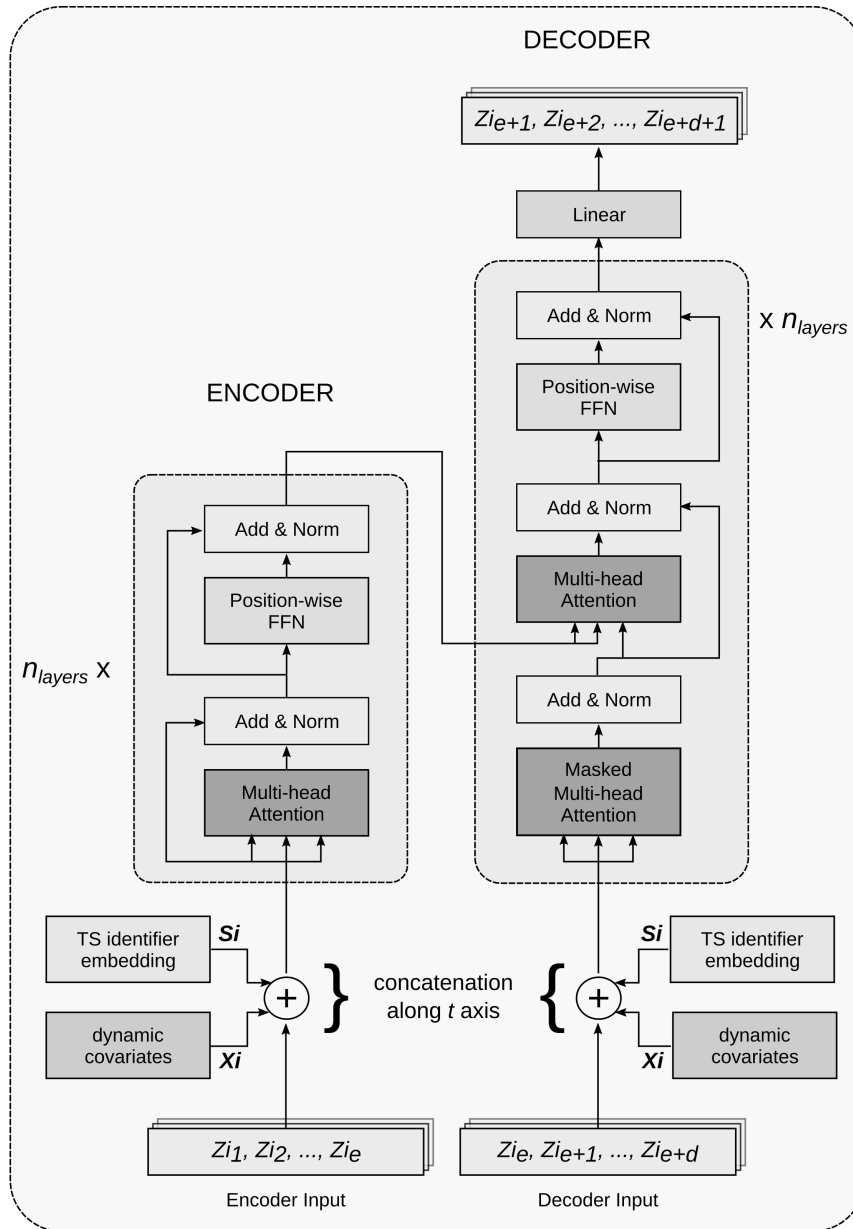


Figure 2.9: Transformer architecture.

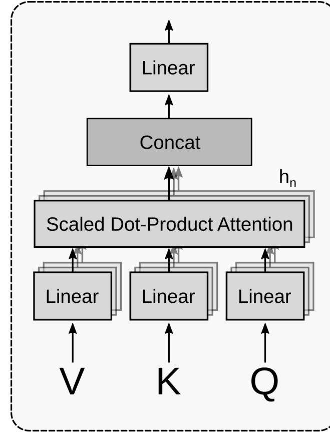


Figure 2.10: Multi-head Attention.

$$\mathbf{y}_{i,t} = [\mathbf{z}_{i,t-1} \circ \mathbf{x}_{i,t} \circ \mathbf{s}_i] \in \mathbb{R}^{d_x+d_s+1}, \quad \mathbf{Y}_t = [\mathbf{y}_1, \cdots, \mathbf{y}_t]^T \in \mathbb{R}^{t\times(d_x+d_s+1)2} \tag{2.19}$$

where $[\cdot \circ \cdot]$ represents concatenation along the t axis.

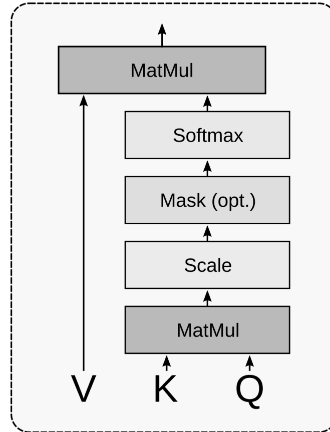


Figure 2.11: Scaled Dot-product Attention.

A Transformer-based model f is used to predict $\mathbf{z}_t$ given $\mathbf{Y}_t$ as $\mathbf{z}_t \sim f(\mathbf{Y}_t)$ The basic constructive block of the Transformer architecture is the generic *Transformer layer*, which is implemented as follows. First an internal

[2]Since the model applies to all the time series, the subscript i is omitted for simplicity.

dimension d_{model} is defined for the Transformer and the augmented matrix $\mathbf{Y}_t$ is linearly projected from $\mathbb{R}^{t\times(d_x+d_s+1)}$ to $\mathbb{R}^{t\times d_{model}}$. Notice in Figure 2.9 that the conditioning range passed as input to this model is split into an encoder input with dimension e and a decoder input with dimension d, where the value of t from (2.19) is given as $t = e + d + 1$ (the last value of the encoder input overlaps the first value of the decoder input, the decoder output is the decoder input shifted one timestep to the right, and the only predicted value is the last time step in the decoder output). Then a Multi-Head Attention (MHA) layer (Figure 2.10) is used to transform $\mathbf{Y}$ into H distinct query matrices $\mathbf{Q}_h = \mathbf{Y}\mathbf{W}_h^Q$, key matrices $\mathbf{K}_h = \mathbf{Y}\mathbf{W}_h^K$, and value matrices $\mathbf{V}_h = \mathbf{Y}\mathbf{W}_h^V$, with $h = 1, \cdots, H$. The weight matrices $\mathbf{W}_h^Q, \mathbf{W}_h^K \in \mathbb{R}^{d_{model}\times d_k}$ and $\mathbf{W}_h^V \in \mathbb{R}^{d_{model}\times d_v}$ are learnable parameters. The Scaled Dot-product attention (Figure 2.11) computes a sequence of vector outputs:

$$\mathbf{O}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) = \text{softmax}\left(\frac{\mathbf{Q}_h\mathbf{K}_h^T}{\sqrt{d_k}}\right)\mathbf{V}_h \tag{2.20}$$

The output of the MHA layer is calculated as follows

$$\text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\mathbf{O}_1, \cdots, \mathbf{O}_\text{h}]\mathbf{W}^O \tag{2.21}$$

where $\mathbf{W}^O \in \mathbb{R}^{hd_v\times d_{model}}$. Finally, the MHA output is passed to a Position-wise Feed-forward Network (FFN) with two dense layers (fully connected). The first dense projects d_{model} to a given d_{ff} using a ReLU activation, while the second dense returns the projection to d_{model}. The Transformer is a deep architecture, then Transformer layers are usually stacked one on top of the other. FFN layers are configured with $d_{model} << d_{ff}$, which allows the output of a Transformer layer to better fit the input for the subsequent Transformer layer. To complete the Transformer architecture, additional *Add and Layer-normalization* layers provide residual connections that allow the gradients to skip the MHA and/or FFN layers if they do not improve the model.

Based on these constructive elements, the canonical VT (see Figure 2.9) implements two Transformer objects and three Multi-head, Self-attention types. The two objects are the *Transformer-encoder* and the *Transformer-decoder*, and the three MHA types are the *encoder-attention* or *global self-attention*, the *decoder-attention* or *causal self-attention*, and the *encoder-decoder-attention* or *cross-attention*. Following is a brief discussion of these elements. The Transformer-encoder object is a multi-layered (deep) network that piles up a number of Transformer-encoder layers, one on top of the

other. Each Transformer-encoder layer consists of four layered elements: one MHA layer (the encoder-attention), one add-and-normalization layer, one position-wise FFN layer, and a second add-and-normalization layer. The Transformer-encoder object encodes the temporal information in the input sequence into a hidden state and a set of attention-weight matrices. The Transformer-decoder object is also multi-layered, as it piles up a number of Transformer-decoder layers. Transformer-decoder layers are built similarly to Transformer-encoder layers, with the exception of two components: first, their initial MHA layer (the decoder-attention) is masked to avoid leaking future information when predicting the value of the sequence at an earlier time step; and second, they feature an intermediate couple of MHA (the encoder-decoder-attention) and add-and-normalization layers. The Transformer-decoder object predicts an output sequence by applying an autoregressive operation to the input sequence and decoding the hidden state of the Transformer-encoder object. In subsequent Chapters, I discuss in further detail the operation of the Transformer objects and MHA types for TSF.

Transformer-based models for TSF allow for a more extensive parallelization, which results in better capturing of long-term dependencies and other relevant patterns by using neural networks that are far deeper and wider than the ones achievable with other architectures. In recent years, Transformers have consistently advanced the state-of-the-art for the global, multi-step, probabilistic TSF problem. Chapters 6 and 7 present two use cases of the Transformer architecture, for local and global forecasting, respectively. They also present relevant applications of this architecture to TSF, as well as a multi-layered implementation for distributed training on cloud computing, the main contribution presented in this Thesis.

Summary

This Chapter discusses the application of machine learning techniques to time series forecasting. It provides a research background for the TSF problem, as well as a description of important forecasting methods used nowadays. On this basis, the three neural network architectures that account for most of the recent research on TSF (Long Short-Term Memory networks, Sequence-to-Sequence with Attention Networks, and Transformers) are discussed. These

three architectures constitute the core of the experimental study of this Thesis.

The next Chapter discusses the implementation of distributed machine learning applications in cloud computing, as a pre-requisite for implementing the selected architectures in an adequate experimentation environment.

Chapter 3

Distributed Machine Learning on Cloud Computing

This Chapter presents the basic elements for implementing Distributed Machine Learning (DML) projects on Cloud Computing (CC). Section 3.1 provides a background for the DML concept and establishes the features that characterize the design and implementation of DML applications. Based on these features, it summarizes the operation of the DML frameworks most used nowadays. It also highlights relevant research directions in the field. Section 3.2 presents the definition of CC, alongside its essential characteristics, service models, and deployment models. On this basis, the implementation of DML projects on different cloud service models is discussed. Finally, Section 3.3 presents the Tensor Processing Unit (TPU) accelerator as well as its basic architecture and operation characteristics. On this ground, the cloud-based computing pattern used for the experimental study of this thesis is described in detail.

3.1 DML Fundamentals

This section provides a background for the DML concept by presenting several research surveys on the topic, as well as a collection of selected projects considered as milestones for its recent development. On the ground of the feature taxonomy from [Langer et al., 2020] the elements for characterizing approaches to DML is discussed. The aforementioned taxonomy is then used to summarize the operation of five of the most used software frameworks for

DML. Finally, additional research lines that stand out in this context are presented.

3.1.1 Background

Galakatos et al. [2017] define DML as multi-node ML algorithms and systems that are designed to improve performance, increase accuracy, and scale to larger input data sizes. This definition is presented alongside an overview of parallelization details for techniques like gradient descent, regression, and k-means clustering, as well as alternative parallelization techniques like ensemble learning techniques and model parallelism. Peteiro-Barral and Guijarro-Berdiñas [2013] have provided some background regarding the advantages of distributed environments, as well as an overview of distributed learning on very large data sets. They identified three main approaches to large-scale learning: design a fast algorithm, use a relational representation, and partition the data. In addition, this work establishes a fundamental classification of the context for distributed computing between datasets that are naturally distributed and artificially distributed. Finally, this work points out two types of distribution according to the nature of the data sets: horizontal fragmentation wherein subsets of instances are stored in different sites, and vertical fragmentation wherein subsets of attributes of instances are stored in different sites. Another overview of distributed machine learning is presented in Pop et al. [2016], including novel platforms, libraries, and cloud services that can be used to extract knowledge from unstructured and semi-structured large data sets. This work includes packages that enable distributed computing in popular ML environments, distributed platforms for ML, and cloud services for ML. Parallelization is also considered a suitable approach to the huge computing workloads required to apply Nearest Neighbors (NN) searching algorithms in high dimensions. A survey of notable advancements in the parallelization efforts of NN search is presented in Aydin [2014]. A section of this survey is devoted to recent GPU implementations, which are classified into three main types: brute force, Kd-Tree-based, and Locality Sensitive Hashing. Mohapatra et al. [2019] present a summary of methods for forecasting of financial time series and distributed learning-based approaches, which ranges from 1996 to 2016 and includes a valuable analysis of the advantages and disadvantages of each proposal. [Verbraeken et al., 2020] present a survey with the ecosystem of the most widely used systems and libraries for DML, as well as their underlying designs. [Mayer and Jacobsen, 2020] analyze and

compare 11 open-source DL frameworks and tools that put scalability into practice. They also present valuable categorizations of approaches on relevant topics for data-parallel training, such as parameter synchronization and efficient communication. [Langer et al., 2020] analyze various distributed DL approaches, and offer insight into the motivations and concepts behind the different design choices made by each approach. On this basis, they design a valuable taxonomy for classifying the existing literature on distributed DL systems.

It is worth noticing that the search for parallel and distributed computing has emphasized the ML algorithms and processes that are most demanding in terms of computing workloads, such as evolutionary algorithm-based optimization, nearest neighbors search, and neural network training. In this context, the following projects can be referenced as milestones for the development of DML.

- In 2012, [Dean et al., 2012] from Google, presented DistBelief, a software framework intended to train large models for DL and unsupervised feature learning, which is able to train a deep network with billions of parameters using tens of thousands of CPU cores. DistBelief was successfully tested for speech recognition and visual recognition models with tens of millions of parameters.

- In 2016, as a succeeding project from DistBelief, Abadi et al. [2016] presented TensorFlow, a machine learning system that operates at large-scale and in heterogeneous environments, which uses data flow graphs to represent computation, shared state, and the operations that mutate that state. TensorFlow maps the nodes of a data flow graph across many machines in a cluster, and within a machine across multiple computational devices, including multi-core CPUs, general-purpose GPUs, and custom-designed ASICs known as TPUs.

- In 2017, Goyal et al. [2017] from Facebook, presented a distributed, synchronous version of the Stochastic Gradient Descent (SGD) algorithm which allowed them to train a ResNet-50 architecture on the ImageNet dataset (over 23 million parameters) with a minibatch size of 8192 on 256 GPUs over 32 servers in just one hour, using TensorFlow. This work is based on the adoption of a linear scaling rule for adjusting learning rates as a function of the minibatch size and the development

of a new warm-up scheme that overcomes optimization challenges early in training, and is considered the starting point of DML.

- In 2020, OpenAI beta-released GPT-3, an autoregressive language model based on DL that scales up to 175×10^9 parameters and produces human-like sequences of words, code, or other data, starting from a source input, referred to as the *prompt* [Floridi and Chiriatti, 2020]. The language modeling capability of GPT-3 is so high that people's ability to identify whether ~500-word articles are model generated (as measured by the ratio of correct assignments to non-neutral assignments) was 52% [Brown et al., 2020]. GPT-3 was trained on the Microsoft Azure cloud, using 10,000 chips of the NVIDIA V-100 GPU, at a combined computing cost of 3.14×10^{23} FLOPS and total energy consumption of 1,287 MWh [Patterson et al., 2021].

- In 2021, Google open-sourced its Switch Transformer [Fedus et al., 2022], a neural network architecture for NLP that scales up to 1.6×10^{12} parameters. The Switch Transformer is based on a *mixture-of-experts* layer that replaces the feed-forward layer of the original Transformer [Vaswani et al., 2017]. The resulting model is sparsely activated and, simultaneously, able to efficiently use hardware accelerators optimized for dense matrix computation. The Switch Transformer was trained on GC, using 1,024 chips of the TPU-v3, at a combined computing cost of 8.22×10^{22} FLOPS and total energy consumption of 179 MWh [Patterson et al., 2021].

Although this enumeration does not intend to be an exhaustive one, it is useful to highlight the fast expansion of the DML problem we have witnessed in less than a decade. Not only the number of trainable parameters in the Switch Transformer is 4 orders of magnitude larger than the parameter set of a typical execution of DistBelief, but also the computing performance of the hardware accelerators available to train such workloads has increased at a similar pace. In the following pages, I provide further detail on the features associated with the design and implementation of DML applications.

3.1.2 DML Features

A number of approaches to DML design and execution are available and several feature sets have been proposed to analyze and classify them. In

this work, I adhere to the taxonomy proposed by [Langer et al., 2020] because it provides a catalog with the fundamental characteristics that have a major impact on how DML systems operate. The main elements of this taxonomy are: data-parallel or model-parallel scaling-out strategy, centralized or decentralized optimization, synchronous or asynchronous scheduling, and parameter exchange and communication patterns.

Model Parallelism vs. Data Parallelism

DML workloads can be executed on the basis of model parallelism or data parallelism strategies. Model parallelism is intended for training a model that cannot fit its complete parameter set into the memory of a single computing device. Consequently, different layers of the ML model are assigned to different workers of the cluster. Thinking about a worker as a single machine that is part of a cluster makes it easier to mentally represent the distributed processing. However, it is important to keep in mind that the term *worker* refers to a software entity that can also be implemented as a computing device (CPU, GPU, TPU, FPGA) inside a machine, a specific set of cores inside a computing device, or even a virtual instance or a container running on the cluster infrastructure. Model parallelism can be understood as a technique that maps the model architecture (network layers) to the available infrastructure resources (computing workers). Usually, the complete training dataset $\mathcal{D}$ is passed to the worker node that executes the input layer of the model architecture, then the intermediate results are transferred to subsequent workers that compute downstream network layers. Gradients are computed starting from the outermost worker, the one which holds the output network layer, and back-propagated towards the worker that holds the input layer. Model parallelism tends to be far more complex to implement than data parallelism because the definition of tasks that can be executed in parallel is not trivial and highly dependent on the model architecture. In general, the parameters of a ML model are not independent of each other, therefore parallel updates to the model have to be restricted to subsets of mutually independent (or weakly correlated) parameters [Xing et al., 2016]. Kim et al. [2016] affirm that model parallelism is motivated by two challenges that data parallelism does not usually address: (1) parameters may be dependent, thus naive concurrent updates can introduce errors that slow convergence or even cause algorithm failure, and (2) model parameters converge at different rates, then a small subset of parameters can bottleneck

the algorithm's completion. Consequently, they propose Scheduled Model Parallelism (SchMP), a programming approach that improves ML algorithm convergence speed by efficiently scheduling parameter updates, taking into account parameter dependencies and uneven convergence.

On the other hand, data parallelism is a collaborative processing approach where the training dataset $\mathcal{D}$ is partitioned into P parts and each part is assigned to one of the available P computing workers. The model parameter set (the architecture and its state) is shared by all the workers, and each worker applies the same algorithm to a different dataset partition $\mathcal{D}^j$, for $j = 0, 1, ..., P$. Data parallelism can be generically applied to any ML algorithm with an independent and identically distributed (i.i.d.) assumption over the training data samples [Xing et al., 2016]. In this direction, ML optimizers like SGD are based on the assumption that the aggregate of the per-parameter gradients obtained from taking samples from subsets of a large set are approximately identical to the per-parameter gradients obtained from the complete set [Langer et al., 2020], as follows

$$\frac{\partial \mathcal{L}(\mathbf{x} \sim \mathcal{D}; w)}{\partial w} \cong \frac{\partial \mathcal{L}(\mathbf{x} \sim \mathcal{D}^1; w)}{\partial w} + \cdots + \frac{\partial \mathcal{L}(\mathbf{x} \sim \mathcal{D}^P; w)}{\partial w} \tag{3.1}$$

where $\mathcal{L}$ is the loss function, $\mathbf{x}$ represents samples taken from subsets $\mathcal{D}^j$ [1], and w represents the current model parameters shared by all the workers in the distribution scheme.

Even though all major DML frameworks support model and data parallel strategies, efforts towards the consolidation of more straightforward data parallelism are the mainstream nowadays. Accordingly, the following paragraphs will be focused on the data parallelism strategy, leaving model parallelism outside of the scope of this thesis.

Centralized vs. Decentralized Optimization

According to [Langer et al., 2020], a ML model training process is based on two fundamental operations that are repeated on each training step t. On *gradient computation*, the current model parameters w_t and the minibatch $\mathbf{x}$ drawn from $\mathcal{D}$ are used to calculate the per-parameter gradients g_t as

$$\mathbf{x} \sim \mathcal{D}, \quad g_t \leftarrow \frac{\partial \mathcal{L}(\mathbf{x}; w_t)}{\partial w_t} \tag{3.2}$$

[1] For simplicity, in this notation I omit the fact that different samples $\mathbf{x}_{j,s}$ are collected from the dataset partition $\mathcal{D}^j$ at the training step s.

Then, on *optimization*, the per-parameter gradients and the learning rate η are used to update the model state as

$$w_{t+1} \leftarrow w_t - \eta g_t \tag{3.3}$$

In DML training, the highly expensive gradient computation is always replicated across the available workers. Conversely, the less expensive optimization cycle can be either centralized on a master node, usually referred to as the *parameter server*, or replicated into the worker nodes. Both alternatives are detailed in the following paragraphs.

In centralized optimization the parameter server broadcasts a global model state $\tilde{w}_t$ to the P available workers (usually, the model state is randomly initialized to obtain $\tilde{w}_0$). The jth worker, for $j = 0, 1, \ldots, P$, samples the mini-batch $\mathbf{x}$ from its dataset partition $\mathcal{D}^j$ and computes per-parameter gradients from 3.2 as

$$\mathbf{x} \sim \mathcal{D}^j, \quad g_t^j \leftarrow \frac{\partial \mathcal{L}(\mathbf{x}; \tilde{w}_t)}{\partial \tilde{w}_t} \tag{3.4}$$

then, the parameter server receives the P gradients from the workers and consolidates them into a new global model state, from 3.3, as

$$\tilde{w}_{t+1} \leftarrow \tilde{w}_t - \eta \sum_{j=1}^{P} g_t^j \tag{3.5}$$

then the parameter server broadcasts the new global model state as $\tilde{w}_t \leftarrow \tilde{w}_{t+1}$ to the workers.

In decentralized optimization both the gradient computation and optimization cycles are performed by the workers independently. Unlike the parameter server used in centralized optimization, a master program, which can be executed simultaneously on more than one worker, broadcasts the global model state $\tilde{w}_t$ to all workers. The workers update their local copy of the model state to the global state $w_t^j \leftarrow \tilde{w}_t$, then compute gradients as

$$\mathbf{x} \sim \mathcal{D}^j, \quad g_u^j \leftarrow \frac{\partial \mathcal{L}(\mathbf{x}; w_u^j)}{\partial w_u^j} \tag{3.6}$$

and optimize as

$$w_{u+1}^j \leftarrow w_u^j - \eta g_u^j \tag{3.7}$$

where $u \leftarrow t+1, t+2, \ldots, t+\tau$ is an interval of τ steps provided to each worker to independently train its own local model before sharing results with

other workers. Usually, the value of τ corresponds to the steps required for the worker to train a minibatch. At the completion of this task, the workers upload local model states $w^j_{t+\tau}$ to the master program, which evaluates a new global model state as

$$\tilde{w}_{t+\tau} \leftarrow \frac{1}{P} \sum_{j=1}^{P} w^j_{t+\tau} \tag{3.8}$$

and the clocks in the master program and all the workers are updated to $t \leftarrow t + \tau$.

Synchronous vs. Asynchronous Scheduling

In data-parallel training, progressive refinements on the model state made by different parallel workers must be periodically merged into a global model. This situation leads to an important trade-off: if the workers are required to share their local model updates only after reaching the same state of progress then the training algorithm risks being delayed by potentially slower workers; conversely, if the workers are allowed to progress at their own specific pace (not forcing them to await for a consensual global update), then the model risks the loss of training quality or convergence speed. Three scheduling approaches have been developed to deal with this trade-off: *synchronous*, *asynchronous*, and *bounded asynchronous*. I provide further detail on them in the following paragraphs.

In synchronous scheduling, the model parameters are globally synchronized on every iteration of the training algorithm, that is every time the workers -as a group- complete processing a minibatch $\mathbf{x}$ drawn from $\mathcal{D}$. Therefore, synchronous scheduling is particularly adequate for infrastructure based on homogeneous workers (there are no potentially slower nodes), with low communication costs (as more parameter communication is required to strictly synchronize the local model updates). This is the case of single-machine configurations using multiple hardware accelerators as workers. From these considerations, it can be noticed that (3.4) and (3.5) refer to a centralized synchronous training implementation (the parameter server awaits for all workers to complete gradient computation on step t), while (3.6), (3.7), and (3.8) depict the decentralized synchronous training case (the master program aggregates local updates to the global state only after all workers train their model version for τ steps).

In asynchronous scheduling, no worker is enforced to wait for the others to achieve the same progress, which results in a higher overall hardware utilization. Consequently, a faster worker is allowed, once it has finished an iteration, to update the global model state, and to immediately download this new global state and use it for its next gradient computation. In the centralized asynchronous case, this means the jth worker downloads the current global state and use it to set its local state as $w^j \leftarrow \tilde{w}$ (the step identifier t is omitted as the clocks from the parameter server and the worker are not synchronized and most probably they differ), then the worker performs gradient computation as

$$\mathbf{x} \sim \mathcal{D}^j \quad g^j \leftarrow \frac{\partial \mathcal{L}(\mathbf{x}; w^j)}{\partial w^j} \tag{3.9}$$

and uploads the per-parameter gradients to the parameter server, The parameter server consumes these gradients to update the model state as

$$\tilde{w} \leftarrow \tilde{w} - \eta g^j \tag{3.10}$$

and sends $\tilde{w}$ back to the jth worker. As a consequence, in asynchronous training, there is only one worker that holds the most recent state of the global model and the rest of the workers operate on locally cached, *stale* versions.

Bounded asynchronous scheduling provides an intermediate solution for the convergence quality/speed to hardware utilization trade-off. It allows workers to operate on stale parameters but defines clear bounds for staleness. [Ho et al., 2013] proposed the Stale Synchronous Parallel approach, which enables the faster workers to progress on training for a given number of iterations δ before sharing parameter updates. Any worker that progresses for more than δ iterations with respect to the slowest worker is forced to wait until the remaining workers catch up. [Mayer and Jacobsen, 2020] affirm that, unlike *barrierless* or full-asynchronous approaches, bounded asynchronous scheduling allows for better mathematical analysis and proof of the model convergence properties. The work of these authors provides a categorization of recent publications on parameter synchronization in data-parallel training, which is useful for a more comprehensive study of DML scheduling.

Communication Pattern

Regardless of the optimization and scheduling techniques applied to a particular data-parallel DML workload, results from gradient computation and

optimization operations have to be communicated across the cluster. Parameter servers, master, and worker nodes are constantly sharing information and the network topology they are implemented on is crucial to achieving adequate speed. In the following paragraphs, I briefly present the communication patterns mainly used for DML executions.

A *tree-based* topology arranges nodes in hierarchical parent-child relationships. It is a simple layout that enforces any node to only share computing results with the nodes in the up- or downstream direction, as in Elastic Averaging SGD (EASGD) [Zhang, 2016]. The tree-based topology can also be used to implement a decentralized *all-reduce* communication strategy [Agarwal et al., 2014], where gradient computations are accumulated from the lowermost levels all the way up to the root node, then this uppermost node broadcasts the resulting global state downwards to all the nodes in the cluster.

A *ring-based* topology communicates each node in the cluster with only two other nodes, therefore the complete cluster can be thought of as a closed ring. Compared to fully-connected or tree-based patterns, the ring topology produces less communication overhead, which makes it adequate for situations that require minimizing networking operations. In conjunction with the all-reduce paradigm, *ring-all-reduce* [Sergeev and Del Balso, 2018] provides a robust communication pattern where each node constantly influences and regularizes its neighbors' operation. As this worker behavior cascades the ring, results found by each node eventually travel in both directions around the cluster, which provides increased stability to the training process.

Other topology patterns used in conjunction with the all-reduce paradigm are the Round-Robin and the Butterfly variants [Zhao and Canny, 2013]. [Chahal et al., 2020] present all-reduce variants like Recursive Halfing/Doubling and Binary Blocks, and propose a variant called Tolerant All-Reduce which is capable of providing fault tolerance in an unstable networking environment.

3.1.3 DML Frameworks

At the time of this writing, tens of different DML frameworks are extensively used for both research and production deployments. As an introduction, I present here only five of the most widely used, which are Apache Spark, TensorFlow, PyTorch, Horovod, and MXNet. It is worth mentioning all of our experiments were developed on TensorFlow.

Apache Spark

Apache Spark [Zaharia et al., 2016] is a powerful, general-purpose, in-memory framework for distributed processing, and a crucial component for the Hadoop - HDFS [Shvachko et al., 2010] ecosystem. As of this writing, it is the de facto standard for advanced data analytics on big data sets Hai and Forouraghi [2018]. Spark is the result of a project started in the Algorithms, Machines, and People Lab (AMPLab), at the University of California at Berkeley. The core concept behind Spark operation is a data structure called Resilient Distributed Dataset (RDD), which provides a basis for in-memory processing, workload distribution, and fault tolerance Zaharia et al. [2012]. Spark was developed as a general-purpose analytics framework, and it integrates specialized libraries, such as MLlib [Meng et al., 2016], to execute ML workloads. Spark provided a basis for early DML frameworks such as TensorFlow-On-Spark [Lu et al., 2018], Caffe-On-Spark [Langer et al., 2020], or SparkNet [Moritz et al., 2015].

TensorFlow

TensorFlow [Abadi et al., 2016], from Google, is a ML system that operates at large scale and in heterogeneous environments. It uses data flow graphs to represent computation, shared state, and the operations that mutate that state. As of this writing, TensorFlow is, by far, the most used native DML system. It implements data parallelism with centralized and decentralized optimization, as well as synchronous and bounded asynchronous scheduling. TensorFlow implements model parallelism via the Mesh TensorFlow [Shazeer et al., 2018] language extension. Native functionalities in TensorFlow include all-reduce and parameter server communication patterns.

PyTorch

PyTorch [Paszke et al., 2019] is a DML framework developed by Facebook and community members. As of this writing, it has the highest growing interest rate in specialized forums, arguably due to its easier and more intuitive interface, when compared to other frameworks. PyTorch implements data parallelism with centralized and decentralized optimization, as well as synchronous and asynchronous scheduling.

Horovod

Horovod [Sergeev and Del Balso, 2018] from Uber, is an open-source library that runs on top of TensorFlow and improves on two typical obstructions to scaling ML operations: it provides efficient inter-GPU communication via ring-all-reduce, and enables distribution with only a few lines of modification to the original TensorFlow code. Horovod makes use of the NVIDIA NCCL, a library for collective communication that provides a highly optimized version of ring-all-reduce.

MXNet

Apache MXNet [Chen et al., 2015] is, as TensorFlow, a modern descendant of DistBelief which abstracts ML models and states as data flow graphs. It supports model parallelism on single-machine, multiple-device configurations. Data parallelism is implemented with centralized optimization as well as with synchronous or bounded asynchronous scheduling.

3.1.4 Additional Research Directions

I briefly present now two research directions that stand out in the current context for DML: parallelization efforts that are mainly based on hardware devices and ML initiatives over datasets that are naturally distributed.

Hardware-based Parallelization

FPGA-based accelerators have attracted the attention of researchers because of their good performance, high energy efficiency, and great flexibility Guan et al. [2017]. [Ovtcharov et al., 2015] used FPGA to accelerate the training of deep CNNs. Ferreira and Fonseca [2016] proposed a hardware architecture for LSTM neural networks, aiming to speed up its operation by means of FPGA's inherent parallelism, as well as to achieve a reduced power consumption. Guan et al. [2017] have presented an FPGA-based accelerator for LSTM-RNNs that optimizes both computation performance and communication requirements. [Zhang et al., 2017] presented a hardware accelerator for the LSTM neural network based on the FPGA Zedboard that is power-efficient and shows a better performance than ARM Cortex A9 and Intel Core i5 processors.

Naturally Distributed Datasets

DML started as a strategy to deal with the problem of processing huge data sets. In this context, centralized information was distributed to different nodes, clusters, or sites to efficiently perform computing operations on it, then make the results available to a central location for making a global decision. This means the information context is artificially distributed. However, in most current scenarios, datasets used for research are not available in a commonplace rather they are distributed across several geographical regions [Mohapatra et al., 2019]. In the Big Data era information is naturally distributed, and this fact places an important requirement on the research and development of ML frameworks to work across multiple, remote, and heterogeneous computing nodes. As a consequence, data communication and network performance increase their value as key research topics in this field. Wang et al. [2018a] affirm that the IoT, social networks, and crowdsource operations generate large amounts of data at the network edge, and that the problem of learning model parameters from data that is naturally distributed across multiple edge nodes emphasizes the question of how to efficiently utilize the limited computation and communication resources at mobile edge computing for the optimal performance of DML applications.

3.2 Cloud Computing for DML

This section introduces the definition of CC, as stated by the National Institute of Standards and Technology (NIST). The five essential characteristics, three service models, and five deployment models found in the definition are briefly described. The cloud service models provide a well-established basis for implementing DML applications, then the general procedures for all implementation cases are presented.

3.2.1 Cloud Computing Fundamentals

The NIST provides the official definition of CC in [Mell et al., 2011]: *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service*

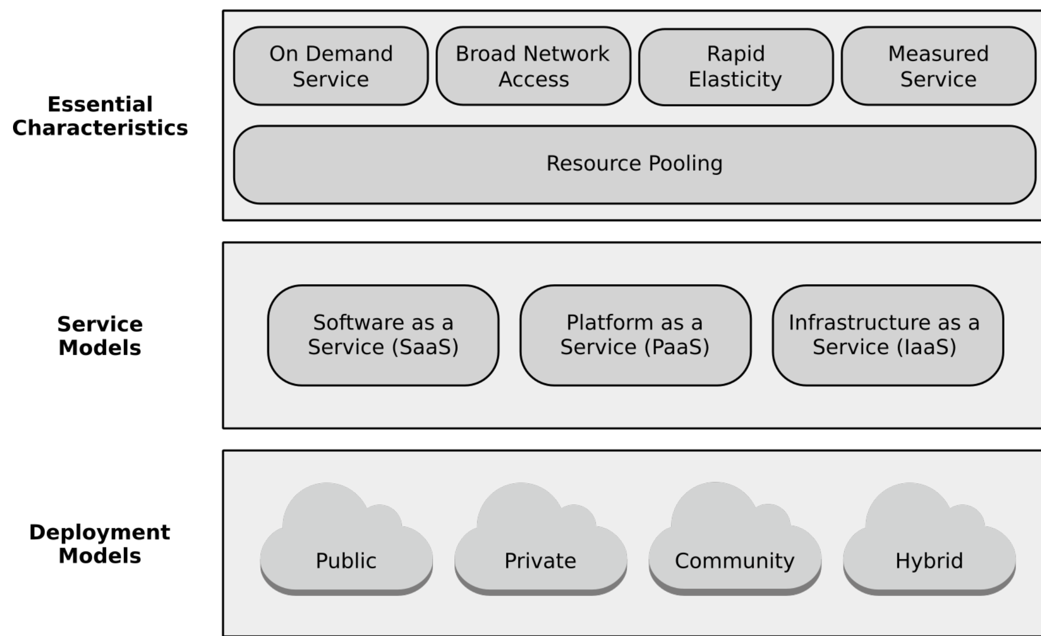


Figure 3.1: Cloud computing characteristics, service, and deployment models.

models, and four deployment models. The elements in the definition are depicted in Figure 3.1. Following are their complete descriptions as extracted from the aforementioned source.

Cloud Essential Characteristics

1. *On-demand self-service*. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

2. *Broad network access*. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms[2] (e.g., mobile phones, tablets, laptops, and workstations).

3. *Resource pooling*. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.

4. *Rapid elasticity*. Capabilities can be elastically[3] provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

5. *Measured service*. Cloud systems automatically control and optimize

[2] A thin client platform is a lightweight computer that basically monitors and controls processes executed in the server side, so it can work efficiently without advanced processing or storage components. On the other hand, a thick client platform usually performs most of the processing workload over data stored on the server side.

[3] In this context elasticity means the ability to flexibly and rapidly increase or decrease the quantity of IT resources to adjust them to the experienced workload.

resource use by leveraging a metering capability[4] at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Cloud Service Models

1. *Software-as-a-Service (SaaS).* The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure[5]. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

2. *Platform-as-a-Service (PaaS).* The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider[6]. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

3. *Infrastructure-as-a-Service (IaaS).* The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and

[4] Typically this is done on a pay-per-use or charge-per-use basis.

[5] A cloud infrastructure is the collection of hardware and software that enables the five essential characteristics of cloud computing. The cloud infrastructure can be viewed as containing both a physical layer and an abstraction layer. The physical layer consists of the hardware resources that are necessary to support the cloud services being provided and typically includes server, storage, and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests the essential cloud characteristics. Conceptually the abstraction layer sits above the physical layer.

[6] This capability does not necessarily preclude the use of compatible programming languages, libraries, services, and tools from other sources.

run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

Cloud Deployment Models

1. *Private cloud*. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on- or off-premises.

2. *Community cloud*. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on- or off-premises.

3. *Public cloud*. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, government organization, or some combination of them. It exists on the premises of the cloud provider.

4. *Hybrid cloud*. The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting[7] for load balancing between clouds).

3.2.2 DML on the IaaS Service Model

Under the IaaS service model, the capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing

[7] Cloud bursting refers to the ability to get infrastructure at runtime from a cloud provider and enabling the applications to use that to meet the service levels during a temporary spike in the workload [Doddavula et al., 2013].

resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. Hence, the IaaS service model maximizes flexibility for the DML workflow design and execution at the cost of increased effort during implementation and maintenance. Following is an enumeration with the primary IaaS components that must be deployed to execute a generic DML workflow, along with their end goals:

1. Virtual Machine (VM). This component provides computing resources for DML workflows. A suitable framework for distributed training must be installed on one or more VMs along with software dependencies such as package managers, numeric computation libraries, and parallel-specific components like the Compute Unified Device Architecture (CUDA) Application Programming Interface (API) or the communication libraries NVIDIA Collective Communication Library (NCCL) and Message Passing Interface (MPI).

2. Binary Large Object (*blob*) storage. This component provides storage units for massive amounts of unstructured data (text, images, video, etc.) usually required for training a ML model. Semi-structured data can be blob-stored via key-value-based file formats like JSON. Structured data can be either serialized to fit in blob storage or imported from relational databases or PaaS components such as data warehouses. High-bandwidth networking allows extremely fast communication between VMs and binary storage.

3. Hardware-based accelerator. This component extends the computing capability available in VMs by off-loading specific tasks from general-purpose CPUs to integrated circuits that are optimized for parallel computation. Hardware accelerators are used in many domains like computer graphics, signal processing, cryptography, and Deep Learning (DL). Due to its long presence in the parallel computation ecosystem, the GPU is the dominant alternative for hardware-accelerated DML. Field-Programmable Gate Array (FPGA)s are increasingly used for accelerating DML training and inference stages due to the inherent parallelism and the deeply-pipelined computation they offer [Kara et al., 2017]. Recently, Application-Specific Integrated Circuit (ASIC)s for cloud-based DL were introduced by GC (TPU) and AWS (Trainium and Inferentia). Hardware accelerators lie in a diffuse spot in the border between the IaaS and PaaS model services, as their utilization requires

dedicated VMs and networking whose management is outside the capability of the consumer.

4. Virtual private cloud. This component provides a logically isolated section of the cloud infrastructure where customers are able to deploy resources under a highly customized network. Sub-networks and firewalls can be defined inside the virtual private cloud to enforce the security of operations.

5. Publication/Subscription (Pub/Sub) channel. This component asynchronously communicates all of the workflow components (applications) by exchanging messages. It implements a message-oriented middleware pattern [Fehling et al., 2014] where incoming messages are elastically queued and transformed to comply with each receiver's specifications.

6. Relational database. This component provides an adequate persistence layer for ML artifact metadata, such as dataset location and statistical description, hyperparameter value sets, model version and evaluation metrics, batch inference results, etc. A relational database instance is crucial to managing the ML model's lineage (the specific dataset, features, architecture, and hyperparameter set that produced it) hence to obtain better variants by combining different artifact versions.

3.2.3 DML on the PaaS Service Model

Under the PaaS service model the capability provided to the consumer is to deploy onto the cloud infrastructure applications created using programming languages, libraries, services, and tools managed by the provider. Based on this model service, all major public cloud providers offer complete, production-ready ML platforms referred to as *Machine Learning-as-a-Service (MLaaS)*. MLaaS functionality is intended to cover the complete life cycle of ML workflows, including dataset management and registry, data validation and preprocessing, feature engineering and registry, model training (including pretrained models for rapid development), interactive notebooks for workflow prototyping, hyperparameter tuning, model evaluation, model deployment to internet endpoints for inference, ML-oriented pipeline authoring, integrated repositories for metadata, datasets, features, models, inference results, etc. A number of options for distributed processing at the training and inference stages are also available.

Relevant examples of MLaaS are Vertex AI from GC [Katti et al., 2022], Amazon SageMaker from AWS [Joshi, 2020], and Azure ML from Microsoft Azure [Team, 2016]. Vertex AI enables DML execution for TensorFlow or PyTorch models, using single- or multi-host data-parallel training with GPUs and the NCCL all-reduce. Hardware-based parallelization is also available in Vertex AI using TPUs with TensorFlow-based models. Amazon SageMaker provides DML libraries for TensorFlow and PyTorch. Hardware-based parallelization based on GPUs is available for single- and multi-host configurations. As of this writing, AWS ASICs Trainium and Inferentia are available in SageMaker, the former only in preview mode. Both data parallelism and model parallelism can be implemented, and extended memory-optimization features are available for model-parallel training in PyTorch. Azure ML enables DML workloads for TensorFlow- and PyTorch-based architectures. Further optimization is provided for these frameworks via Horovod for GPU-accelerated single- and multi-host configurations. In addition, distributed inference can be executed on Azure ML using FPGAs.

3.2.4 DML on the SaaS Service Model

Under the SaaS service model the capability provided to the consumer is to use the provider's applications that run on a cloud infrastructure the consumer does not have control over. This is the case of fully-managed (serverless) applications referred to as *AutoML*. AutoML services enable the consumer to process his own datasets to solve a specific use-case that fits on generic ML tasks, such as object recognition, time series forecasting, sentiment analysis, or video tracking, to mention a few. Although AutoML services are able to distribute training or inference stages, the customer does not have direct control over parallelization design choices. Relevant examples of AutoML can be found in GC Vertex AI and the BigQuery data warehousing service, in AWS' AutoGluon and SageMaker, and in AutoTrain from Hugging Face [Wolf et al., 2019].

3.3 Tensor Processing Units

This sections presents the TPU hardware-based accelerator, which is the foundation for the DML application implemented for the experimental study of this Thesis. The TPU architecture and operation characteristics are de-

scribed in detail. On this ground, a TPU-based CC pattern deployed using the IaaS cloud service model of GC is proposed to train the selected forecasting DL models.

3.3.1 Background

In recent years, the massive computing requirements placed by ML workloads led to the continuous development of specialized hardware, like high-performance CPUs, GPUs, and TPUs. Since its public release, TPU accelerators have been applied to a diversity of research topics that demand very high processing and memory capabilities. Examples of these domains are Deep Neural Network (DNN) training subject to large batch sizes and specialized learning rate algorithms [You et al., 2019], [Wongpanich et al., 2021], distributed evolution strategies for meta-learning [Sheng and He, 2020], acceleration of explainable machine learning [Pan and Mishra, 2022], and simulation of quantum physics [Hauru et al., 2021], to mention a few. In a scope closer to this thesis, TPUs have also been used to accelerate the training of Transformer-based neural networks. NLP and CV projects constitute the majority of these research cases, partially due to the high availability of pre-trained models and TPU-ready software implementations. Examples of the application of TPU-assisted Transformers can be found in machine translation [Xu et al., 2021a], language modeling [Alrowili and Vijay-Shanker, 2021], image generation [Zhao et al., 2021], and visual recognition [Srinivas et al., 2021]. Regarding TSF research, TPUs have been applied to weather forecasting via convolutional networks enriched with a one-layer self-attention encoder block [Abdellaoui and Mehrkanoon, 2020], and short-term wind speed forecasting via symbolic regression [Abdellaoui and Mehrkanoon, 2021].

3.3.2 TPU Basic Characteristics

A TPU is an ASIC developed by Google to accelerate large-scale ML. Google Cloud (GC) first released TPU v1 in 2018 and, at the time of this writing, its highest publicly available version is TPU v3. Each one of the 8 cores in a TPU v3 board provides a Matrix Multiplier Unit (MXU) with 65,536 8-bit multiply-and-add units, a Unified Buffer (UB) with 24MB of SRAM, and an Activation Unit (AU) with hardwired activation functions. These computational resources are designed to process basic neural network-related operations at a very low level: the MXU performs matrix operations, while

the UB holds intermediate results, and the AU performs the nonlinear operations over the MXU output [Jouppi et al., 2017]. A Complex Instruction Set Computer (CISC) controls the TPU resources with an instruction set also designed for neural network-specific operations. The core of TPU logic is a *systolic array* [Kung and Leiserson, 1979]. This chip architecture accelerates matrix multiplications on the MXU by making the input values flow through fixed patterns of multiply-and-add units, substantially reducing read-write operations to the UB.

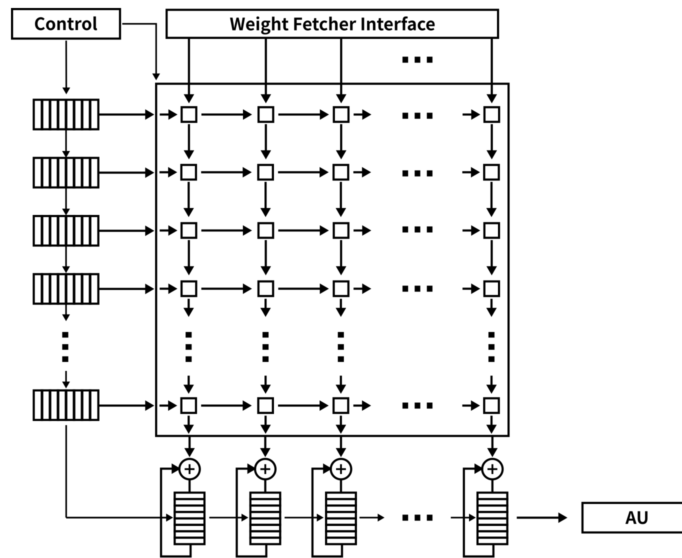


Figure 3.2: Systolic data flow of the MXU. Based on [Jouppi et al., 2017].

In Figure 3.2, weights are pre-loaded from the Weight Fetcher Interface into the computation cells inside the MXU. Data flows in at regular intervals from the vertical stack of systolic data setup blocks (left). A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. Partial sums are collected on the horizontal stack of accumulators (bottom). Results flow out the MXU at the bottom-right corner into the Activation Unit. For a detailed description of the elements and flows in this figure, the reader is advised to consult [Jouppi et al., 2017].

TPU accelerators are only available via Cloud TPU, a high-performance computing service of GC. Cloud TPU provides a TPU board connected via PCI to a host VM. Inside this combination, the TPU runs the consuming model training and evaluation stages in parallel. TPU computing perfor-

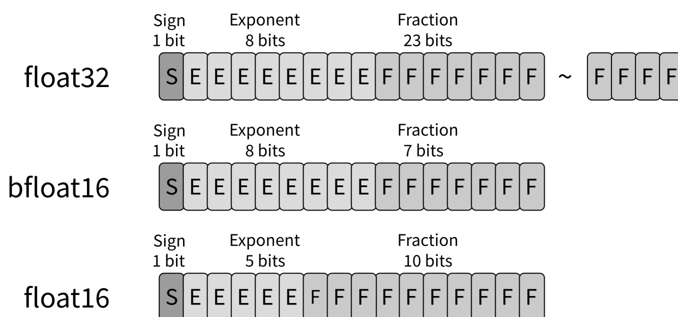


Figure 3.3: Comparison of the float32, bfloat16, and float16 numerical formats.

mance is extremely high (up to 420 TFLOPS) and data in-feed usually becomes an operation bottleneck. For this reason, the host VM is used to execute code that feeds data to the TPU as fast as possible. TPU hardware is language-agnostic, so an Accelerated Linear Algebra (XLA) compiler maps the ML-code representations from a specific programming language to TPU machine code. The XLA compiler is designed for linear algebra and vector computations; any programming construct that relies on different operations has to be directed to the host VM's CPU. The MXUs of a TPU can perform at *mixed precision*, combining different numerical formats in the same computational workload. TPU cores can use `bfloat16` format for multiplications and `float32` format for results accumulation. As most DL applications do not need a high precision to get the target application accuracy [You et al., 2019], floating point operations, and consequently training processes, are accelerated if weights, gradients, data, and activation outputs are represented in bfloat16 format. Figure 3.3 compares float32, bfloat16, and float16 numerical formats. The bfloat16 format implements the same range as the float32 format but with lower precision. TPUs build mixed precision as a combination of bfloat16 and float32, while GPUs commonly used for ML workloads use a combination of float16 and float32. The TPU implements data parallelism, as training data batches are split into the eight cores of the board. The model, on the other hand, is not distributed, then each core is required to maintain a copy of it on memory [Wang et al., 2019b].

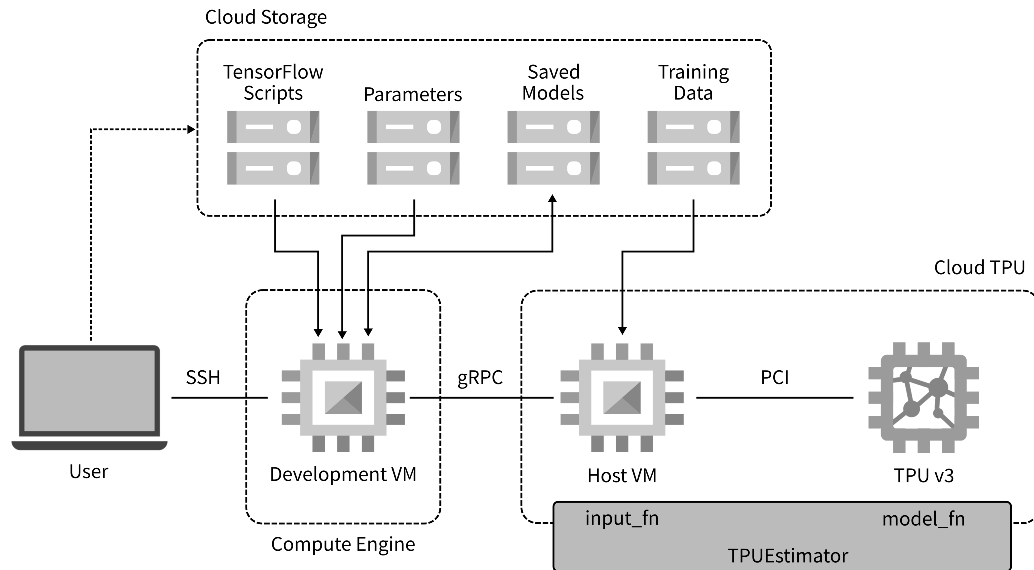


Figure 3.4: Cloud-based TPU Computing Pattern.

3.3.3 TPU-based Cloud Computing Pattern

Figure 3.4 shows the cloud-based TPU computing pattern deployed on GC for training the DL forecasting models. The dotted boxes represent the three fundamental cloud computing services of this deployment: Cloud Storage is a Binary Large-object storage service, Compute Engine is a Virtual Machine (VM) provisioning service, and Cloud TPU is a high-performance computing service. Communication between Compute Engine, Cloud Storage, and Cloud TPU is based on Google Remote Procedural Calls (gRPC) over high-speed, in-data-center networking. In order to set up and execute the complete application, the client connects to a development VM using the Secure Shell (SSH) protocol.

I programmed the algorithms with TensorFlow [Abadi et al., 2016] as front end, using `TPUEstimator`, a high-level API that simplifies TPU usage by handling numerous low-level, hardware-specific details, and is built on the basis of `Estimator`, a high-level API for TensorFlow. `TPUEstimator` presents an interface similar to `SciKit-learn`, and some adaptations to simplify its deployment to production stages Cheng et al. [2017a]. The `Estimator` class is the base for executing the different stages of the ML model (training, evaluation, serialization, and inference) from the same base code expressed

in a `model_fn` function. A user-defined function called `input_fn` produces the input for the `Estimator`, decoupled from the `model_fn`. Based on this decoupling, the TPU runs the `model_fn`, which encloses anything involving trainable variables or gradients. The host VM's CPU runs the `input_fn`, which focuses on data preparation and in-feed. The `TPUEstimator` API implements decentralized optimization via *all-reduce*, synchronous-scheduling data parallelism, which means it evenly splits the training batch among the workers (cores) of the TPU board. Conversely, the ML model is not distributed and every worker keeps a replication of the TensorFlow computation graph. Workers operate synchronously, with each worker performing the same step simultaneously. A reduction operation is performed across all the workers once the training step is completed.

For cloud-based implementation of the ML-based forecasting algorithms, I built an advanced architecture that organizes the involved cloud services as follows: A development VM retrieves TensorFlow scripts and parameters from Cloud Storage, spins-up a Cloud TPU as a training co-processor, sends the TensorFlow computation graph to the TPU's host VM and, during training, writes the resulting serialized model back to Cloud Storage. TPU's host VM pulls data directly from Cloud Storage and distributes it to the TPU cores. In accordance with best-TPU-performance guidelines, raw and preprocessed data is persisted in Cloud Storage as sequences of binary strings (`TFRecords`). A data ingestion pipeline for training the models was defined on the basis of the `tf.data.dataset` TensorFlow API. Pre-fetching was also used for parallel preprocessing, that means the application is able to start preparing the next batch of data while still training the current batch. To ensure good throughput during ingestion from Cloud Storage, individual `TFRecord` files were prepared for each time series fed to the model.

Summary

This Chapter discusses the implementation of DML applications on CC. It provides a research background for the DML concept in general, as well as specific features regarding workflow design choices and available software frameworks. It also addresses the development of DML projects on different cloud service models. On this ground, a TPU-based CC pattern for training the DL forecasting models in this thesis is proposed.

The next Chapter presents the implementation and experimental results of the first DL model trained on the TPU computing pattern: a deep multi-sequence, LSTM neural network for short-term, electric power load forecasting.

Chapter 4

Deep Multi-Sequence LSTM Architecture

This Chapter presents in detail the Deep Multi-Sequence Long Short-Term Memory (DMSLSTM) architecture that I designed and implemented for local TSF. Section 4.1 presents previous research work related to expanding the original RNN and LSTM networks for forecasting purposes. Section 4.2 presents the electric power dataset used for local forecasting experiments. It also presents in detail the architecture and the parametrization of the model. Section 4.3 presents the baseline model that used to evaluate the DMSLSTM performance. On this basis, results achieved by DMSLSTM concerning forecasting accuracy, computation times, and predictive performance degradation are discussed.

4.1 Related Work

Among ML-based methods for TSF, RNNs have received special attention from research on the electric power domain because of their ability to keep a *memory* of the dynamic processes that lie beneath power-related time series. Bianchi et al. [2017] reviewed five of the most frequently used RNN architectures for electrical load short-term forecasting: Elman Recurrent Neural Network (ERNN), Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), Non-linear Auto-Regressive with Exogenous Inputs (NARX), and Echo State Network (ESN). This work affirms there is not a specific RNN model that outperforms the others in every prediction problem, and

the choice of the most suitable architecture depends on the specific task at hand. LSTM, a class of RNN that incorporates special gates for internal connections to deal with the problem of *vanishing/exploding* gradients Hochreiter and Schmidhuber [1997], comes forth as the most reported architecture for ML-based TSF in the electric power sector. Electrical load accounts for the majority of reported research on LSTM and RNN-based forecasting Cheng et al. [2017b] Kong et al. [2017b] Liu et al. [2017] Zheng et al. [2017b] Zheng et al. [2017a], Jiao et al. [2018] Kumar et al. [2018] Wang et al. [2018b] Zhang et al. [2018] Ke et al. [2019] Wang et al. [2019a] Zhang et al. [2020] however, other explored variables include wind power generation Devi et al. [2020], photovoltaic power generation Abdel-Nasser and Mahmoud [2019] Li et al. [2019a], and power quality indexes Eisenmann et al. [2019] Zhang et al. [2019].

This collection of projects shows the important efforts that have been made to expand the standard RNN-based forecasting model with additions aimed to improve accuracy, to extend prediction intervals, or to increase the granularity a relevant variable is predicted at. The following list attempts to categorize those contributions:

- *Multi-sequence forecasting.* A learning algorithm is trained over multiple sequences that result from applying different feature engineering schemes to the source time series. An LSTM-based model for non-residential, electrical load short-term forecasting that inputs sequences built from varying delay values is presented in Jiao et al. [2018]. Forecasting models based on LSTMs and RNNs are used in Zhang et al. [2018] to predict short-term electrical load on the basis of multiple sequences built from combinations of continuous and discrete time series (short-term, cycle, long short-term, and cross-long short-term).

- *Multi-source forecasting.* Time series from multiple variables, such as power, weather, and calendar data, conform a combined input for the forecast model. In Cheng et al. [2017b] a LSTM is trained on the basis of the 12 most significant features obtained from the historical load, weather, and calendar data. In Jiao et al. [2018] a LSTM-based model for short-term non-residential load forecasting is trained on the basis of historical load and calendar data. In Wang et al. [2018b], historical load, as well as weather, temperature, and calendar data are used as the input for a GRU-based model for load forecasting. Load and calendar

data are used as input for a LSTM-based model in Kong et al. [2017b]. A GRU-based forecast model which uses load, temperature, humidity, and date information as input is presented in Ke et al. [2019].

- *Multi-step forecasting*. The variable of interest is predicted for multiple time steps ahead (the output of the forecast model is a time series of predictions) using diverse forecasting strategies. In Cai et al. [2019], electrical load is predicted a day-ahead using gated-RNN and gated-CNN networks, on the basis of recursive and direct strategies. In Marino et al. [2016], a Seq2Seq LSTM architecture Sutskever et al. [2014] is used to predict load values for 60-timesteps ahead.

- *Consumer-level forecasting*. Electrical variables are not predicted at the system or substation level but over time series collected from smart meters, at the consumer level. This gives a better understanding of consumption patterns at the cost of handling a huge amount of local forecast models, or high-complexity global models. Load is predicted at building-level, using RNN-based models in Marino et al. [2016] and Cai et al. [2019]. In Kong et al. [2017a] a deep LSTM forecasting framework is enhanced by including electrical appliance consumption sequences. In Wang et al. [2019a] a traditional LSTM-based point forecasting is extended to probabilistic forecasting in the form of quantiles.

- *Heuristic-based optimization*. Parameters that define the RNN architecture or control its training stage are optimized on the basis of heuristic algorithms. Bouktif et al. [2018] use a genetic algorithm to find the optimal values for time lags and the number of layers for a LSTM-based model. An LSTM-based forecast model is optimized using *fruit-fly optimization algorithm* (FOA) in Peng et al. [2020]. The parameters optimized by FOA are the length of the input sequence, the number of hidden units, the batch size, and the number of epochs. In Zhang et al. [2020] LSTM weight parameters are optimized on the basis of a combination of genetic algorithms, pattern search, and particle swarm optimization.

- *Deep RNN*. According to Pascanu et al. [2013], deep learning models are proven to be exponentially more efficient (they may require exponentially fewer units to represent the same function) than shallow ML models. This work presents different ways to extend a RNN to a *deep*

RNN. However, most of the reported applications of deep RNNs in the power domain correspond to *stacked* RNNs. That is RNN architectures where a number of hidden state layers are piled up, one on top of the other. Stacked RNNs, particularly LSTMs, have been extensively applied to load forecasting, as in Kong et al. [2017b], Kumar et al. [2018], Mujeeb et al. [2018], Jiao et al. [2018], Ke et al. [2019], and Abdel-Nasser and Mahmoud [2019]. They have also been used to predict photovoltaic power Ke et al. [2019] and power system state Zhang et al. [2019].

- *Augmented RNNs.* Extra components for achieving specific tasks are added to the RNN-based architecture to enhance its predictive performance. Olah and Carter [2016] identify four directions that stand out in augmenting RNNs: Neural Turing Machines, Attention-based Interfaces, Adaptive Computation Time, and Neural Programmers. Regarding TSF in the power domain, attention mechanisms are the most reported technique for RNN augmentation. Attention was introduced in Bahdanau et al. [2014] to deal with issues detected when applying the RNN encoder-decoder architecture to neural machine translation. It gives a RNN the capability to focus on a subset of the input information when predicting a subset of the output. In Cinar et al. [2017b] an attention mechanism is used on top of a RNN for modeling periodicity and handling missing values in electricity and weather time series. In Fan et al. [2019], a temporal attention mechanism is designed to capture latent patterns in historical data for electricity price forecasting. Another direction for augmenting RNNs is to modify their cell architecture. In Devi et al. [2020], the standard LSTM structure is extended with an enhanced forget gate (EFG) to improve accuracy in wind power forecasting. EFG is based on the addition of two *peepholes* to the LSTM block, switching the activation function from *tanh* to $softsign$, deletion of the input gate, and subtracting the previous output value through the forget-gate layer from the impact of the entire single matrix.

4.2 Experimental Study

This section presents the basics of the experimental study with the DMSLSTM architecture. It discusses the electric power dataset used as the basis of the forecasting process. Later, it expands on the specifics of the forecasting model architecture. Finally, it reports on the DMSLSTM complete parametrization for the use case.

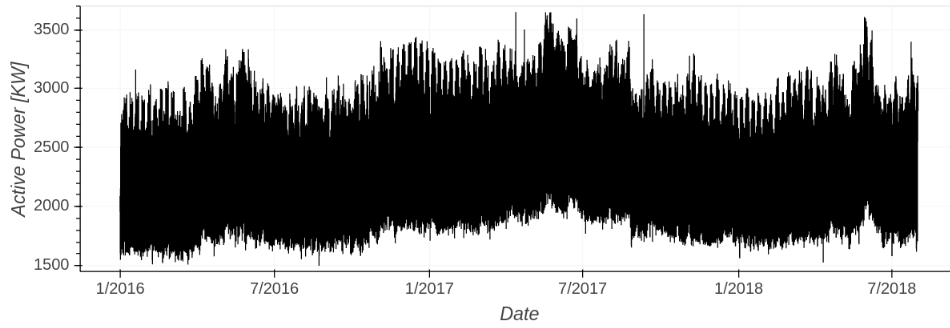
4.2.1 Dataset

I trained DMSLSTM on the active power readings recorded by an urban substation meter for 31 months. These readings account for the active power, expressed as Kilowatts, the substation delivers to a power sub-network located in southeast Morelia [1]. Table 4.1 shows the main statistics of the dataset. To produce the training, validation, and test datasets of the forecasting model, I divided the global dataset at the time series level, to avoid readings overlapping between stages and leaking information from the past into the future. I used a split percentage of 76/12/12 for training, validation, and test datasets, respectively. Important considerations emerge from the analysis of the electric power load profile at different intervals.

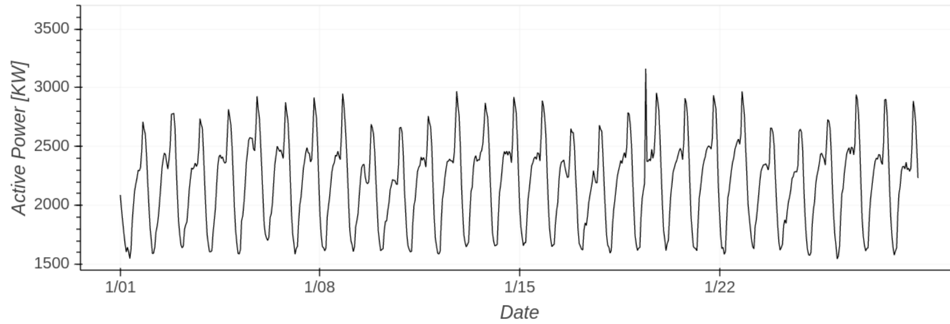
Figure 4.1a shows the complete dataset with more than 22K hourly readings recorded over 943 days. It is clear the active power variance remains approximately constant, although a subtle ascending trend is present. A seasonal, climate-related pattern shows the delivered active power increases in summer and decreases in winter. Figure 4.1b shows the first four weeks of the dataset. There is clearly a weekly pattern where delivered active power decreases during the weekends and Mondays, then increases for the rest of the week. Figure 4.1c shows the first week of the dataset. It is clear there is a daily pattern that consists of low active power values at the beginning of the day that increase up to a couple of peaks: one in the afternoon and a maximum at night time. This behavior is consistent with aggregated power consumption patterns in urban environments.

Table 4.1: Dataset statistics.

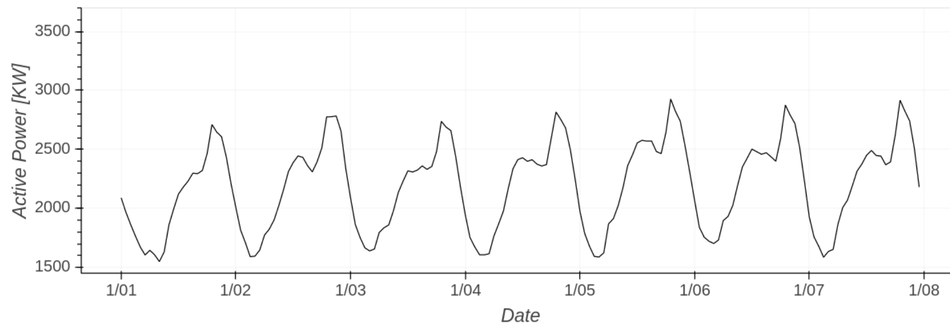
global statistics	
start datetime	2016-01-01 00:00:00
end datetime	2018-07-31 23:00:00
# of days in time series	943
# of readings	22,629
time granularity	hour
domain	$\mathbb{R}^+$
mean	2430.08
std. dev.	455.42
min	1498.95
25%	2020.80
50%	2471.60
75%	2793.38
max	3650.23
training time series	
start datetime	2016-01-01 00:00:00
end datetime	2017-12-31 23:00:00
# of readings	17,542
evaluation time series	
start datetime	2018-01-01 00:00:00
end datetime	2018-04-30 23:00:00
# of readings	2,879
test time series	
start datetime	2018-05-01 00:00:00
end datetime	2018-07-31 23:00:00
# of readings	2,208



(a) Complete dataset.



(b) First four weeks.



(c) First week.

Figure 4.1: Active power for an urban substation meter in southeast Morelia.

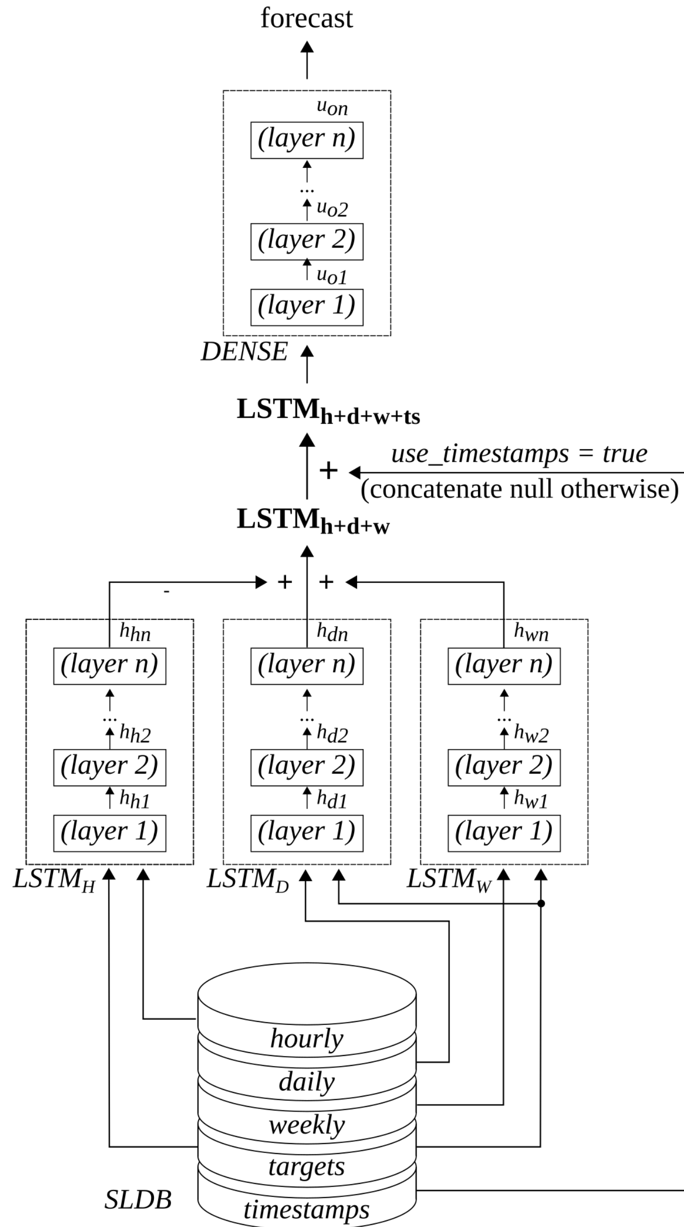


Figure 4.2: Deep Multi-Sequence LSTM Network Architecture.

4.2.2 Architecture Description

Implementing the DMSLSTM architecture comprises three stages: Supervised-Learning DataBase (SLDB) generation, model architecture definition, and model training. In the first stage, features, and targets (also known as *labels*) are extracted from historical time series of electric power load and stored in a SLDB. The SLDB constitutes what Goodfellow et al. [2016] call a *design matrix*. The SLDB structure reflects a design decision concerning the length of the sequences that act as input for the DMSLSTM network (the *conditioning range*). Decoupling the SLDB generation from the other two stages enable the user to build and persist many different SLDBs, then use them in multiple combinations with diverse model architectures or training configurations. In the second stage, a specific DMSLSTM architecture is defined on the basis of the input SLDB and the stacked LSTMs composite structure. Finally, in the third stage, the DMSLSTM is trained on the SLDB and the resulting model is evaluated and deployed for prediction.

Figure 4.2 illustrates the DMSLSTM architecture. The SLDB that provides an input for the DMSLSTM network is generated from the historical record, at 1-hour resolution, of the power load at the substation level, and the embedding dimensions m_h, m_d, m_w, used to build the hourly, daily, and weekly feature collections. Once generated, the SLDB is serialized and persisted as a collection of example rows, where each example row includes the following features:

- *timestamp*: a string that provides the date and time when the target value was read,
- *hourly*: the adjacent-hour delay vector that precedes the target,
- *daily*: the adjacent-day (same-hour) delay vector that precedes the target, and
- *weekly*: the adjacent-week (same-hour-and-day) delay vector that precedes the target.
- *target*: the power load value vector, assumed to be a function of its preceding delay vectors,

[1]I am indebted to División de Distribución Centro-Occidente, of Comisión Federal de Electricidad, for generously allowing me to use this relevant dataset in the experiments.

At its core, DMSLSTM consists of three stacked LSTM sub-networks. The first sub-network is trained to minimize the loss with respect to the targets on the collection of hourly features, and comprises a total of hn stacked LSTMs, with h_{h1} hidden units in the first LSTM, h_{h2} hidden units in the second LSTM, and so on, up to h_{hn} hidden units in the uppermost LSTM. The resulting stacked LSTM is identified as $LSTM_H$ and is built from the list of parameters $[h_{h1}, h_{h2}, \ldots, h_{hn}]$. Similarly, the stacked LSTM sub-networks $LSTM_D$ and $LSTM_W$ are trained to minimize the loss with respect to the targets on the collections of daily and weekly features, respectively. $LSTM_D$ is built from the list of parameters $[h_{d1}, h_{d2}, \ldots, h_{dn}]$, while $LSTM_W$ is built from the list of parameters $[h_{w1}, h_{w2}, \ldots, h_{wn}]$. It is worth noticing that $LSTM_H, LSTM_D$, and $LSTM_W$ are jointly trained.

The last hidden states from $LSTM_H, LSTM_D$, and $LSTM_W$ are concatenated, which results in a vector $\mathbf{LSTM}_{h+d+w} \in \mathbb{R}^{h_{hn}+h_{dn}+h_{wn}}$. The target timestamp is one-hot encoded in two parts: a 7-bit vector that identifies the day of the week (Sunday to Saturday), and a 24-bit vector that identifies the hour of the day (00:00 to 23:00). At this point, the boolean flag *use-timestamps* is used to define whether the target timestamp will be included as a feature or not: if *true* then $\mathbf{LSTM}_{h+d+w}$ is concatenated to the obtained one-hot timestamp vectors. The result is a vector $\mathbf{LSTM}_{h+d+w+ts} \in \mathbb{R}^{h_{hn}+h_{dn}+h_{wn}+7+24}$ that is passed to the next stage in the DMSLSTM network. Otherwise, if *use-timestamps* is *false*, $\mathbf{LSTM}_{h+d+w}$ is passed instead. The result of concatenation, either $\mathbf{LSTM}_{h+d+w+ts}$ or $\mathbf{LSTM}_{h+d+w}$, is transformed into a scalar prediction through a multi-layered dense neural network which comprises a total of on layers, with u_{o1} neurons in the first layer, u_{o2} neurons in the second layer, and so on, up to u_{on} neurons in its uppermost layer. This network is identified as $DENSE$, and is built from the list of parameters $[u_{o1}, u_{o2}, \ldots, u_{on}]$.

4.2.3 Model Parametrization

The model's parametrization is summarized in Table 4.2. DMSLSTM was trained using the Adam optimizer, as described in [Kingma and Ba, 2015], with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. For the learning rate, a custom schedule was used to help stabilize the learning process. This schedule was implemented as a succession of intervals measured in training epochs. In the first interval, the learning rate starts at 0, then increases linearly during a number of *warmup* steps, up to a maximum learning rate value. A base

learning rate, which linearly depends on the training batch size, is employed to ensure the maximum learning rate is fixed at 0.01. Subsequently, the learning rate is kept constant for the training epochs in the first interval, then divided by 10 before entering each next interval. This results in the step-like function observed in Figure 4.3.

Table 4.2: DMSLSTM parameters.

SLDB parameters	
$[m_h, m_d, m_w]$	[8, 8, 8]
# of training examples	16,168
# of evaluation examples	1,512
# of test examples	841
architecture parameters	
$[h_{h1}, h_{h2}]$	[64, 128]
dropout for *hourly* sub-network	0.2
$[h_{d1}, h_{d2}]$	[64, 128]
dropout for *daily* sub-network	0.2
$[h_{w1}, h_{w2}]$	[64, 128]
dropout for *weekly* sub-network	0.2
use-timestamps	true
$[u_{o1}, u_{o2}, u_{o3}]$	[512, 128, 24]
DENSE activation functions	[ReLU, ReLU, sigmoid]
forecast horizon	24
# of trainable parameters	689 K
training parameters	
base learning rate	0.0025
batch size	[128, 256, 512, 1,024]
# of training steps	1,580
warm-up steps	90
learning rate schedule (epochs)	[5, 40, 80, 95]
# of epochs	100

The MSE was used as the training and evaluation loss function and is

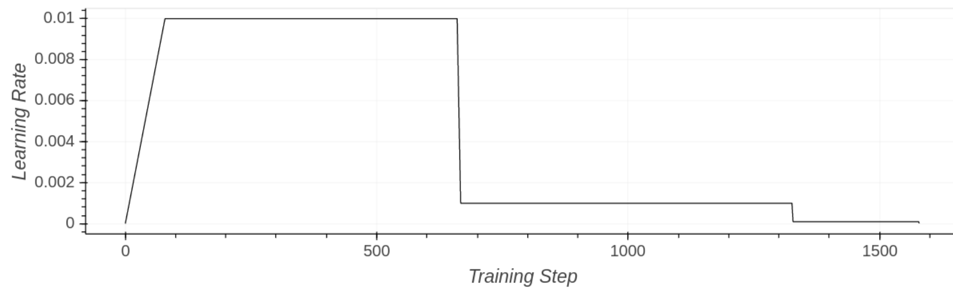


Figure 4.3: DMSLSTM Learning Rate Schedule. The learning rate starts at 0 and linearly increases during a given number of warmup steps up to a maximum, or base learning rate. For a total of three different training intervals, the learning rate is kept constant, then divided by 10 before entering the next interval.

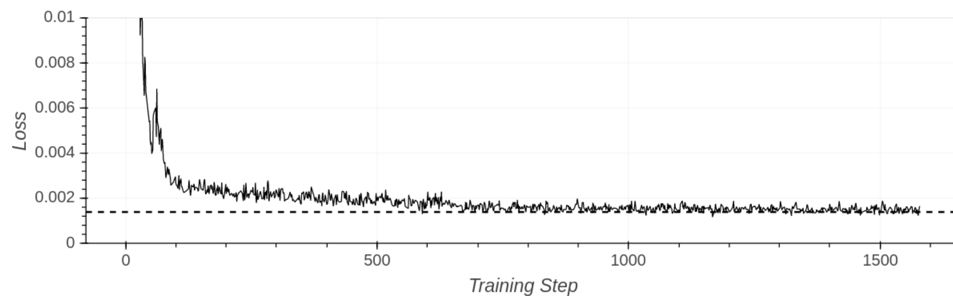


Figure 4.4: DMSLSTM Training Loss. The dotted line corresponds to the loss metric on evaluation, which is calculated only once, when training is completed.

defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(z_i - \hat{z}_i)^2 \tag{4.1}$$

The RMSE was used as an additional custom metric, via the `metric_fn` component of the TPUEstimator API, and is defined as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(z_i - \hat{z}_i)^2} \tag{4.2}$$

Concerning hyperparameter optimization, I recovered for this DMSLSTM multi-horizon forecasting model the collection of parameters obtained in previous experiments for the one-step-ahead version. This collection encompasses the selected values for $[m_h, m_d, m_w]$, $[h_{h1}, h_{h2}]$, $[h_{d1}, h_{d2}]$, $[h_{w1}, h_{w2}]$, and $[u_{o1}, u_{o2}, u_{o3}]$. Additionally, this multi-horizon version of DMSLSTM was specifically optimized for the training batch size.

The model was trained 10 different times, or executions, for each parameter set. Figure 4.4 shows the behavior of the main loss metric (MSE) over the training steps for one of the executions. The dotted line shows the value of the error metric for the model evaluation at the end of the training stage. In order to achieve high efficiency, training a model on TPUs is not compatible with frequent evaluation of the model, as it regularly happens when training on CPUs. The reason is starting the evaluation stage requires completely stopping the training loop, including shut down the TPU. As the time required to shut down and restart the TPU is usually higher than the time consumed by one training loop, it is a good practice to schedule evaluation, whenever possible, only at the check-pointing time, when the TPU is required to shut down in order to save the model training progress.

4.3 Results and Discussion

This Section presents the baseline model used to evaluate the DMSLSTM performance. On this basis, results achieved by DMSLSTM concerning forecasting accuracy, computation times, and predictive performance degradation are discussed.

4.3.1 Baseline

ARIMA (see Subsection 2.2.2) was selected as the baseline for evaluating DMSLSTM results. To produce a fair comparison, considering the massive computation capability that hardware accelerators provide to DMSLSTM, a cloud-based implementation of ARIMA was prepared and executed. BigQuery, the *data warehouse* service of GC, provides an optimized ARIMA implementation in the SaaS service model. BigQuery's modeling pipeline for ARIMA includes robust functionalities such as handling duplicated timestamps by mean calculation, handling missing values by local linear interpolation, detecting and cleaning spike and dip outliers, and using local calendars to detect and adjust the holiday effect on forecasting. Consequently, BigQuery's ARIMA provides an advanced, fast, and optimized forecasting model that is conveniently available on the SaaS mode and represents an appropriate baseline for the experiments.

Table 4.3: ARIMA parametrization on BigQuery ML.

number of days in the training set	56
number of readings in the training set	1,344
number of parallel models trained	42
forecast horizon	24
confidence level	90%
AUTO ARIMA max_order	5

Table 4.3 shows the parameter configuration used for fitting ARIMA models on BigQuery. For a given 24-hour-ahead forecast task, 7 weeks of previous active power readings were used as the training dataset. This translates as using 1,344 hourly readings for predicting the next 24 values. With this information, BigQuery produces up to 42 different ARIMA models by setting different combinations of integer values for the p, q, and d parameters. Parameter values are tested starting at $0, 1, \ldots$ up to the value defined by `AUTO ARIMA max_order`. All of the models are fit and evaluated in parallel, then the model with the lowest value for the Akaike Information Criterion (AIC) is selected as the winner and made available for prediction. At inference time, the selected ARIMA model is called with a forecast horizon and a confidence level as arguments. The forecast horizon defines the number of time steps in the future the model predicts. The confidence level is the percentage value

that expresses how confident we are the real values observed in the future will lie in the range defined by the prediction interval.

The above ARIMA parametrization was intended to keep the time required for fitting each model under 4 seconds. The reason for this constraint is that, unlike DMSLSTM models which are trained once and used to produce multiple inferences over long periods of time, ARIMA models are specifically trained to predict only one forecast horizon. This results in many ARIMA models competing against a single DMSLSTM model over the test time period, which enhances ARIMA predictive performance and results in a more balanced comparison. For instance, to produce daily predictions over a week, I trained seven BigQuery ARIMA models, each one predicting 24 hourly time steps ahead. Conversely, only one DMSLSTM model was trained to effectively produce 24-hour ahead predictions over forecasting periods that range from one day to three months.

4.3.2 Metrics

Three error metrics were used for DMSLSTM comparison with the baseline: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Symmetric Mean Absolute Percentage Error (SMAPE). MAE, delivers an error value that is conveniently expressed in the same units as the predicted variable. It is defined as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |\hat{z}_i - z_i| \tag{4.3}$$

RMSE is also expressed in the same units as the predicted variable, as defined in 4.2. Due to its quadratic nature, it tends to grow larger than MAE for sizable deviations with respect to the ground truth. This makes RMSE especially useful to penalize outliers. Finally, SMAPE provides an intuitive, percentage-based error metric that conveniently approximates Mean Absolute Percentage Error (MAPE), without having the disadvantage of exploding errors that emerge when values of the ground truth are close to zero. It is defined as follows:

$$SMAPE = \frac{100\%}{n}\sum_{i=1}^{n} \frac{|\hat{z}_i - z_i|}{(|z_i| + |\hat{z}_i|)/2} \tag{4.4}$$

It is worth noticing that, due to the addition of absolute values in the denominator, SMAPE tends to penalize under-forecasting slightly higher than over-forecasting.

4.3.3 Forecasting Accuracy

Figures 4.5 to 4.11 show the predictive performance on a one-week period, for daily predictions of active power produced by DMSLSTM and BigQuery ARIMA. It is worth mentioning that the DMSLSTM figures and metrics in the comparison come from the average of the ten training executions of the model, and not from the best one. Table 4.4 summarizes the error metrics obtained from inference on the first week of the test dataset. ARIMA metrics come from the only execution of this baseline while DMSLSTM metrics are average plus-minus the standard deviation of the executions set. The last column in the table shows the change in the error metrics that results when switching from the baseline to the implemented model, as a percentage of the metric achieved by the baseline.

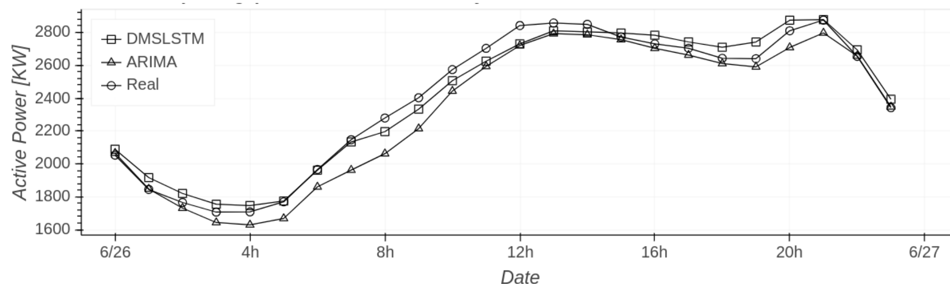


Figure 4.5: DMSLSTM vs. ARIMA vs. Real for Day 1. DMSLSTM shows a higher capacity than ARIMA on the predictions from the first half of the day.

DMSLSTM outperforms ARIMA on six of the seven days, with the exception of Day 4 (see Figure 4.8). For the days when DMSLSTM defeats the baseline, important reductions in the error metrics are achieved from switching ARIMA for DMSLSTM: 18% to 60% for MAE, 18% to 57% for RMSE, and 21% to 61% for SMAPE. Conversely, on the day that DMSLSTM does not outperform the baseline, the error metrics growths due to switching from ARIMA to DMSLSTM are significantly smaller: 2% for MAE, 3% for RMSE, and 1% for SMAPE.

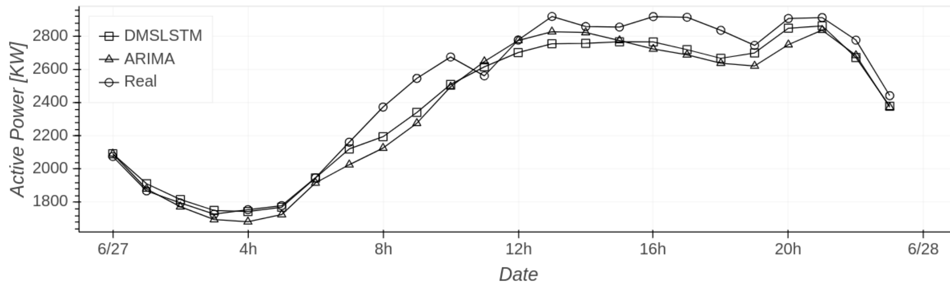


Figure 4.6: DMSLSTM vs. ARIMA vs. Real for Day 2. DMSLSTM shows a higher capacity than ARIMA from 07:00 to 09:00 hours and from 16:00 to 21:00 hours.

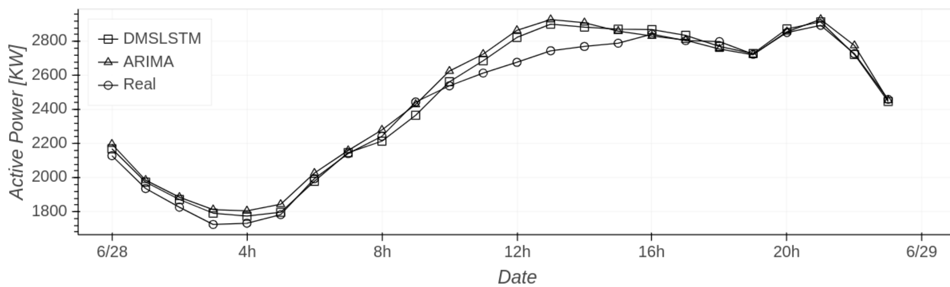


Figure 4.7: DMSLSTM vs. ARIMA vs. Real for Day 3. DMSLSTM shows a higher capacity than ARIMA from 10:00 to 15:00 hours.

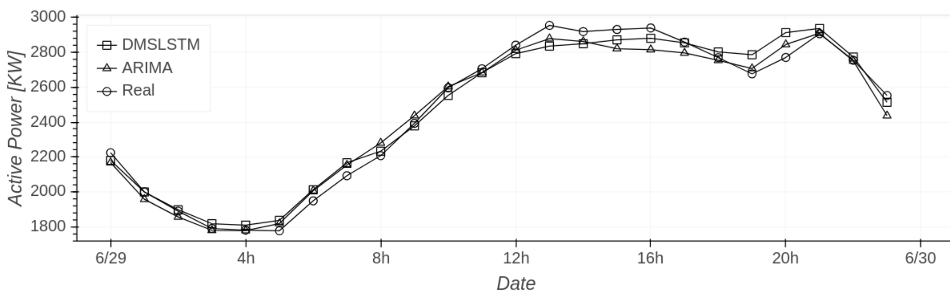


Figure 4.8: DMSLSTM vs. ARIMA vs. Real for Day 4. DMSLSTM shows a lower capacity than ARIMA from 19:00 to 20:00 hours.

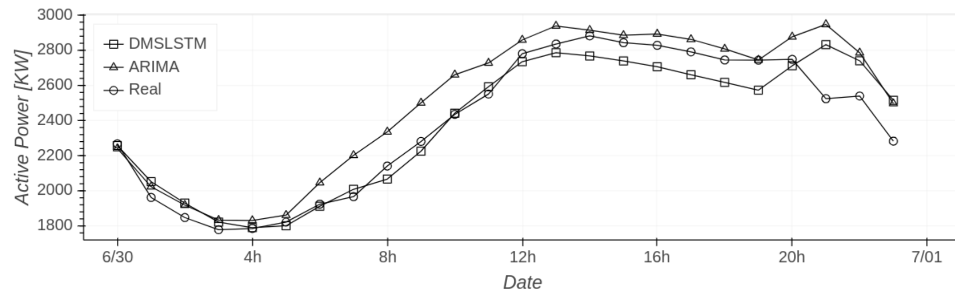


Figure 4.9: DMSLSTM vs. ARIMA vs. Real for Day 5. DMSLSTM shows a higher capacity than ARIMA from 06:00 to 13:00 hours.

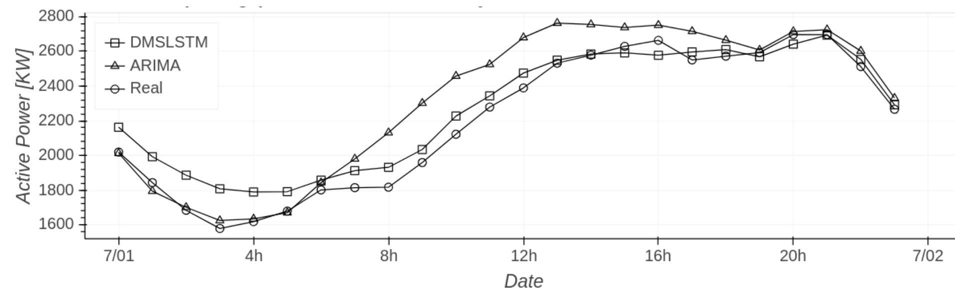


Figure 4.10: DMSLSTM vs. ARIMA vs. Real for Day 6. DMSLSTM shows a higher capacity than ARIMA from 07:00 to 18:00 hours.

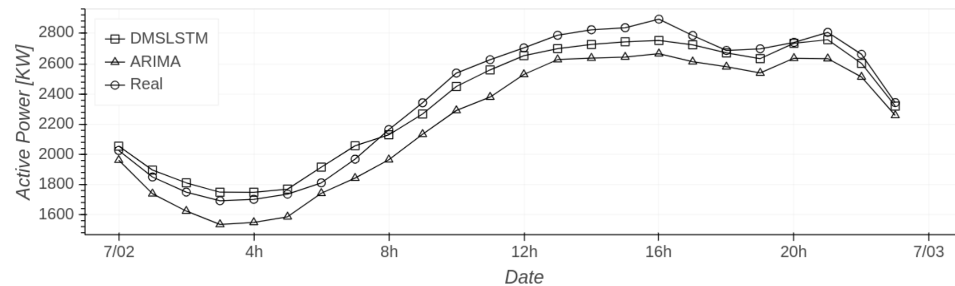


Figure 4.11: DMSLSTM vs. ARIMA vs. Real for Day 7. DMSLSTM shows a higher capacity than ARIMA for all of the predictions.

Table 4.4: DMSLSTM Vs. ARIMA Comparison.

MAE			
Interval	ARIMA	DMSLSTM	Δ(%)
Day 1	76.3232	**51.4102** ± 3.5676	**-32.64**
Day 2	104.6651	**85.5390** ± 3.6049	**-18.27**
Day 3	59.6213	**47.4907** ± 2.4059	**-20.35**
Day 4	**47.9564**	48.8155 ± 2.9000	1.79
Day 5	122.5390	**89.2606** ± 3.9335	**-27.16**
Day 6	123.2520	**85.3257** ± 4.4253	**-30.77**
Day 7	156.7935	**62.1411** ± 3.6187	**-60.37**
RMSE			
Interval	ARIMA	DMSLSTM	Δ(%)
Day 1	96.6190	**58.8745** ± 4.1009	**-39.07**
Day 2	130.3832	**107.4322** ± 4.4043	**-17.60**
Day 3	78.7560	**63.2627** ± 3.2333	**-19.67**
Day 4	**59.1122**	60.6888 ± 2.9308	2.67
Day 5	156.7504	**117.4109** ± 4.6368	**-25.10**
Day 6	165.9855	**104.4914** ± 6.2990	**-37.05**
Day 7	164.5955	**70.1199** ± 2.9061	**-57.40**
SMAPE			
Interval	ARIMA	DMSLSTM	Δ(%)
Day 1	3.3754	**2.1446** ± 0.1484	**-36.46**
Day 2	4.2182	**3.3229** ± 0.1459	**-21.22**
Day 3	2.4816	**1.9352** ± 0.1008	**-22.02**
Day 4	**1.9413**	1.9580 ± 0.1234	0.86
Day 5	5.0354	**3.5931** ± 0.1573	**-28.64**
Day 6	5.3700	**4.2671** ± 0.2278	**-20.54**
Day 7	6.9179	**2.6683** ± 0.1651	**-61.43**

As a common result, both DMSLSTM and ARIMA are able to capture the daily pattern in the active power time series. However, DMSLSTM shows a higher capacity that allows it to capture the short-term pattern better than the baseline model. This is clearly appreciated on the first half of Day 1 (see Figure 4.5), from 07:00 to 09:00 hours and from 16:00 to 21:00 hours of Day 2 (see Figure 4.6), from 10:00 to 15:00 hours of Day 3 (see Figure 4.7), from 06:00 to 13:00 hours of Day 5 (see Figure 4.9), from 07:00 to 18:00 hours of Day 6 (see Figure 4.10), and all of the predictions of Day 7 (see Figure 4.11). This higher capacity is a result of the multiple-resolution sequences DMSLSTM uses as input. It is less probable the augmented capacity comes as a result of the consideration of time covariates (hour-of-day and day-of-week indexes) as they are also employed by the baseline.

4.3.4 Computation Time

DMSLSTM was trained on a Cloud TPU node V3 with a peak computing performance of 420 TFlOPS, at bfloat16 (see Figure 3.3). During training, this massive capacity is mainly directed to the gradient computation and weight updating operations. Therefore, the number of trainable parameters in the forecasting model is crucial for assessing the time required in cloud infrastructure to complete each training execution. and its sequential-calculation nature due to the multiple LSTM layers in its architecture, the cloud-based TPU implementation is able to train DMSLSTM for 100 epochs of the reference dataset in just 24.6 seconds wall-time[2]. For comparison, I trained this configuration of DMSLSTM without the TPU accelerator using a Google Cloud Compute Engine n1-standard-4 virtual instance with 4 Intel Haswell vCPUs and 16 GB RAM, in 2,424 seconds for 100 epochs of the reference dataset, representing $\times 99$ the time required by the proposed infrastructure.

To properly compare the computation time of the model with the baseline, it is important to take into consideration the different training and inference approaches they apply. While ARIMA uses a specific training dataset to fit a prediction function for a contiguous forecasting interval, DMSLSTM is able to forecast time intervals that are not only next to the training dataset but also far in the future. In other words, the computation time required to

[2]Elapsed real time, or wall-clock time is the time taken to complete the training process. Training wall time is reported by TensorBoard and does not include the time spent on evaluation or checkpoint-related operations.

complete one training event for ARIMA is dedicated to forecasting a single prediction interval, while the computation time spent in training DMSLSTM once accounts for many prediction intervals. The number of forecasting intervals, therefore the total computation time, required by the baseline to match the usability of DMSLSTM depends on the forecasting scenario the comparison is built upon. Table 4.5 shows a total of seven common use cases for short-term forecasting in the electric power distribution field. For each use case, the number of training events required by the baseline model to achieve the forecasting goal is presented, along with the involved training computation time. According to the table, DMSLSTM outperforms the baseline in computation time whenever more than a single, 24-hour forecasting interval are required. For instance, training the baseline model a total of 7 times to produce one 24-hour prediction interval for each day of one week, requires 28 seconds, which is longer than the time required to train DMSLSTM once. Moreover, the trained model can be used to predict much more demanding common scenarios. The remainder of this Section is dedicated to discussing the predictive performance degradation of DMSLSTM in two widely-used, short-term forecasting scenarios: daily predictions over four weeks (28 inference executions) and hourly predictions over four weeks (672 inference executions).

Table 4.5: DMSLSTM Vs. ARIMA Computation Time Comparison.

Number of 24-hour predictions	DMSLSTM training time [s]	ARIMA training time [s]	Forecasting use case
1	24.5889 ± 0.2471	4	Single prediction
7	(fixed)	28	Daily predictions over 1 week
14	(fixed)	56	Daily predictions over 2 weeks
24	(fixed)	96	Hourly predictions over 1 day
28	(fixed)	112	Daily predictions over 4 weeks
168	(fixed)	672	Hourly predictions over 1 week
672	(fixed)	2,688	Hourly predictions over 4 weeks

4.3.5 Predictive Performance Degradation

Besides forecasting accuracy and training computation time, another relevant benchmark to assess the operation of an ML forecasting model is given

by the rate at which its predictive quality degrades over time. This predictive performance depends on the quality assured on all of the stages in the ML life cycle: data selection, data preprocessing, feature engineering, model parametrization, model training, and model evaluation. However, regardless of the quality assurance at its different stages, any ML model will suffer from performance degradation after being released. The reason is the model was trained on a static time frame of data while the underlying distribution expressed in that data has a dynamic nature. This is equivalent to saying that the model will render accurate predictions as long as the assumptions captured from past data are still relevant in the future. Therefore, continuous evaluation of the model's outcomes is required to ensure its reliability over different time frames.

ML model retraining is the first step to counteract performance degradation. It basically consists of training a new model using an up-to-date input dataset. Model retraining can be defined on a particular schedule (once a week, for instance) or triggered by a specific event, such as a performance metric value that falls below a given threshold or the availability of a new, significant-sized input dataset. Nevertheless, if the dynamic of the process that produces the data changes significantly, for instance, a reconfiguration in the power distribution network for the active power dataset, retraining a previous ML model on new data will not be sufficient. In such cases, a deeper reformulation of the model, in the form of a new feature engineering, a new architecture, or even a different algorithm set will be required to keep the model functional.

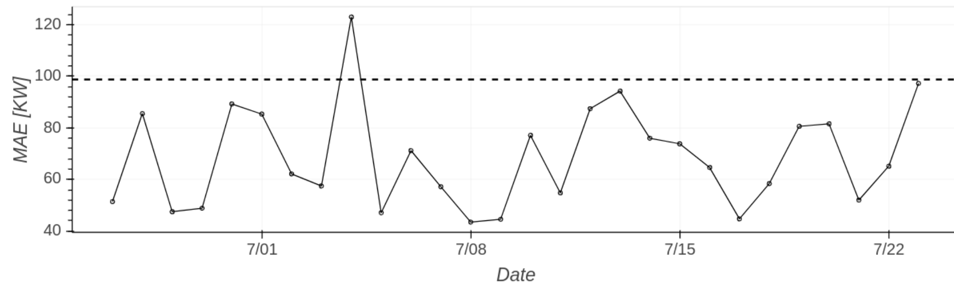
As a result, whether the forecasting model is just retrained or entirely redesigned, training it many times is a certain activity in the future. It could be thought that, with a training computation time under 25 seconds, the DMSLSTM model is apt to be retrained on a daily or even an hourly fixed schedule. However, in day-to-day electric power distribution operations, the real problem does not come from forecasting a single variable for a single metering device but from the Big Data situation that emerges from predicting dozens of variables for hundreds of meters. To estimate the massive volume of the data that is produced along with the benchmark dataset it has to be said that more than 30 electric variables, including three-phase-voltage, three-phase-current, reactive power, frequency, power quality indexes, etc., are continuously reported by a standard substation meter. In addition, it is usual for a power distribution operator to manage tens to hundreds of meters at the substation level, which accounts for several thousand single-variable

forecasting models to be constantly managed and monitored [Flores et al., 2022]. As a result, despite its extremely reduced training time, DMSLSTM is likely to be retrained following schedules in the range of once a week to once every four weeks, which encompasses the typical model training frequencies in mid- to large-sized electric power distribution operators.

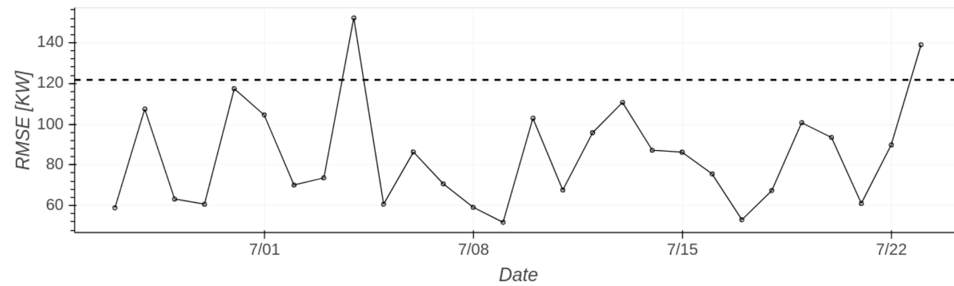
In this context, to evaluate the predictive performance degradation of the model I calculated two collections of 24-hour-ahead predictions distributed over a four-week interval: the first collection consists of 28 predictions, each one made at the start of a consecutive day in the four-week interval (rolling day); and the second collection consists of 672 predictions, each one made at a consecutive hour in the four-week interval (rolling hour). For each collection of predictions MAE, RMSE, and SMAPE metrics were calculated. For each error metric, the average of the corresponding baseline's first-week values (see Table 4.4) was calculated, then used as a threshold to classify the model's outcomes as better or worse with respect to the baseline.

Regarding the rolling-day collection of predictions, Figure 4.12a shows that 27 out of 28 (96%) predictions produce MAE values lower than the baseline's first-week average. Figure 4.12b shows that 26 out of 28 (93%) rolling-day predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 4.12c shows that 26 out of 28 (93%) rolling-day predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training DMSLSTM once, then using the model for rolling-day predictions over a four-week interval predominantly results in lower error metrics than forecasting with the baseline model. In addition, the baseline model would require a training computation time of close to 112 seconds to address the task, which is $\times 4.6$ the time spent in DMSLSTM's training.

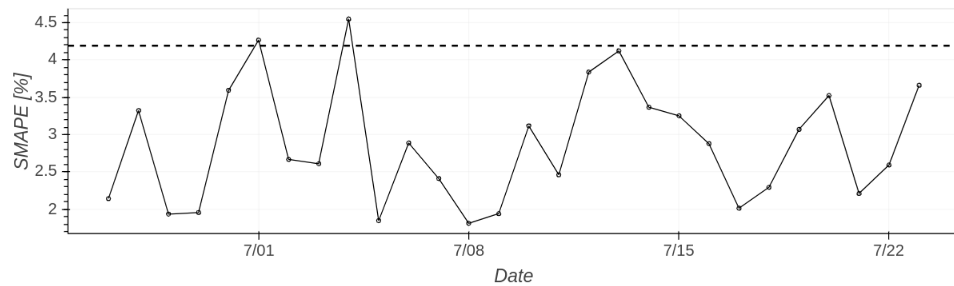
Regarding the rolling-hour collection of predictions, Figure 4.13a shows that 549 out of 672 (82%) predictions produce MAE values lower than the baseline's first-week average. Figure 4.13b shows that 545 out of 672 (81%) rolling-hour predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 4.13c shows that 565 out of 672 (84%) rolling-hour predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training DMSLSTM once, then using the model for rolling-hour predictions over a four-week interval predominantly results in lower error metrics than forecasting with the baseline model. In addition, the baseline model would require a training computation time of close to 2,688 seconds to address the task, which is $\times 109$ the time



(a) MAE.

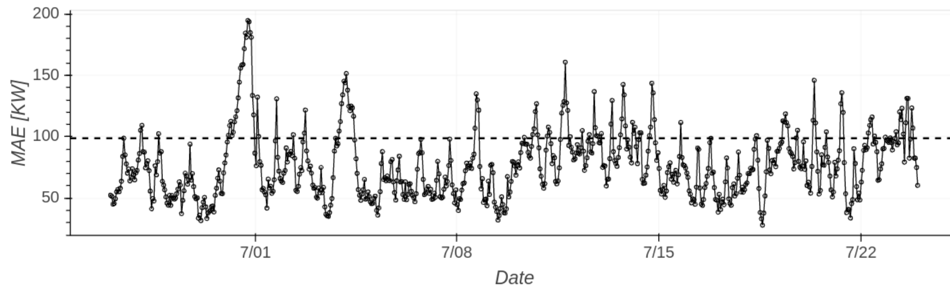


(b) RMSE.

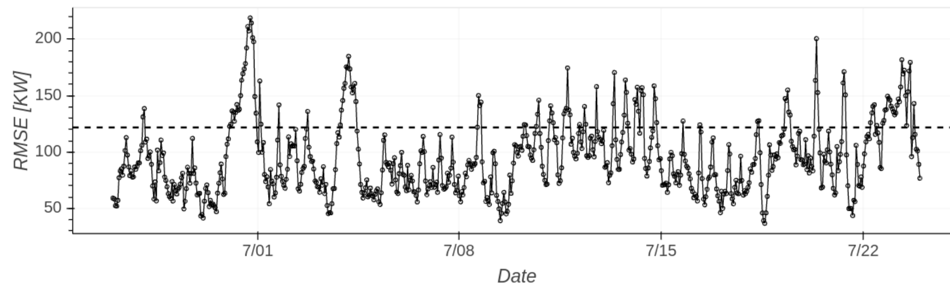


(c) SMAPE.

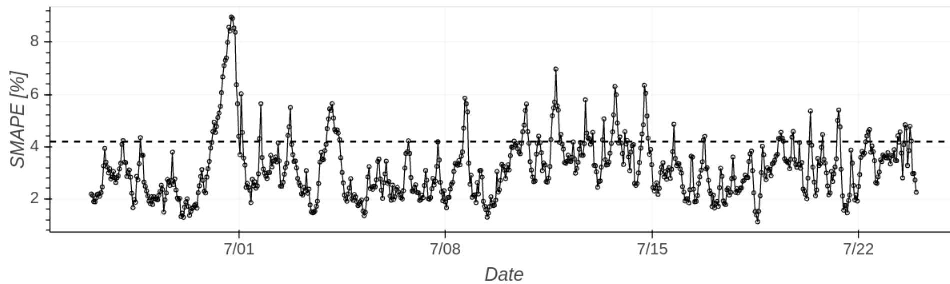
Figure 4.12: DMSLSTM metrics for 28 consecutive (rolling-day) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.



(a) MAE.



(b) RMSE.



(c) SMAPE.

Figure 4.13: DMSLSTM metrics for 672 consecutive (rolling-hour) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.

spent in training DMSLSTM.

Table 4.6: DMSLSTM Summarized Results.

Forecasting accuracy	
Daily predictions over a week MAE average	67.14
Weekly MAE increment w.r.t baseline	-32.0%
Daily predictions over a week RMSE average	74.51
Weekly RMSE increment w.r.t baseline	-31.7%
Daily predictions over a week SMAPE average	2.84
Weekly SMAPE increment w.r.t baseline	-32.2%
Computation time	
# of trainable parameters	629 K
Training time with TPU acceleration (100 epochs)	24.59 s
Training time factor without TPU acceleration	$\times$99
Training time factor for the baseline, 28 rolling-day predictions	$\times$4.6
Training time factor for the baseline, 672 rolling-hour predictions	$\times$109
Predictive performance degradation	
Four-week rolling-day predictions with MAE below the baseline	96%
Four-week rolling-day predictions with RMSE below the baseline	93%
Four-week rolling-day predictions with SMAPE below the baseline	93%
Four-week rolling-hour predictions with MAE below the baseline	82%
Four-week rolling-hour predictions with RMSE below the baseline	81%
Four-week rolling-hour predictions with SMAPE below the baseline	84%

Summary

This Chapter presents in detail DMSLSTM, a DL architecture based on stacked LSTM layers, parallel multi-sequence input, and date encoding that produces multi-horizon predictions for a univariate forecasting problem. This architecture was tested on an extensive dataset from the active power readings of an urban substation metering device. The baseline model used for evaluating DMSLSTM performance was a robust, cloud-based ARIMA implementation from the GC BigQuery data warehousing service.

Table 4.6 summarizes the experimental study. Some important facts that can be observed from that summary are:

- DMSLSTM overperforms the baseline in forecasting accuracy and training computation time.
- DMSLSTM exceeds the baseline in two typical evaluations for predictive performance degradation.
- TPU-based acceleration allows to train DMSLSTM $\times 99$ faster than CPUs.
- DMSLSTM's predictive performance slightly degrades from one-week to one-month intervals after training, although it still overperforms the baseline's first-week average.

The next Chapter presents the second neural network architecture selected for TPU-based implementation: a Seq2Seq with attention TSF model identified as EDSLSTM.

Chapter 5

Sequence-to-Sequence with Attention Architecture

This Chapter presents in detail the Encoder-Decoder with attention Stacked Long Short-Term Memory (EDSLSTM) architecture that I designed and implemented for local TSF. Section 5.1 presents related research work. Section 5.2 presents in detail the architecture and the parametrization of the model. Section 5.3 presents the results achieved by EDSLSTM concerning forecasting accuracy, computation times, and predictive performance degradation.

5.1 Related Work

The EDSLSTM forecasting model is based on the Sequence to Sequence (Seq2Seq) architecture. It includes an encoder module, a decoder module, and an attention mechanism (see Subsection 2.3.2). The Seq2Seq architecture was originally developed for NLP applications [Sutskever et al., 2014], [Bahdanau et al., 2014], [Luong et al., 2015], and has been successfully adapted to the TSF problem.

[Cinar et al., 2017a] extend the original *content attention*, introduced by Seq2Seq models for NLP problems, with an enhanced *position-based attention* that incorporates the relative positions in the input and output sequences, making the Seq2Seq architecture more suitable for TSF. [Fan et al., 2019] propose an end-to-end DL framework for multi-horizon TSF with novel structures to better capture temporal patterns in the future. It includes a decoder module based on bi-directional LSTM layers, and a combined *temporal con-*

text feature that is produced by calculating the separated contribution of different *modalities* (different periods of history) in the hidden state of each prediction timestep.

For the case of multivariate TSF [Shih et al., 2019] propose a method that allows capturing time-invariant patterns across multiple time steps. As a result, this *temporal pattern attention* model learns not only the relevant input timesteps (its contribution in predicting the output sequence) but also the relevant input time series (or variables). To address this task, the authors introduce 1-D CNN filters that extract temporal pattern information from each individual variable. In the same direction, [Du et al., 2020] propose an encoder-decoder architecture with temporal attention for multivariate TSF, which comprises a bi-directional LSTM encoder network, a LSTM decoder network, and a temporal attention mechanism that can select relevant encoder hidden states across all timesteps. The authors claim this architecture is able to learn hidden long-term dependent features and non-linear correlation features from the raw multivariate time series data.

[Kharlova et al., 2020] propose and evaluate a Seq2Seq model with attention to perform hourly, day-ahead forecasts of residential photovoltaic power production. The authors show this model can leverage high-resolution historical data by learning to forecast a time series of binned probability distributions instead of expected values. [Sehovac and Grolinger, 2020] implement a Seq2Seq with attention architecture applied to power load forecasting and evaluate its prediction accuracy on several variants of the base RNN and the attention mechanism. The authors found Badhanau attention [Bahdanau et al., 2014] overperforms three variants of Luong attention [Luong et al., 2015] for this particular problem. [Xu et al., 2021b] propose a novel architecture for TSF of water quality: they implement a factorization machine for feature extraction of water quality time series data to obtain potential relationships between water quality features, and extend the traditional Seq2Seq, GRU-based network with a dual attention mechanism that solves the problem of decoding distraction and long-range information loss. In a similar direction, [Zhang et al., 2021] propose a hybrid architecture for short-term forecasting of multi-energy loads in an integrated energy microgrid. The authors use a CNN network to extract features of the input time series and then a Seq2Seq, LSTM-based with attention network to improve forecasting accuracy on multi-load (electricity, cooling, and heating) predictions.

5.2 Experimental Study

This section presents the experimental study with the EDSLSTM architecture. It discusses the electric power dataset used as the basis of the forecasting process. Later, it expands on the specifics of the forecasting model architecture. Finally, it reports the model's parametrization for the use case.

5.2.1 Dataset

To benchmark the EDSLSTM forecasting model, I used the same active power, hourly resolution, urban substation meter dataset employed for DMSLSTM, as described in Subsection 4.2.1. As the time window specifications for SLDB construction differ between DMSLSTM and EDSLSTM, the two models' resulting time intervals for training, evaluation, and test datasets are not coincident. The main difference is in the sizes of the test datasets, with 841 example rows available for DMSLSTM and 2,121 example rows available for EDSLSTM.

5.2.2 Architecture Description

Implementing the EDSLSTM architecture comprises, like in the DMSLSTM case, three stages: Supervised-Learning DataBase (SLDB) generation, model architecture definition, and model training. In the first stage, features, and targets (or *labels*) are extracted from historical time series of electric power load and stored in the corresponding SLDB. In the second stage, a specific EDSLSTM architecture is defined on the basis of the input SLDB and the model's composite structure. Finally, in the third stage, the EDSLSTM is trained on the SLDB and the resulting model is evaluated and deployed for prediction.

Figure 5.1 illustrates the EDSLSTM architecture. The SLDB that provides an input for the EDSLSTM network is generated from the historical record, at 1-hour resolution, of the active power measured at the substation level and the embedding dimension m. The result is a collection of hourly-based time windows of the form $\{z_1, z_2, \dots, z_m\}$ (see Subsection 2.1.1). Once generated, the SLDB is serialized and persisted as a collection of example rows, where each example row includes the following features:

- *hourly*: the adjacent-hour, m-sized delay vector that precedes the forecasting horizon label,

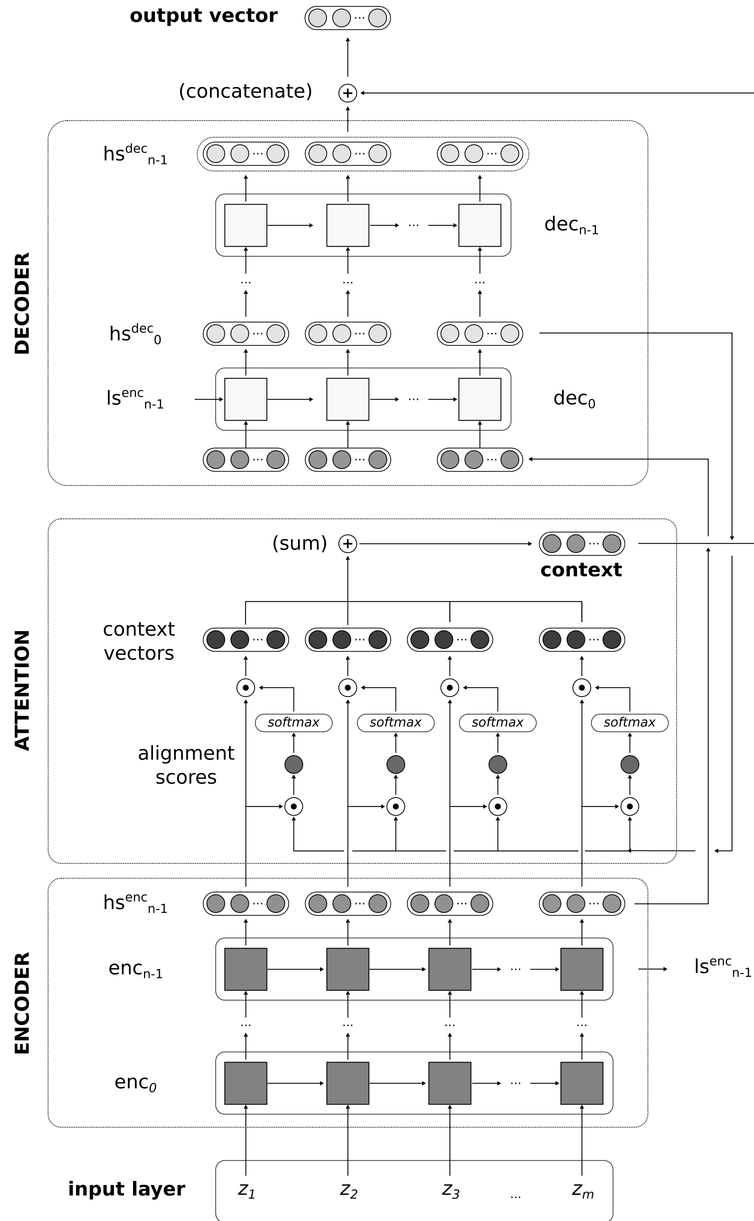


Figure 5.1: Encoder-Decoder with attention Stacked LSTM Network Architecture.

- *target*: the active power value vector along the forecasting horizon, assumed to be a function of its preceding delay vector.

The EDSLSTM network consists of three modules or sub-networks: the encoder, the decoder, and the attention mechanism. The encoder module is built with n stacked LSTM layers. All the encoder layers $enc_i, i \in [0, 1, \ldots, n-1]$ are built with the same number h of hidden units. The encoder layer $enc_i, i \in [0, 1, \ldots, n-2]$ delivers its hidden stack hs_i^{enc}, that is the collection of hidden states for all of its timesteps, as input for the upstream encoder layer enc_{i+1}. The uppermost encoder layer enc_{n-1}, passes its hidden stack hs_{n-1}^{enc} as input for the attention module. It also passes the hidden state of its last timestep as input for the decoder module, and both the hidden state and the cell state of its last timestep, represented in the illustration as ls_{n-1}^{enc}, as the initial state for the lowermost decoder layer dec_0.

Same as the encoder, the decoder module is built with n stacked LSTM layers. All the decoder layers $dec_i, i \in [0, 1, \ldots, n-1]$ are built with the same number h of hidden units. The decoder layer $dec_i, i \in [0, 1, \ldots, n-2]$ delivers its hidden stack hs_i^{dec}, as input for the upstream decoder layer dec_{i+1}. The hidden state in the last timestep of enc_{n-1} is repeated along the time axis a number of times equal to the number of target timesteps, then passed as input to the lowermost decoder layer dec_0. This decoder layer also passes its hidden stack hs_0^{dec} as input for the attention mechanism. The uppermost decoder layer dec_{n-1} delivers its hidden stack hs_{n-1}^{dec} as output of the decoder module.

The attention mechanism learns the separated contribution that each element of the encoded sequence has in calculating each element of the decoded sequence. In other words, it allows the computation of each element in the lowermost decoder layer's state by attending to all of the elements (the hidden stack) in the uppermost encoder layer's state, and not only to the last hidden state, as it happens with traditional Seq2Seq architectures. To address this task, *alignment scores* are calculated as

$$alignment = hs_0^{dec} \cdot hs_{n-1}^{enc} \tag{5.1}$$

The *softmax* activation function is applied to the alignment scores to learn the *attention weights* matrix as a probability distribution. Then, *context vectors* for each timestep in the network state are calculated as the dot product of the hidden stack in the uppermost encoder layer's state with the alignment

scores. Finally, the global *context* is calculated by adding up the context vectors as:

$$context = \sum_{j=0}^{n-1} hs^{enc}_{n-1\,j} \cdot alignment_j \tag{5.2}$$

where j denotes the index in the timestep axis of the hidden stack.

When using a standard Seq2Seq architecture, the hidden state at the decoder's output is projected to the target dimension using a dense neural network layer that is time-distributed along the timesteps in the forecast horizon. In a Seq2Seq with attention architecture, like EDSLSTM, the hidden state at the decoder's output is concatenated with the context to generate an attention-based decoder output, which is finally projected to the forecast horizon. As a result, the calculation of each timestep in the prediction sequence is affected by the attention weights on all of the elements in the input sequence. It is worth mentioning that, in order to stabilize and speed up the training process, batch normalization [Ioffe and Szegedy, 2015] was applied to standardize the last hidden state and the last cell state of the uppermost layer in the encoder, as well as the global context.

5.2.3 Model Parametrization

The EDSLSTM model parametrization is summarized in Table 5.1. The model was trained using the Adam optimizer, as described in [Kingma and Ba, 2015], with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. For the learning rate, a custom schedule was used to help stabilize the learning process. This schedule was implemented as a succession of intervals measured in training epochs. In the first interval, the learning rate starts at 0, then increases linearly during a number of *warmup* steps, up to the base learning rate value. The base learning rate linearly depends on the training batch size, and its value is adjusted to be equal to 0.01 at a batch size of 256, 0.005 at a batch size of 512, and so on. Subsequently, the learning rate is kept constant for the training epochs in the first interval, then divided by 10 before entering each next interval. This results in the step-like behavior observed in Figure 5.2. The MSE was used as the training and evaluation loss function. The RMSE was used as an additional custom metric, via the `metric_fn` component of the TPUEstimator API.

Concerning hyperparameter optimization, grid-search processes were conducted for the number of layers in the encoder and the number of layers in

Table 5.1: EDSLSTM parameters.

SLDB parameters	
context timesteps	64
target timesteps	24
# of training examples	17,448
# of evaluation examples	2,792
# of test examples	2,121
architecture parameters	
number of layers	[2, 3]
number of hidden units	256
activation function	ELU
dropout	0.2
recurrent dropout	0.2
momentum (hidden state)	0.6
momentum (cell state)	0.6
context momentum	0.6
DENSE structure	[1]
DENSE activation function	[sigmoid]
forecast horizon	24
# of trainable parameters	2.3 M
training parameters	
base learning rate	[0.001, 0.002, 0.003, 0.004, 0.005]
learning rate schedule steps	[5, 15, 18, 20]
batch size	256
# of training steps	1,340
warm-up steps	280
# of epochs	20

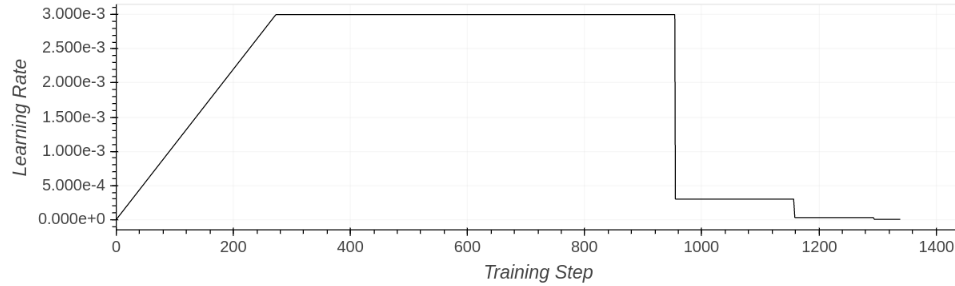


Figure 5.2: EDSLSTM Learning Rate Schedule. The learning rate starts at 0 and linearly increases during a given number of warmup steps up to a maximum, or base learning rate. For a total of three different training intervals, the learning rate is kept constant, then divided by 10 before entering the next interval.

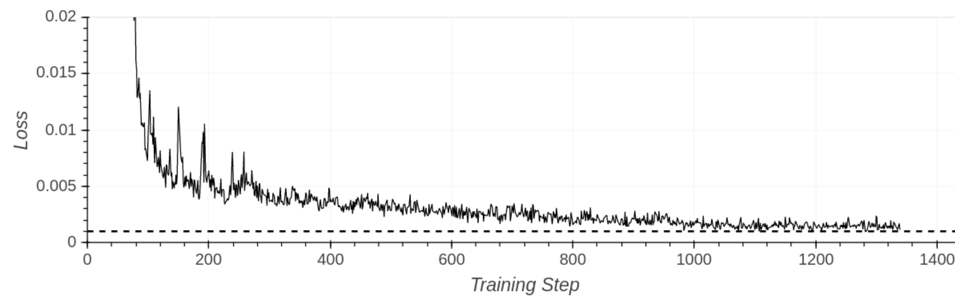


Figure 5.3: EDSLSTM Training Loss. The dotted line corresponds to the loss metric on evaluation, which is calculated only once, when training is completed.

the decoder (architecture parameters) and for the base learning rate (training parameter) as indicated in Table 5.1. The remaining hyperparameters were selected according to experiments reported in the literature for similar neural network architectures applied to electric power load forecasting, particularly from [Sehovac and Grolinger, 2020]. The model was trained 10 different times, or executions, for each parameter set. Figure 5.3 shows the behavior of the main loss metric (MSE) over the training steps for one of the executions of the selected configuration. The dotted line shows the value of the error metric for the model evaluation at the end of the training stage. In order to achieve high efficiency, training a model on TPUs is not compatible with frequent evaluation, as it regularly happens with CPU-based training. The reason is starting the evaluation stage requires completely stopping the training loop, including shut down the TPU. As the time required to shut down and restart the TPU is usually higher than the time consumed by one training loop, it is a good practice to schedule evaluation, whenever possible, only at the check-pointing time, when the TPU is required to shut down in order to save the model training progress.

5.3 Results and Discussion

This Section presents the baseline used for evaluating EDSLSTM performance. It also discusses the results of the model concerning forecasting accuracy, computation times, and predictive performance degradation.

5.3.1 Baseline

As in the DMSLSTM case, BigQuery's ARIMA implementation was used to provide a baseline for evaluating EDSLSTM results. The BigQuery's ARIMA parametrization is detailed in Subsection 4.3.1. As previously stated, this parametrization was intended to keep the time required for fitting each model under 4 seconds.

5.3.2 Metrics

Three error metrics were used for EDSLSTM comparison with the baseline: MAE (4.1), RMSE (4.2), and SMAPE (4.4).

5.3.3 Forecasting Accuracy

Figures 5.4 to 5.10 show the predictive performance on a one-week period, for daily predictions of active power produced by the model and BigQuery's ARIMA. It is worth mentioning that the EDSLSTM figures and metrics in the comparison come from the average of the ten training executions of the model, and not from the best one. Table 5.2 summarizes the error metrics obtained from inference on the first week of the test dataset. ARIMA metrics come from the only execution of this baseline while EDSLSTM metrics are average plus-minus the standard deviation of the executions set. The last column in the table shows the change in the error metrics that results when switching from the baseline to EDSLSTM, as a percentage of the metric achieved by the baseline.

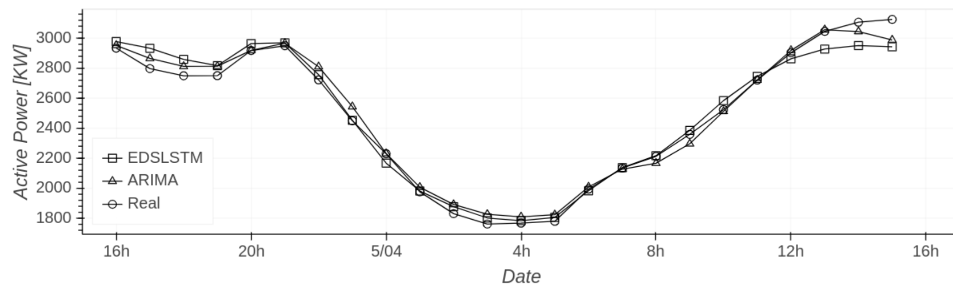


Figure 5.4: EDSLSTM vs. ARIMA vs. Real for Day 1. ARIMA shows a higher capacity than EDSLSTM from 17:00 to 19:00 hours and from 12:00 to 15:00 hours.

EDSLSTM outperforms ARIMA on five of the seven days, with the exception of Day 1 (see Figure 5.4) and Day 5 (see Figure 5.8). For the days when EDSLSTM defeats the baseline, important reductions in the error metrics are achieved from switching ARIMA for EDSLSTM: 19% to 66% for MAE, 24% to 60% for RMSE, and 22% to 68% for SMAPE. Conversely, on the days that EDSLSTM does not outperform the baseline, the error metrics growths due to switching from ARIMA to EDSLSTM are significantly smaller: 14% to 28% for MAE, 2% to 35% for RMSE, and 10% to 17% for SMAPE.

As a common result, both EDSLSTM and ARIMA are able to capture the daily pattern in the active power time series. However, EDSLSTM shows a higher capacity that allows it to capture the short-term pattern better than the baseline model. This is clearly appreciated from 16:00 to 20:00 hours and

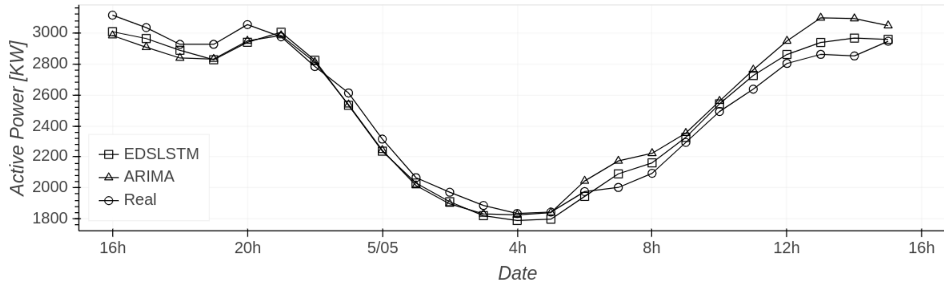


Figure 5.5: EDSLSTM vs. ARIMA vs. Real for Day 2. EDSLSTM outperforms ARIMA from 16:00 to 20:00 hours and from 11:00 to 15:00 hours.

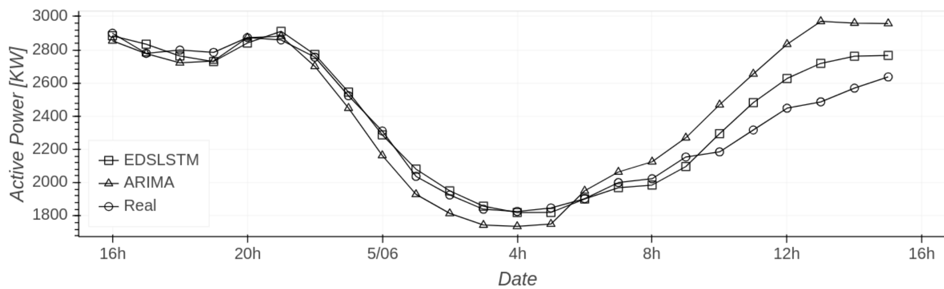


Figure 5.6: EDSLSTM vs. ARIMA vs. Real for Day 3. EDSLSTM outperforms ARIMA from 22:00 to 15:00 hours.

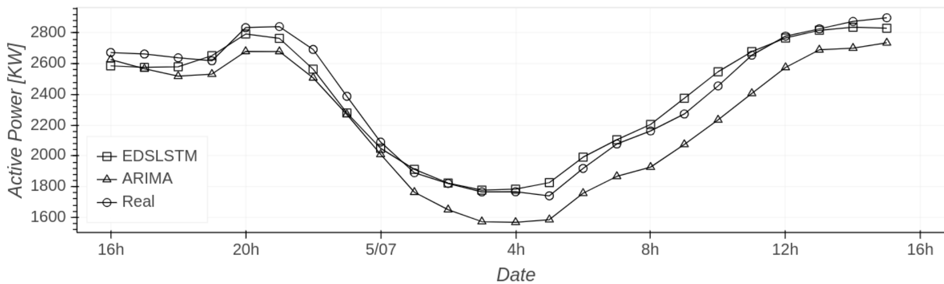


Figure 5.7: EDSLSTM vs. ARIMA vs. Real for Day 1. EDSLSTM outperforms ARIMA from 18:00 to 15:00 hours.

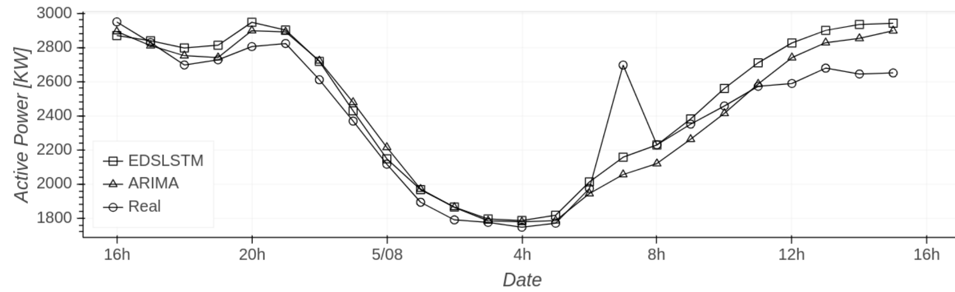


Figure 5.8: EDSLSTM vs. ARIMA vs. Real for Day 5. ARIMA shows a higher capacity than EDSLSTM in the last hours of the day, after its outlier-removal functionality was applied at the 07:00 hours reading.

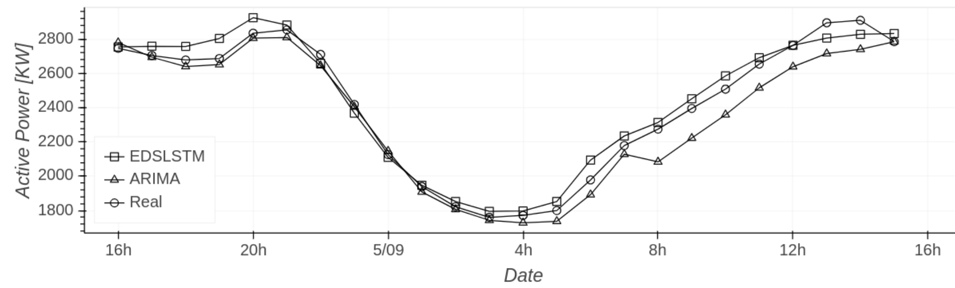


Figure 5.9: EDSLSTM vs. ARIMA vs. Real for Day 6. EDSLSTM outperforms ARIMA from 04:00 to 15:00 hours.

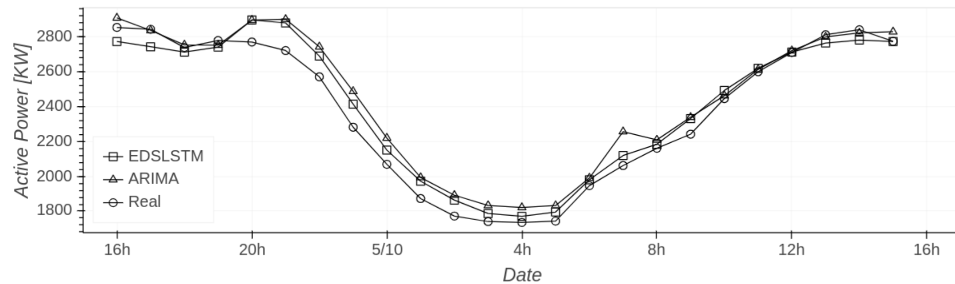


Figure 5.10: EDSLSTM vs. ARIMA vs. Real for Day 7. EDSLSTM outperforms ARIMA from 21:00 to 09:00 hours.

Table 5.2: EDSLSTM Vs. ARIMA Comparison.

MAE			
Interval	ARIMA	EDSLSTM	Δ(%)
Day 1	**42.7504**	54.5234 ± 2.3951	27.54
Day 2	94.4938	**63.8401** ± 4.0132	**-32.44**
Day 3	146.2935	**65.6171** ± 4.4611	**-55.15**
Day 4	160.2514	**55.0586** ± 5.6558	**-65.64**
Day 5	**104.5032**	118.6643 ± 4.6559	13.55
Day 6	71.7451	**55.7155** ± 16.8942	**-22.34**
Day 7	81.3132	**66.2347** ± 4.3673	**-18.54**
RMSE			
Interval	ARIMA	EDSLSTM	Δ(%)
Day 1	**54.8031**	73.7896 ± 2.5624	34.64
Day 2	112.5502	**70.4892** ± 3.7769	**-37.37**
Day 3	199.6776	**92.3386** ± 11.6959	**-53.76**
Day 4	167.9489	**66.9185** ± 6.9864	**-60.16**
Day 5	**164.4347**	168.3410 ± 4.4768	2.38
Day 6	94.2247	**67.6061** ± 18.6421	**-28.25**
Day 7	102.9821	**78.0422** ± 5.2829	**-24.22**
SMAPE			
Interval	ARIMA	EDSLSTM	Δ(%)
Day 1	**1.7587**	2.0528 ± 0.0971	16.72
Day 2	3.6541	**2.5696** ± 0.1727	**-29.68**
Day 3	6.0585	**2.6614** ± 0.1491	**-56.07**
Day 4	7.2724	**2.3029** ± 0.2539	**-68.33**
Day 5	**4.2612**	4.6691 ± 0.1741	9.57
Day 6	2.9962	**2.2712** ± 0.7302	**-24.20**
Day 7	3.6075	**2.8146** ± 0.1797	**-21.98**

from 11:00 to 15:00 hours of Day 2 (see Figure 5.5), from 22:00 to 15:00 hours of Day 3 (see Figure 5.6), from 18:00 to 15:00 hours of Day 4 (see Figure 5.7), from 04:00 to 15:00 hours of Day 6 (see Figure 5.9), and from 21:00 to 09:00 hours of Day 7 (see Figure 5.10).

5.3.4 Computation Time

Computation time for a DL model is mainly defined by the number of trainable parameters in its architecture. EDSLSTM is a particularly heavy model: with close to 2.3 million trainable parameters and six sequential-calculation LSTM layers, it represents a challenge for a typical training infrastructure. Nevertheless, the cloud-based TPU implementation is able to train EDSLSTM for 20 epochs of the reference dataset in just 28.6 seconds of wall time. For comparison, I trained this configuration of EDSLSTM without the TPU accelerator using a Google Cloud Compute Engine n1-standard-4 virtual instance with 4 Intel Haswell vCPUs and 16 GB RAM, in 528 seconds for one epoch of the reference dataset, representing $\times 369$ the time required by the proposed infrastructure. It is worth to mention it was not possible to train EDSLSTM for a longer period without using TPU acceleration, as the training process crashed several times due to excessive CPU utilization.

Table 5.3: EDSLSTM Vs. ARIMA Computation Time Comparison.

Number of 24-hour predictions	EDSLSTM	ARIMA	Use case
1	28.6380 ± 0.0341	4	Single prediction
7	(fixed)	28	Daily predictions over a week
14	(fixed)	56	Daily predictions over two weeks
24	(fixed)	96	Hourly predictions over a day
28	(fixed)	112	Daily predictions over four weeks
168	(fixed)	672	Hourly predictions over a week
672	(fixed)	2,688	Hourly predictions over four weeks

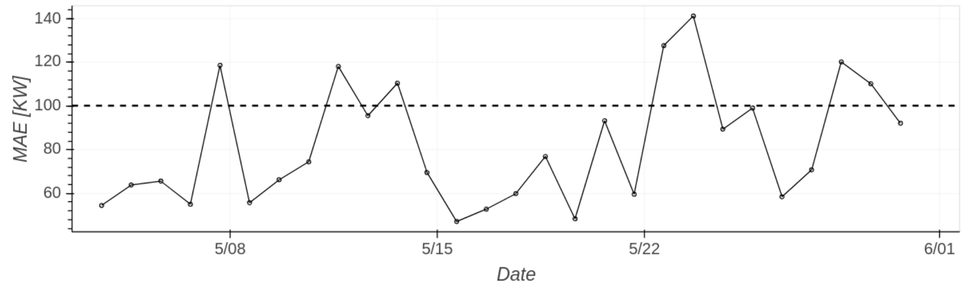
To properly compare the computation time of EDSLSTM with the baseline, it is important to take into consideration the different training and inference approaches they apply. I addressed the basics of these approaches and their repercussions on the benchmark evaluation on Subsection 4.3.4. Table 5.3 shows a total of seven common use cases for short-term forecas-

ting in the electric power distribution field. For each use case, the number of training events required by the baseline model to achieve the forecasting goal is presented, along with the involved training computation time. According to the table, EDSLSTM outperforms the baseline in computation time whenever more than seven 24-hour forecasting intervals are required. For instance, training the baseline model a total of 14 times to produce one 24-hour prediction interval for each day of two weeks, requires 56 seconds, which is more than the time required to train EDSLSTM once. Moreover, the trained model can be used to predict much more demanding common scenarios. The remainder of this Section is dedicated to discussing the predictive performance of EDSLSTM in two widely-used, short-term forecasting scenarios: daily predictions over four weeks (28 inference executions) and hourly predictions over four weeks (672 inference executions).

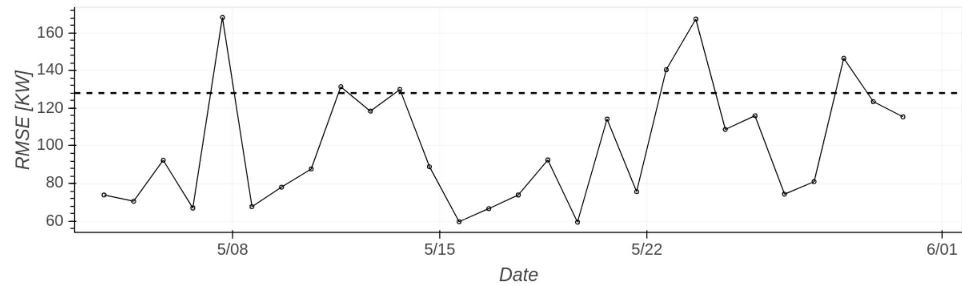
5.3.5 Predictive Performance Degradation

As in the DMSLSTM case, I evaluated the predictive performance degradation of EDSLSTM using two collections of 24-hour-ahead predictions distributed over a four-week interval: the first collection consists of 28 predictions, each one made at the start of a consecutive day in the four-week interval (rolling day); and the second collection consists of 672 predictions, each one made at a consecutive hour in the four-week interval (rolling hour). For each collection of predictions MAE, RMSE, and SMAPE metrics were calculated. For each error metric, the average of the corresponding baseline's first-week values (see Table 5.2) was calculated, then used as a threshold to classify EDSLSTM's outcomes as better or worse with respect to the baseline.

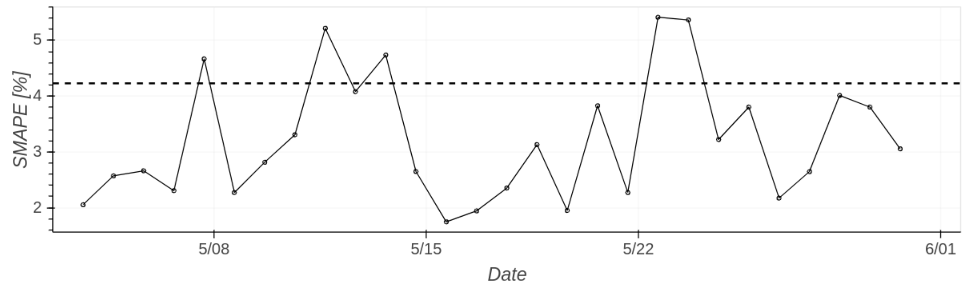
Regarding the rolling-day collection of predictions, Figure 5.11a shows that 21 out of 28 (75%) predictions produce MAE values lower than the baseline's first-week average. Figure 5.11b shows that 22 out of 28 (79%) rolling-day predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 5.11c shows that 23 out of 28 (82%) rolling-day predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training EDSLSTM once, then using the model for rolling-day predictions over a four-week interval mostly results in lower error metrics than forecasting with the baseline model. In addition, the baseline model would require a training computation time of close to 112 seconds to address the task, which is $\times 3.9$ the time spent in training EDSLSTM.



(a) MAE.

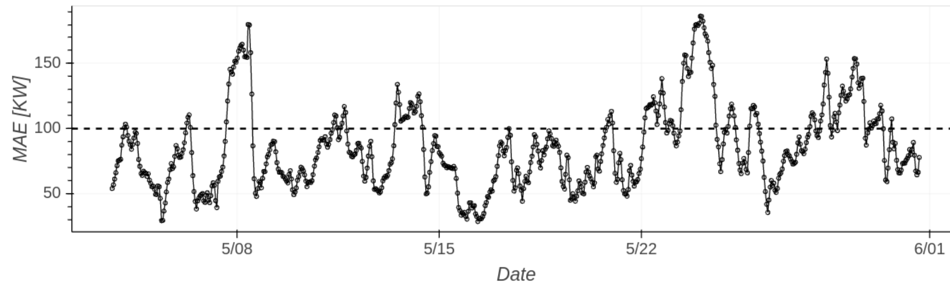


(b) RMSE.

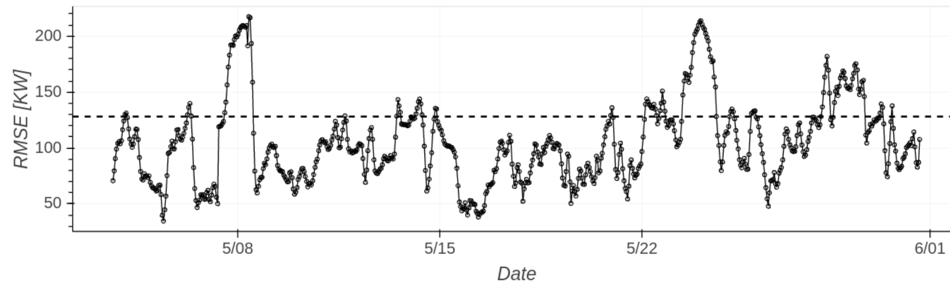


(c) SMAPE.

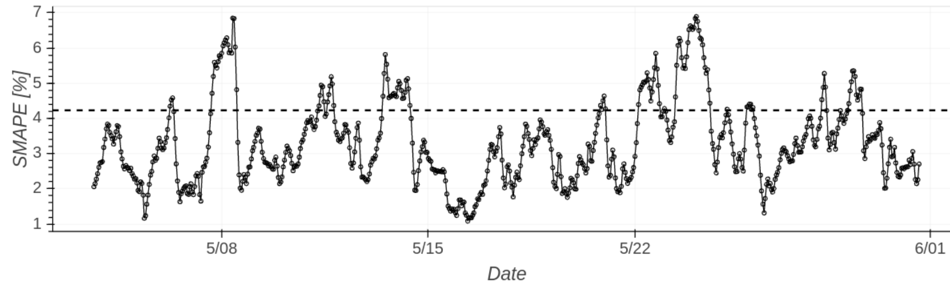
Figure 5.11: EDSLSTM metrics for 28 consecutive (rolling-day) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.



(a) MAE.



(b) RMSE.



(c) SMAPE.

Figure 5.12: EDSLSTM metrics for 672 consecutive (rolling-hour) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.

Regarding the rolling-hour collection of predictions, Figure 5.12a shows that 486 out of 672 (72%) predictions produce MAE values lower than the baseline's first-week average. Figure 5.12b shows that 539 out of 672 (80%) rolling-hour predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 5.12c shows that 538 out of 672 (80%) rolling-hour predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training EDSLSTM once, then using the model for rolling-hour predictions over a four-week interval mostly results in lower error metrics than forecasting with the baseline model. In addition, the baseline model would require a training computation time of close to 2,688 seconds to address the task, which is ×94 the time spent in training EDSLSTM.

Table 5.4: EDSLSTM Summarized Results.

Forecasting accuracy	
Daily predictions over a week MAE average	68.52
Weekly MAE increment w.r.t baseline	-31.6%
Daily predictions over a week RMSE average	88.22
Weekly RMSE increment w.r.t baseline	-31.1%
Daily predictions over a week SMAPE average	2.76
Weekly SMAPE increment w.r.t baseline	-34.7%
Computation time	
Number of trainable parameters	2.3 M
Training time with TPU acceleration (20 epochs)	28.64 s
Training time factor without TPU acceleration	×369
Training time factor for the baseline, 28 rolling-day predictions	×3.9
Training time factor for the baseline, 672 rolling-hour predictions	×94
Predictive performance degradation	
Four-week rolling-day predictions with MAE below the baseline	75%
Four-week rolling-day predictions with RMSE below the baseline	79%
Four-week rolling-day predictions with SMAPE below the baseline	82%
Four-week rolling-hour predictions with MAE below the baseline	72%
Four-week rolling-hour predictions with RMSE below the baseline	80%
Four-week rolling-hour predictions with SMAPE below the baseline	80%

Summary

This Chapter presents in detail EDSLSTM, a deep, encoder-decoder, LSTM-based architecture with attention mechanism that produces multi-horizon predictions for a univariate forecasting problem. This architecture was tested on an extensive dataset from the active power readings of an urban substation metering device. The baseline model used for evaluating EDSLSTM performance was a robust, cloud-based ARIMA implementation from the GC BigQuery data warehousing service.

Table 5.4 summarizes the experimental study. Some important facts that can be observed from that summary are:

- EDSLSTM overperforms the baseline in forecasting accuracy and training computation time.
- EDSLSTM exceeds the baseline in two typical evaluations for predictive performance degradation.
- TPU-based acceleration allows to train EDSLSTM $\times 369$ faster than CPUs.
- Conversely to DMSLSTM, EDSLSTM's predictive performance does not degrade from one-week to one-month intervals after training. This can be attributed to its longer input window and recurrent hidden state, as well as its attention mechanism.

The next Chapter presents the third neural network architecture selected for TPU-based implementation: a self-attention-based, deep transformer for TSF identified as BSCTRFM.

Chapter 6

Transformer Architecture for Local Forecasting

This Chapter presents in detail the Basic Transformer (BSCTRFM) architecture that I adapted and implemented for local TSF. Section 6.1 presents previous research work concerning the application of Transformer-based architectures for TSF. Section 6.2 presents in detail the architecture and the parametrization of the model. Section 6.3 presents the results achieved by BSCTRFM concerning forecasting accuracy, computation times, and predictive performance degradation.

6.1 Related Work

BSCTRFM is a DL model that implements the functionality of the original Transformer neural network [Vaswani et al., 2017] for multi-step, local, univariate TSF applications. The Transformer architecture provides important advantages for modeling Seq2Seq problems, such as capturing long-term dependencies from the data and allowing high-extent parallelization (see Subsection 2.3.3). For this reason, a number of TSF models that deliver state-of-the-art results have been developed on the basis of this neural network.

The `Deep Transformer` [Wu et al., 2020a] implements the basic architecture of the original NLP transformer [Vaswani et al., 2017] adapted to the TSF domain, providing a practical context to discuss further model improvements. This Transformer includes an encoder-decoder architecture, multiple layers featuring multi-head, full self-attention, layer normalization,

residual connections, and an iterative strategy for multi-horizon prediction. The `Temporal Fusion Transformer` [Lim et al., 2021] implements an interpretable MHA block on top of a Seq2Seq, LSTM-based encoder-decoder. This two-layer architecture is able to learn both short- and long-term dependencies on time series data. It uses a direct strategy for multi-horizon forecasting and can learn regime-specific temporal dynamics from time series.

Among the TSF research projects focused on the Transformer architecture a significant subset comprises projects that improve the Transformer performance by replacing the canonical, dense-matrix, full self-attention mechanism with different sparse-matrix attention mechanisms. The `LogSparse Transformer` [Li et al., 2019b] implements a decoder-only architecture that employs convolutional self-attention to provide queries and keys with enhanced locality knowledge. It also introduces *LogSparse* self-attention, which reduces the transformer complexity from $\mathcal{O}(L^2)$ to $\mathcal{O}(L(\log L)^2)$, where L is the input sequence length. The `Adversarial Sparse Transformer` [Wu et al., 2020b] implements a sparse attention mechanism by replacing the $softmax$ activation function in the attention heads with the α-*entmax* activation. It uses an encoder-decoder architecture for primary forecasting tasks and further utilizes it as the *generator* of a GAN. The *discriminator* of the GAN is trained with predicted and ground truth time series that classifies them as generated or real. A composite loss function, including the generator and the discriminator losses, is optimized during *adversarial training* of the complete network. This training mitigates the error accumulation on further iterative inference. The `Informer` [Zhou et al., 2021] proposes *ProbSparse* self-attention which reduces the Transformer complexity to $\mathcal{O}(L(\log L))$, where L is the input sequence length. It also introduces a self-attention distilling operation across the Transformer layers, that decreases memory usage, and a generative-style decoder produces all the prediction outputs with only one forward step.

A recent research line of Transformer-based TSF includes projects implementing series decomposition as a built-in block of the Transformer architecture that radically modifies the canonical or sparse self-attention mechanism. The `Autoformer` [Xu et al., 2021c] introduces time series decomposition to seasonal and trend-cyclical sub-series as an internal operation of the forecasting model. It also replaces self-attention with an Auto-Correlation mechanism, which focuses on period-based relationships at the sub-series level. The `ETSformer` [Woo et al., 2022] replaces self-attention with exponential smoothing attention and frequency attention and also redesigns the Trans-

former architecture with modular decomposition blocks that allow the model to learn to decompose the time series into interpretable components.

6.2 Experimental Study

This section presents the experimental study with the BSCTRFM architecture. It discusses the electric power dataset used in the forecasting process. Later, it expands on the specifics of the model architecture. Finally, it reports the model's parametrization for the use case.

6.2.1 Dataset

To benchmark the BSCTRFM forecasting model, I used the same active power, hourly resolution, urban substation meter dataset employed for DMSLSTM and EDSLSTM, as described in Subsection 4.2.1. As the time window specifications for BSCTRFM SLDB construction differ between DMSLSTM and EDSLSTM, the three models' resulting time intervals for training, evaluation, and test datasets are not coincident. The main difference is in the sizes of the test datasets, with 841 example rows available for DMSLSTM, 2,121 example rows available for EDSLSTM, and 1,873 example rows available for BSCTRFM.

6.2.2 Architecture Description

Implementing the BSCTRFM architecture comprises, like in the DMSLSTM and EDSLSTM cases, three stages: Supervised-Learning DataBase (SLDB) generation, model architecture definition, and model training. In the first stage, features, and targets (or *labels*) are extracted from historical time series of electric power load and stored in the corresponding SLDB. In the second stage, a specific BSCTRFM architecture is defined on the basis of the input SLDB and the model's composite structure. Finally, in the third stage, the BSCTRFM is trained on the SLDB and the resulting model is evaluated and deployed for prediction.

Figure 6.1 illustrates the BSCTRFM architecture for local forecasting. The SLDB that provides an input for the BSCTRFM network is generated from the historical record, at 1-hour resolution, of the active power measured at the substation level and the embedding dimension m. The result

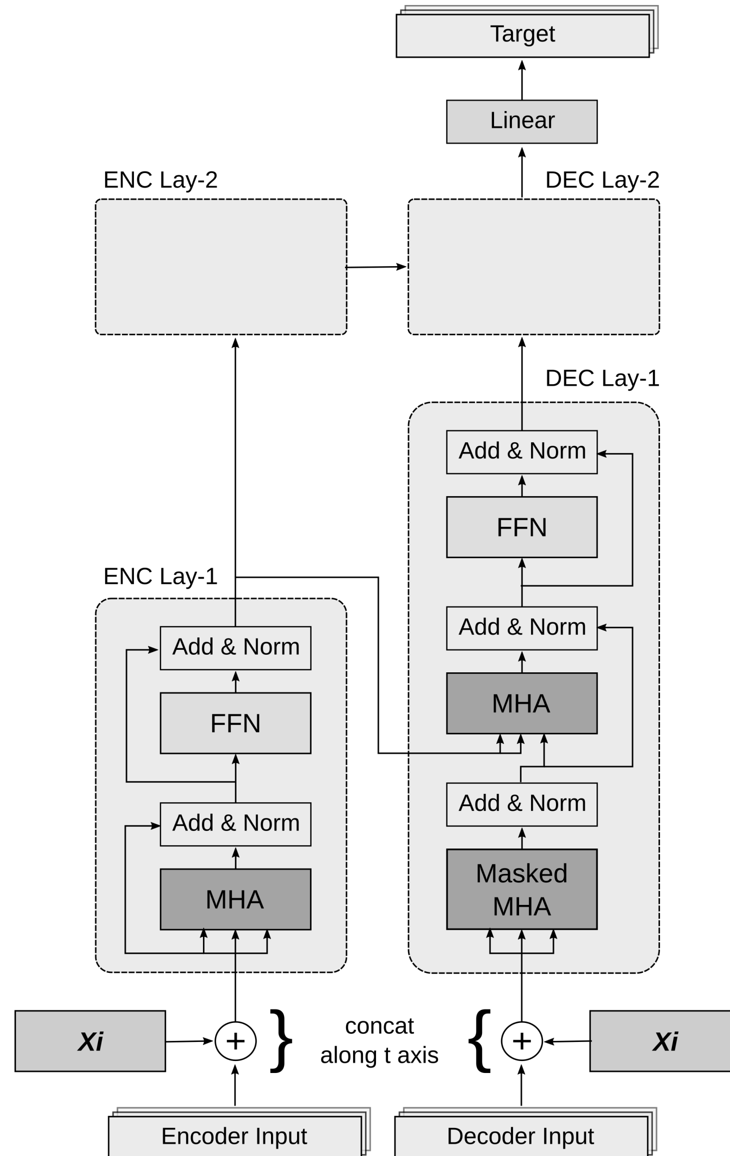


Figure 6.1: BSCTRFM Network Architecture for Local Forecasting. Adapted from Figure 2.9. The gray boxes labeled Encoder Layer 2 and Decoder Layer 2 have identical internal structures as their corresponding downstream layers.

is a collection of hourly-based time windows of the form $\{z_1, z_2, \ldots, z_m\}$ (see Subsection 2.1.1). Once generated, the SLDB is serialized and persisted as a collection of example rows, where each example row includes the following features:

- *encoder input*: the adjacent-hour, m-sized delay vector of active power readings that is fed to the encoder object,
- *encoder input date-time encoding*: a (m, 4) vector that encodes the date-time values of the encoder input sequence.
- *decoder input*: the adjacent-hour, m-sized delay vector of active power readings that is fed to the decoder object,
- *decoder input date-time encoding*: a (m, 4) vector that encodes the date-time values of the decoder input sequence.
- *target*: the active power value vector along the forecasting horizon, assumed to be a function of the encoder and decoder input delay vectors.

Each timestep date-time value is encoded in a 4D vector $datetime_{enc}$ obtained from the following equation:

$$datetime_{enc} = (\sin(\frac{2\pi h_D}{24}), \cos(\frac{2\pi h_D}{24}), \sin(\frac{2\pi d_W}{7}), \cos(\frac{2\pi d_W}{7})) \qquad (6.1)$$

where h_D is a consecutive integer that identifies the hour of the day when the observation was acquired (0-23), and d_W is a consecutive integer that identifies the day of the week when the observation was acquired (0-6).

The BSCTRFM network consists of two objects or sub-networks: the encoder object (left column in the figure) and the decoder object (right column). Also, three types of MHA components are managed into those objects: causal self-attention, global self-attention, and cross-attention (see Subsection 2.3.3).

The decoder object mainly processes the *decoder input* delay vector to fit it against the *target* (also referred to as *label*) vector. The causal self-attention comprises the lowermost MHA component shown in the decoder layer (Masked MHA) as well as the add-and-normalization layer on top of it. It takes the concatenation of the decoder input $\{z_{i,e}, z_{i,e+1}, \ldots, z_{i,e+d}\}$ and the $datetime_{enc}$ vectors along the decoder input (the dynamic covariates x_i) as the queries, keys, and values for the self-attention operation. Based on this

information, the decoder layer, and therefore the decoder object, contribute to fitting the model to output the target sequence $\{z_{i,e+1}, z_{i,e+2}, \ldots, z_{i,e+d+1}\}$. As a result, $z_{i,e+d+1}$, which is the only unseen value for each example row, is predicted.

Two important facts deserve to be emphasized from the above operation: First, the causal self-attention has to be masked with a *look-ahead* only mask (a lower triangular matrix with zero-value coefficients over the main diagonal) to avoid leaking future information when predicting the value of the sequence at an earlier time step. Second, relevant autoregressive, decoder-only Transformer models for TSF that employ causal self-attention only, in conjunction with other specialized components, have reported state-of-the-art results, as in [Li et al., 2019b].

The encoder object mainly processes the *encoder input* delay vector to generate a more robust, augmented *context* for the decoder object operation. The global self-attention encompasses the MHA component in the encoder layer as well as the add-and-normalization layer on top of it. It takes the concatenation of the encoder input $\{z_{i,0}, z_{i,1}, \ldots, z_{i,e}\}$ and the $datetime_{enc}$ vectors along with the encoder input (the dynamic covariates x_i) as the queries, keys, and values for the self-attention operation. Based on this information, the encoder layer, and therefore the encoder object, contribute to fitting the model by passing the encoder output (augmented context) as the keys and values for the cross-attention component. In other words, instead of using the decoder input as the only context for the decoder sub-network (as it happens with autoregressive, decoder-only Transformers), global attention generates a wider context based on encoder input sequences that take into consideration the information further in the past.

Finally, the cross-attention comprises the uppermost MHA component in the decoder layer as well as the add-and-normalization layer on top of it. Its self-attention operation is performed by taking the output of the causal attention as the queries, and the output of the global attention as the keys and values. To complete the training graph, the output of the uppermost decoder layer in the decoder sub-network is projected back to the dimensionality of the target using a multi-layer, dense neural network.

6.2.3 Model Parametrization

The model's parametrization is summarized in Table 6.1. I trained BSC-TRFM using the Adam optimizer [Kingma and Ba, 2015] with $\beta_1 = 0.9$,

Table 6.1: BSCTRFM parameters.

SLDB parameters	
context window	168
# of training examples	17,200
# of evaluation examples	2,544
# of test examples	1,873
architecture parameters	
input dropout	0.1
encoder input length	168
decoder input length	168
model dimension d_{model}	256
number of attention heads	4
number of layers	2
feed-forward dimension d_{ff}	512
layer dropout	0.2
DENSE structure	[64, 8, 1]
DENSE activation functions	[ReLU, ReLU, sigmoid]
forecast horizon	24
# of trainable parameters	2.7 M
training parameters	
learning rate schedule exp_1	-0.5
learning rate schedule exp_2	-0.85
batch size	[128, 256, 512, 1024]
# of training steps	[4800, 2,400, 1200, 600]
# of epochs	36

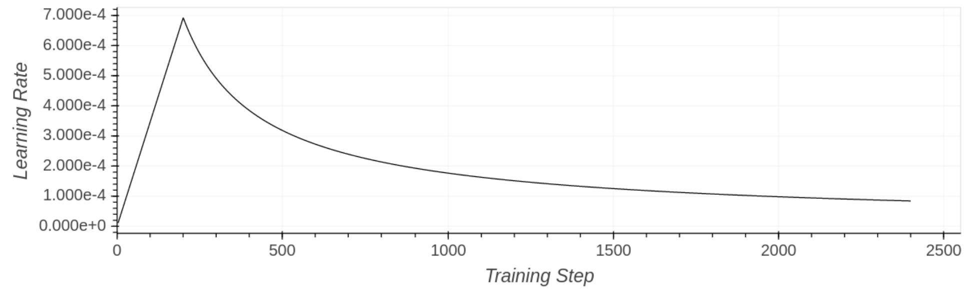


Figure 6.2: BSCTRFM Learning Rate Schedule. The learning rate starts at 0 and linearly increases during a given number of warmup steps, then it decreases proportionally to the inverse square root of the step number.

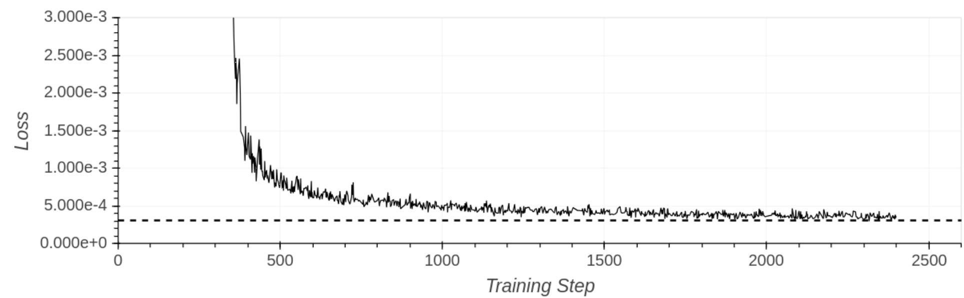


Figure 6.3: BSCTRFM Training Loss. The dotted line corresponds to the loss metric on evaluation, which is calculated only once, when training is completed.

$\beta_2 = 0.98$, and $\epsilon = 10^{-8}$, and a custom learning rate $lrate = d_{model}^{exp_1} \cdot \min(step_num^{exp_2}, step_num \cdot warmup_steps^{exp_2 - 1})$ which results in increasing the learning rate linearly for the first *warmup_steps* training steps, and decreasing it thereafter proportionally to the inverse of the $\frac{1}{exp_2}$ root of the step number [Vaswani et al., 2017]. This learning rate schedule is shown in Figure 6.2. The MSE was used as the training and evaluation loss function. The RMSE was used as an additional custom metric, via the `metric_fn` component of the TPUEstimator API.

I used a grid search to tune a couple of training hyperparameters: learning rate and training batch size. As previously stated, the learning rate is given by a custom schedule that is a function of the number of warmup steps, the model dimension, and the current training step, then the number of warmup steps was the variable that defined the grid search. The number of epochs was fixed to 36 by modifying the batch size according to the total training steps, approximately $\times 12$ the warmup steps.

BSCTRFM was trained 10 different times, or executions, for each parameter set. Figure 6.3 shows the behavior of the main loss metric (MSE) over the training steps for one of the executions of the selected configuration. The dotted line shows the value of the error metric for the model evaluation at the end of the training stage. In order to achieve high efficiency, training a model on TPUs is not compatible with frequent evaluation, as it regularly happens with CPU-based training. The reason is starting the evaluation stage requires completely stopping the training loop, including shut down the TPU. As the time required to shut down and restart the TPU is usually higher than the time consumed by one training loop, it is a good practice to schedule evaluation, whenever possible, only at the check-pointing time, when the TPU is required to shut down in order to save the model training progress.

6.3 Results and Discussion

This Section presents the baseline used for evaluating BSCTRFM performance. It also discusses the results of the model concerning forecasting accuracy, computation times, and predictive performance degradation.

6.3.1 Baseline

As in the DMSLSTM and EDSLSTM cases, BigQuery's ARIMA implementation was used to provide a baseline for evaluating BSCTRFM results. The BigQuery's ARIMA parametrization is detailed in Subsection 4.3.1. As previously stated, this parametrization was intended to keep the time required for fitting each model under 4 seconds.

6.3.2 Metrics

Three error metrics were used for BSCTRFM comparison with the baseline: MAE (4.1), RMSE (4.2), and SMAPE (4.4).

6.3.3 Forecasting Accuracy

Figures 6.4 to 6.10 show the predictive performance on a one-week period, for daily predictions of active power produced by BSCTRFM and BigQuery ARIMA. It is worth mentioning that the BSCTRFM figures and metrics in the comparison come from the average of the ten training executions of the model, and not from the best one.

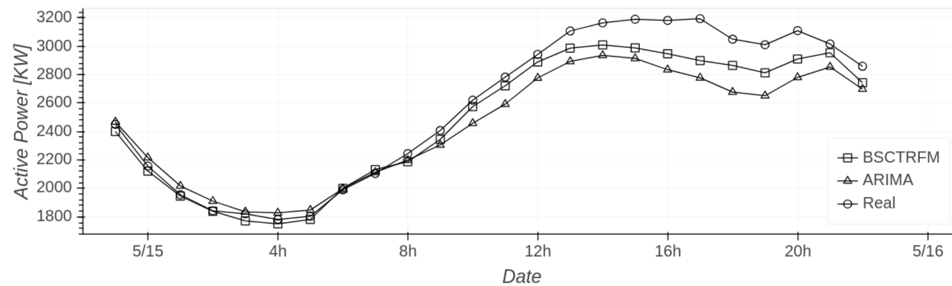


Figure 6.4: BSCTRFM vs. ARIMA vs. Real for Day 1. BSCTRFM outperforms ARIMA from 09:00 to 22:00 hours.

Table 6.2 summarizes the error metrics obtained from inference on the first week of the test dataset. ARIMA metrics come from the only execution of this baseline while EDSLSTM metrics are average plus-minus the standard deviation of the executions set. The last column in the table shows the change in the error metrics that results when switching from the baseline to BSCTRFM, as a percentage of the metric achieved by the baseline.

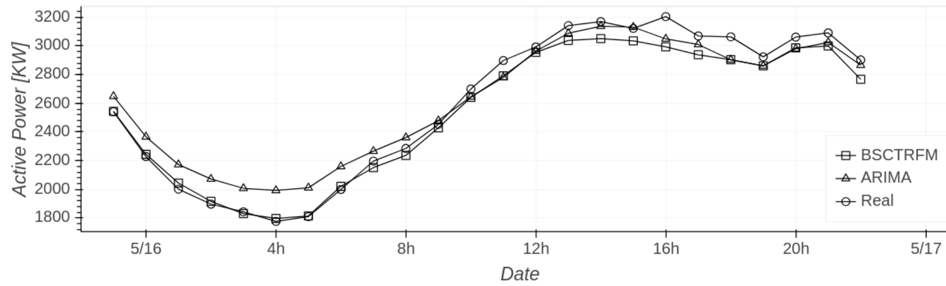


Figure 6.5: BSCTRFM vs. ARIMA vs. Real for Day 2. BSCTRFM outperforms ARIMA from 23:00 to 08:00 hours. ARIMA is better from 13:00 to 17:00 hours.

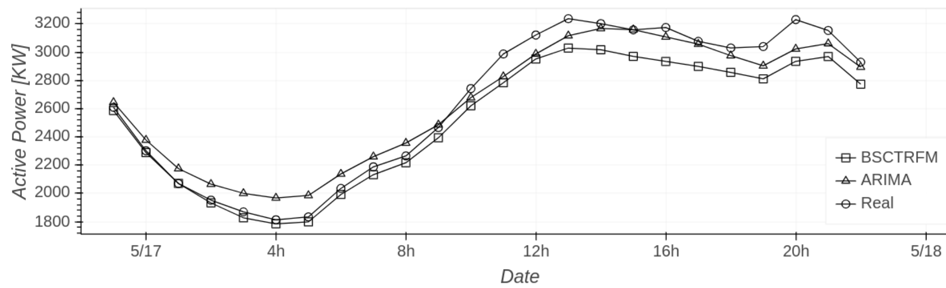


Figure 6.6: BSCTRFM vs. ARIMA vs. Real for Day 3. BSCTRFM outperforms ARIMA from 00:00 to 09:00 hours. ARIMA is better for the rest of the day.

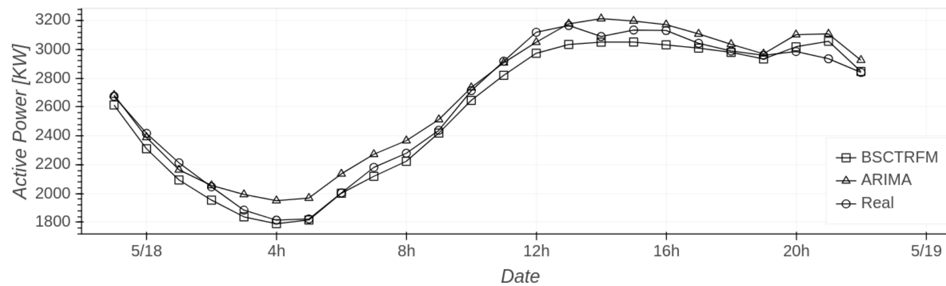


Figure 6.7: BSCTRFM vs. ARIMA vs. Real for Day 4. BSCTRFM outperforms ARIMA from 03:00 to 09:00 hours.

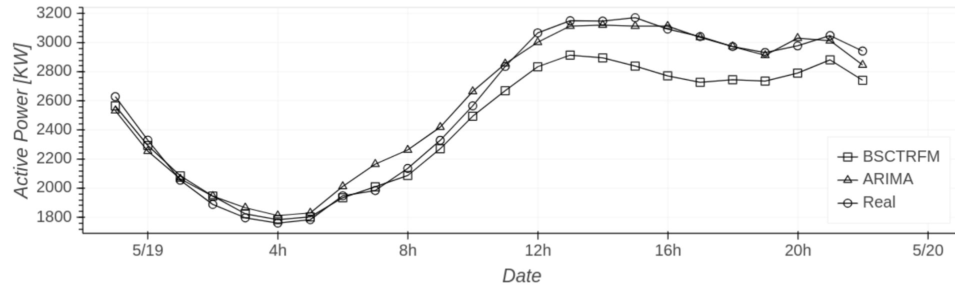


Figure 6.8: BSCTRFM vs. ARIMA vs. Real for Day 5. ARIMA outperforms BSCTRFM from 11:00 to 22:00 hours.

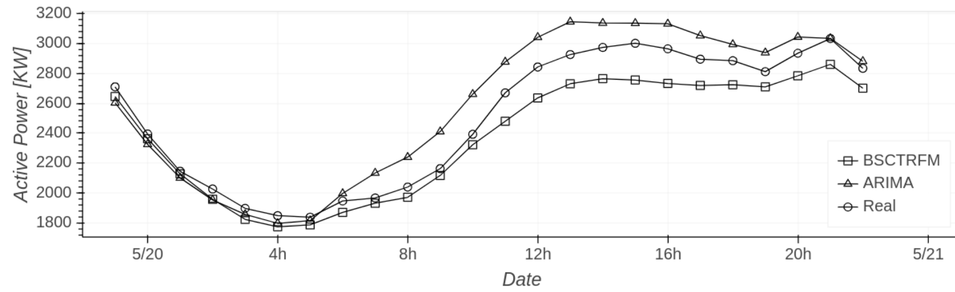


Figure 6.9: BSCTRFM vs. ARIMA vs. Real for Day 6. BSCTRFM outperforms ARIMA from 07:00 to 10:00 hours.

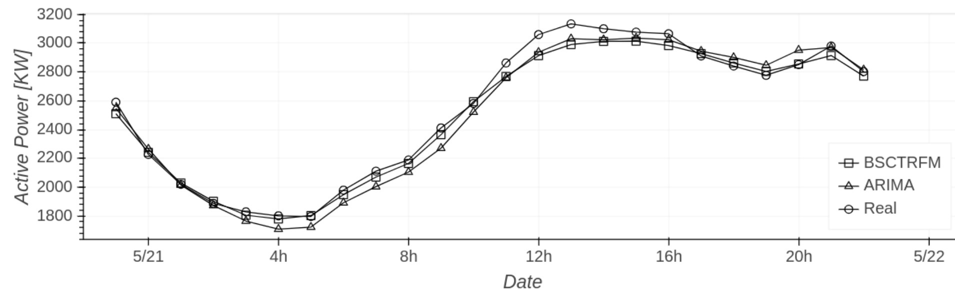


Figure 6.10: BSCTRFM vs. ARIMA vs. Real for Day 7. BSCTRFM outperforms ARIMA from 04:00 to 10:00 hours.

Table 6.2: BSCTRFM Vs. ARIMA Comparison.

MAE			
Interval	ARIMA	BSCTRFM	Δ(%)
Day 1	160.6660	**98.0120** ± 15.1866	**-39.00**
Day 2	100.7960	**69.7494** ± 16.6103	**-30.80**
Day 3	**90.7451**	121.9528 ± 10.0949	34.39
Day 4	71.1904	**65.9552** ± 11.4839	**-7.35**
Day 5	**58.2965**	138.2748 ± 25.6004	137.19
Day 6	124.5701	**119.1735** ± 33.1681	**-4.33**
Day 7	65.9860	**50.1857** ± 9.6781	**-23.94**
RMSE			
Interval	ARIMA	BSCTRFM	Δ(%)
Day 1	205.4609	**127.4635** ± 17.8284	**-37.96**
Day 2	117.6361	**89.5919** ± 20.8463	**-23.84**
Day 3	**104.4511**	150.1624 ± 13.0129	43.76
Day 4	86.0171	**79.5432** ± 13.1284	**-7.53**
Day 5	**71.5076**	176.9964 ± 33.3925	147.52
Day 6	144.9715	**140.0774** ± 39.3061	**-3.38**
Day 7	75.4550	**65.5064** ± 14.2024	**-13.18**
SMAPE			
Interval	ARIMA	BSCTRFM	Δ(%)
Day 1	5.8152	**3.5497** ± 0.5502	**-38.96**
Day 2	4.3112	**2.5155** ± 0.5738	**-41.65**
Day 3	**3.6705**	4.3103 ± 0.3610	17.43
Day 4	2.8952	**2.5464** ± 0.4579	**-12.05**
Day 5	**2.4724**	4.9897 ± 0.9020	101.81
Day 6	4.8394	**4.6401** ± 1.2906	**-4.12**
Day 7	2.7351	**1.8822** ± 0.3484	**-31.18**

BSCTRFM outperforms ARIMA on five of the seven days, with the exception of Day 3 (see Figure 6.6) and Day 5 (see Figure 6.8). For the days when BSCTRFM defeats the baseline, fair to important reductions in the error metrics are achieved from switching ARIMA for BSCTRFM: 4% to 39% for MAE, 3% to 37% for RMSE, and 4% to 42% for SMAPE. Conversely, on the days that BSCTRFM does not outperform the baseline, the error metrics growths due to switching from ARIMA to BSCTRFM are significantly higher: 34% to 137% for MAE, 44% to 148% for RMSE, and 17% to 102% for SMAPE.

As a common result, both EDSLSTM and ARIMA are able to capture the daily pattern in the active power time series. However, BSCTRFM shows a higher capacity than the baseline in capturing the first part of the short-term pattern (up to the 17:00 hours), while ARIMA has a higher capacity to capture the sharper profile of the nighttime pattern. This reveals the need to increase the capacity of the model which can be done by adding encoder and decoder layers to its architecture, training for a longer number of epochs of the dataset, or both.

6.3.4 Computation Time

Despite the huge size of the BSCTRFM model (2.7 million trainable parameters), the cloud-based TPU implementation is able to train it for 36 epochs of the reference dataset in just 28.4 seconds of wall time. For comparison, I trained this configuration of BSCTRFM without the TPU accelerator using a Google Cloud Compute Engine n1-standard-4 virtual instance with 4 Intel Haswell vCPUs and 16 GB RAM, in 284 seconds for 0.37 epochs of the reference dataset (6,400 examples), representing $\times 968$ the time required by the proposed infrastructure. It is worth to mention it was not possible to train BSCTRFM for a longer period without using TPU acceleration, as the training process crashed several times due to excessive CPU utilization.

To properly compare the computation time of BSCTRFM with the baseline, it is important to take into consideration the different training and inference approaches they apply. I addressed the basics of these approaches and their repercussions on the benchmark evaluation on Subsection 4.3.4. Table 6.3 shows a total of seven common use cases for short-term forecasting in the electric power distribution field. For each use case, the number of training events required by the baseline model to achieve the forecasting goal is presented, along with the involved training computation time. Ac-

Table 6.3: BSCTRFM Vs. ARIMA Computation Time Comparison.

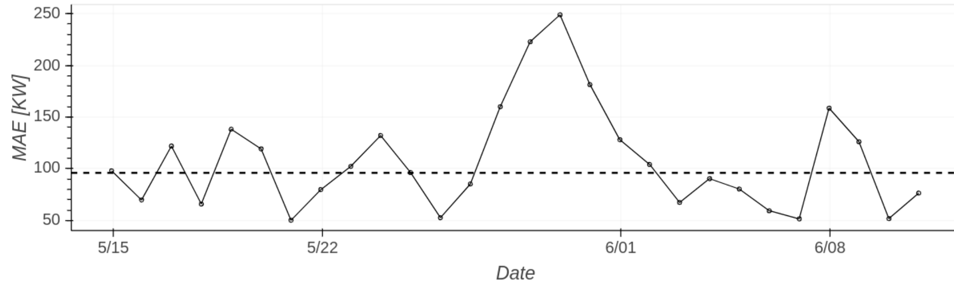
Number of 24-hour predictions	BSCTRFM	ARIMA	Use case
1	28.3504 ± 0.0064	4	Single prediction
7	(fixed)	28	Daily predictions over a week
14	(fixed)	56	Daily predictions over two weeks
24	(fixed)	96	Hourly predictions over a day
28	(fixed)	112	Daily predictions over four weeks
168	(fixed)	672	Hourly predictions over a week
672	(fixed)	2,688	Hourly predictions over four weeks

cording to the table, BSCTRFM outperforms the baseline in computation time whenever more than fourteen 24-hour forecasting intervals are required. For instance, training the baseline model a total of 14 times to produce one 24-hour prediction interval for each day of two weeks, requires 56 seconds, which is $\times 2$ time required to train BSCTRFM once. The remainder of this Section is dedicated to discussing the predictive performance of BSCTRFM in two widely-used, short-term forecasting scenarios: daily predictions over four weeks (28 inference executions) and hourly predictions over four weeks (672 inference executions).

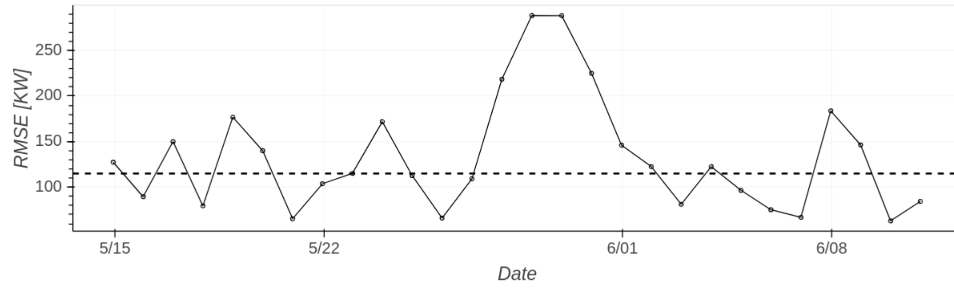
6.3.5 Predictive Performance Degradation

As in the DMSLSTM and EDSLSTM cases, I evaluated the predictive performance degradation of BSCTRFM using two collections of 24-hour-ahead predictions distributed over a four-week interval: the first collection consists of 28 predictions, each one made at the start of a consecutive day in the four-week interval (rolling day); and the second collection consists of 672 predictions, each one made at a consecutive hour in the four-week interval (rolling hour). For each collection of predictions MAE, RMSE, and SMAPE metrics were calculated. For each error metric, the average of the corresponding baseline's first-week values (see Table 6.2) was calculated, then used as a threshold to classify BSCTRFM's outcomes as better or worse with respect to the baseline.

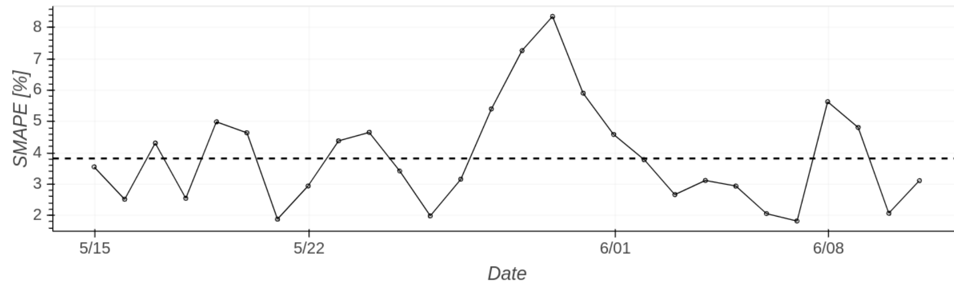
Regarding the rolling-day collection of predictions, Figure 6.11a shows that 13 out of 28 (46%) predictions produce MAE values lower than the



(a) MAE.

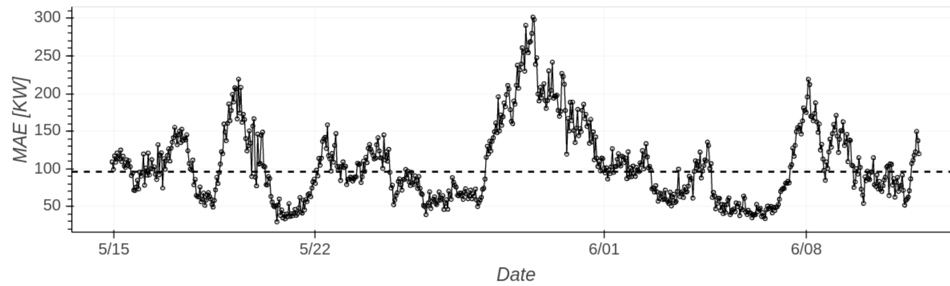


(b) RMSE.

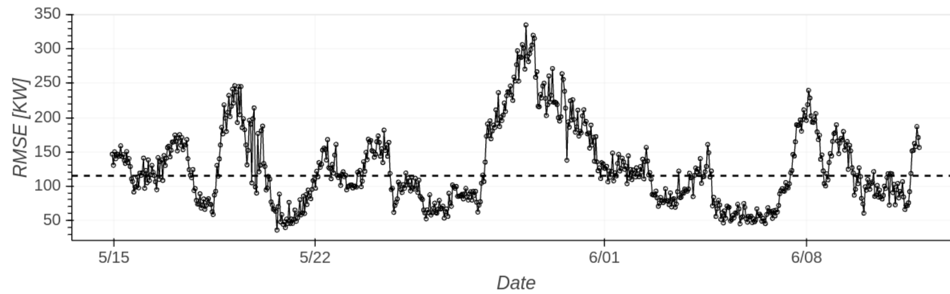


(c) SMAPE.

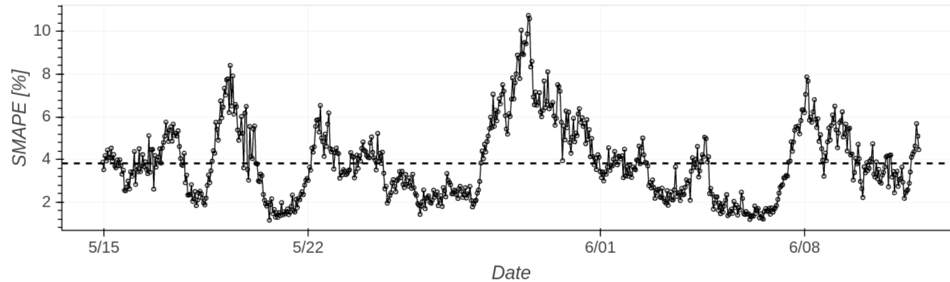
Figure 6.11: BSCTRFM metrics for 28 consecutive (rolling-day) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.



(a) MAE.



(b) RMSE.



(c) SMAPE.

Figure 6.12: BSCTRFM metrics for 672 consecutive (rolling-hour) 24-hour-ahead predictions. The dotted line shows the baseline's first-week average for each error metric.

baseline's first-week average. Figure 6.11b shows that 13 out of 28 (46%) rolling-day predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 6.11c shows that 16 out of 28 (57%) rolling-day predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training BSCTRFM once, then using the model for rolling-day predictions over a four-week interval fairly results in higher error metrics than forecasting with the baseline model. As a consequence, this version of BSCTRFM should be used for daily predictions over one week only. If the model is required to produce predictions up to a one-month interval after training it has to be trained and evaluated for a greater number of epochs. For comparison, producing rolling-day 24-hour predictions for the first week with the baseline model would require a training computation time of close to 28 seconds, which is fairly the same time spent in training BSCTRFM.

Regarding the rolling-hour collection of predictions, Figure 6.12a shows that 324 out of 672 (48%) predictions produce MAE values lower than the baseline's first-week average. Figure 6.12b shows that 331 out of 672 (49%) rolling-hour predictions produce RMSE values lower than the baseline's first-week average. Finally, Figure 6.12c shows that 360 out of 672 (54%) rolling-hour predictions produce SMAPE values lower than the baseline's first-week average. I conclude from this analysis that training BSCTRFM once, then using the model for rolling-hour predictions over a four-week interval fairly results in higher error metrics than forecasting with the baseline model. As a consequence, this version of BSCTRFM should be used for hourly predictions over one week only. If the model is required to produce predictions up to a one-month interval after training it has to be trained and evaluated for a greater number of epochs. For comparison, producing rolling-hour 24-hour predictions for the first week with the baseline model would require a training computation time of close to 672 seconds, which is $\times 24$ the time spent in training BSCTRFM. Nevertheless, BSCTRFM exhibits moderate instability, especially in the rolling-hour predictions set, and its training process should be further adjusted by attending to recommendations in the literature [Popel and Bojar, 2018], [Liu et al., 2020].

Table 6.4: BSCTRFM Summarized Results.

Forecasting accuracy	
Daily predictions over a week MAE average	94.76
Weekly MAE increment w.r.t baseline	-1.3%
Daily predictions over a week RMSE average	118.48
Weekly RMSE increment w.r.t baseline	3.0%
Daily predictions over a week SMAPE average	3.49
Weekly SMAPE increment w.r.t baseline	-8.6%
Computation time	
Number of trainable parameters	2.7 M
Training time with TPU acceleration (36 epochs)	28.35 s
Training time factor without TPU acceleration	×968
Training time factor for the baseline, 28 rolling-day predictions	×4.0
Training time factor for the baseline, 672 rolling-hour predictions	×95
Predictive performance degradation	
Four-week rolling-day predictions with MAE below the baseline	46%
Four-week rolling-day predictions with RMSE below the baseline	46%
Four-week rolling-day predictions with SMAPE below the baseline	57%
Four-week rolling-hour predictions with MAE below the baseline	48%
Four-week rolling-hour predictions with RMSE below the baseline	49%
Four-week rolling-hour predictions with SMAPE below the baseline	54%

Summary

This Chapter presents in detail BSCTRFM, a DL architecture based on the canonical Transformer architecture (multi-layer encoder-decoder, full self-attention MHA layers, and residual connections) that produces multi-horizon predictions for a univariate forecasting problem. This architecture was tested on an extensive dataset that comes from the active power observations of an urban substation metering device. The baseline model used for evaluating EDSLSTM performance was a robust, cloud-based ARIMA implementation from the GC BigQuery data warehousing service.

Table 6.4 summarizes the experimental study. Some important facts that can be observed from that summary are:

- BSCTRFM overperforms the baseline in forecasting accuracy for the basic short-term forecasting task, which is producing daily predictions over a week.

- BSCTRFM ties the baseline in training computation time for the above-mentioned task. For any larger short-term forecasting task, BSCTRFM overperforms the baseline.

- BSCTRFM does not exceed the baseline in the evaluations for predictive performance degradation.

- TPU-based acceleration allows to train BSCTRFM $\times 968$ faster than CPUs, which is equivalent to saying typical CPU-based architectures are impeded to train this architecture.

- Conversely to DMSLSTM and EDSLSTM, BSCTRFM's predictive performance greatly degrades from one-week to one-month intervals after training. This is attributed to the instability the model shows in successive rolling-hour loss metrics. As in other Transformers, this situation has to be addressed with a further experimental study on the learning rate schedule design and optimization.

The next Chapter extends the experimental study on BSCTRFM by applying it to the global forecasting problem.

Chapter 7

Transformer Architecture for Global Forecasting

This Chapter presents in detail the Basic Transformer (BSCTRFM) architecture for global TSF. Section 7.1 establishes a context by discussing the datasets that have been predominantly used in the literature for benchmarking the application of Transformer-based architectures to global TSF. Section 7.2 presents the two standard datasets used in the experiments with BSCTRFM. It also presents in detail the architecture and the parametrization of the model. Section 7.3 presents the collection of models for global TSF used as a benchmark for evaluating BSCTRFM's performance. On this basis, results achieved by the model concerning forecasting accuracy and computation times are discussed.

7.1 Related Work

Previously, Section 6.1 discussed a collection of research projects that have contributed relevant applications of the Transformer architecture to global, multi-horizon TSF. Several publicly available datasets have been predominantly used to standardize the evaluation of such approaches. [Li et al., 2019b] report experiments based on the datasets referred to in the litera-

ture as `electricity`[1], `traffic`[2], `solar`[3], `wind`[4], and `M4Hourly`[5]. [Wu et al., 2020b] report experiments on `electricity` and `traffic`. [Lim et al., 2021] report experiments on `electricity`, `traffic`, and `retail`[6]. [Zhou et al., 2021] report experiments on `ETT`[7], `weather`[8], and `electricity`. Finally, [Xu et al., 2021c] and also [Woo et al., 2022] report experiments on `ETT`, `electricity`, `traffic`, `weather`, and `ILI`[9].

A couple of these predominant datasets were selected to train and evaluate the global forecasting implementation of BSCTRFM: `electricity` and `traffic`. The motivation behind this selection comprises two important facts. First, most of the projects in the collection of global TSF approaches report forecasting accuracy results on these datasets. Second, the only results on computation training time found in the literature correspond to these datasets.

7.2 Experimental Study

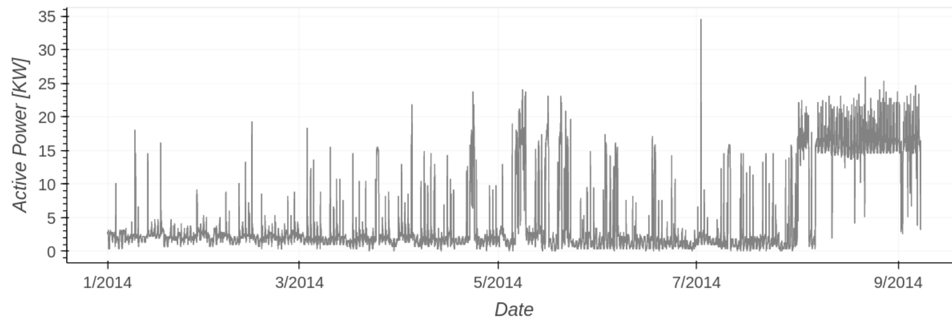
This Section discusses the datasets, data preprocessing, architecture, model parametrization, and inference procedures used in the experimental study.

7.2.1 Datasets and Data Preprocessing

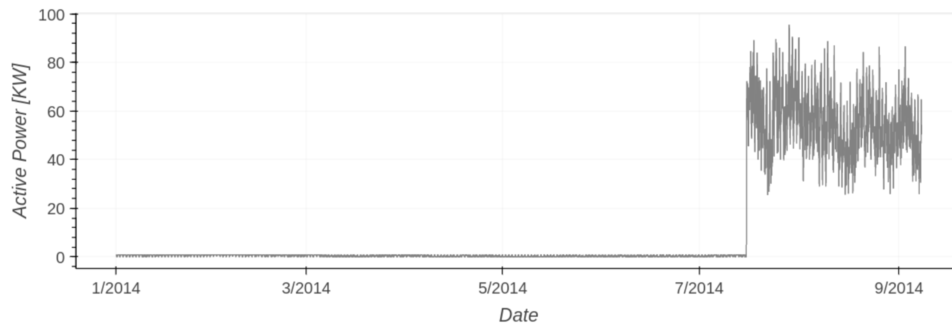
The global forecasting implementation of BSCTRFM was trained and tested on two widely-known datasets: `electricity` that contains hourly time series of the electricity consumption of 370 customers, and `traffic` which is composed of 963 time series with hourly observations of car lane occupancy of San Francisco bay area freeways.

Figure 7.1 shows three time series from the `electricity` dataset. As expected, these time series exhibit daily and weekly patterns. However, they also show conditions that tend to difficult the forecasting process, such as

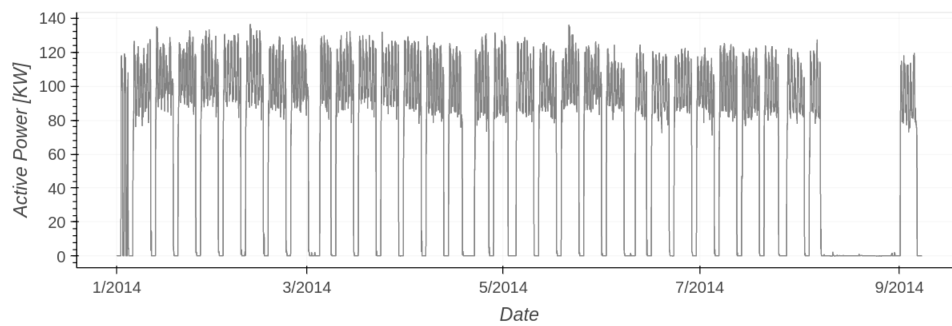
[1]http://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014
[2]http://archive.ics.uci.edu/ml/datasets/PEMS-SF
[3]http://www.nrel.gov/grid/solar-power-data.html
[4]http://www.kaggle.com/sohier/30-years-of-european-wind-generation
[5]http://paperswithcode.com/dataset/m4
[6]http://www.kaggle.com/c/favorita-grocery-sales-forecasting
[7]http://paperswithcode.com/dataset/ett
[8]https://www.bgc-jena.mpg.de/wetter
[9]http://gis.cdc.gov/grasp/fluview/fluportaldashboard.html



(a) MT-001.

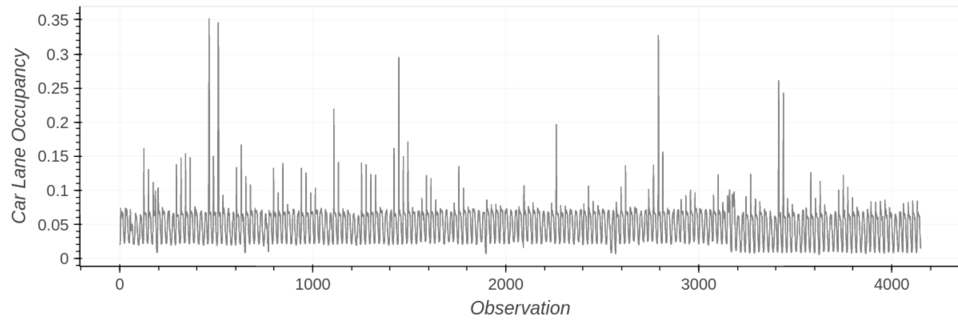


(b) MT-066.

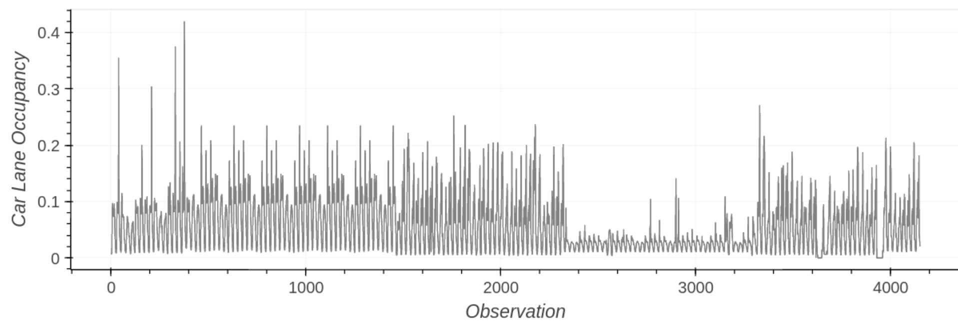


(c) MT-127.

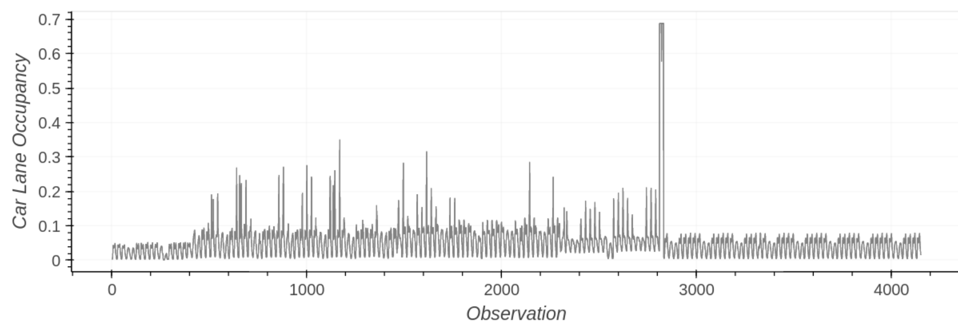
Figure 7.1: Three time series from the `electricity` dataset.



(a) ST-400000.



(b) ST-400828.



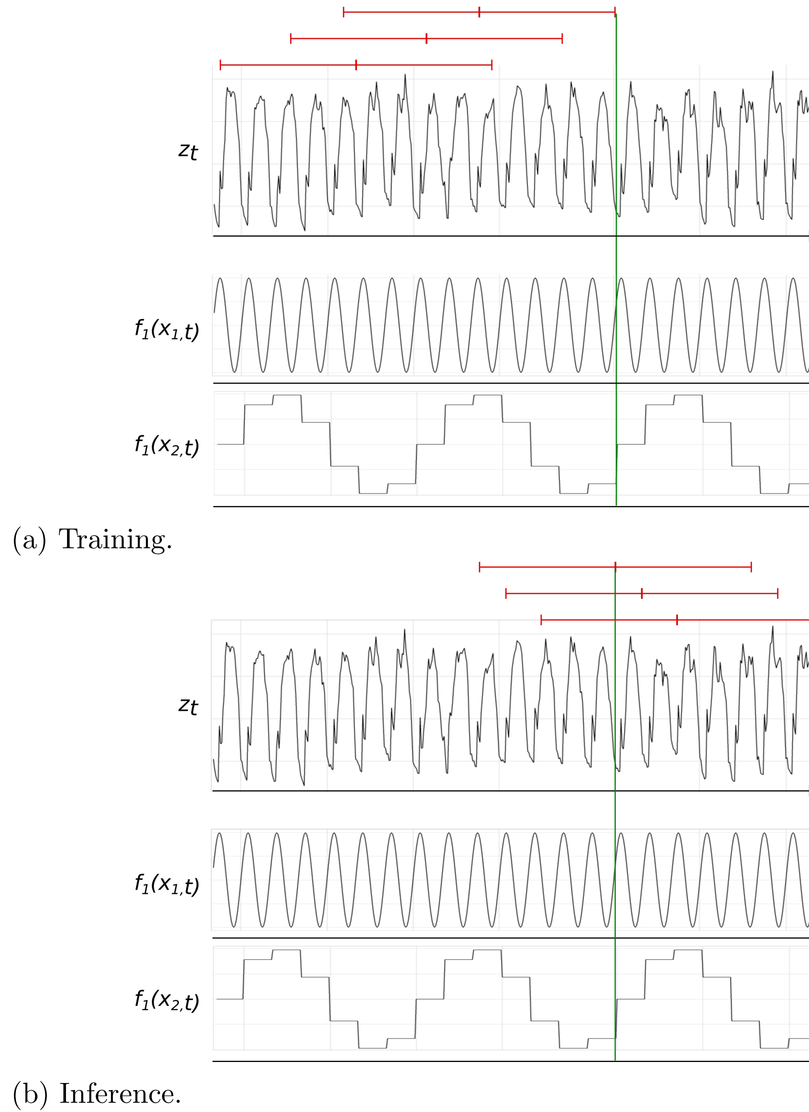
(c) ST-401623.

Figure 7.2: Three time series from the `traffic` dataset.

consumption regime changes (MT-101), incomplete time series (MT-066), and missing values (MT-127). As previously stated, global forecasting approaches are able to overcome the conditions mentioned above by leveraging the relatedness inside collections of similar time series, therefore capturing patterns across time series. Figure 7.2 shows three time series from the `traffic` dataset.

Figure 7.3 illustrates BSCTRFM data setup for the training and inference stages. The input to the model includes all of the univariate time series z_t, and, for each time series, two positional encoding functions $f_1(x_{k,t})$ and $f_2(x_{k,t}), k \in [1, j]$ of j time-dependent covariates, where f_1 and f_2 define sine- and cosine-based cycles over the period length of the covariate, e.g. 24 for the hour of the day. In the figure, the part of the time series to the left of the green vertical line is the data that is presented to the model for training. The part to the right of the green line is the data used to test the model once it is trained. The bisected red horizontal lines represent time windows used as data examples: the left section of the red line is the data passed as input to the model (*conditioning range*), while the right section is the data used as the target or output from the model (*prediction range*). During training (top), time windows are required to completely lie to the left of the green line, as only *seen data* can be used for training or evaluation purposes. During inference (bottom), the prediction range of the data examples passed to test the model must entirely lie to the right of the green line (*unseen data*).

Data setup is completed with two additional components (not shown in the figure): a time-dependent covariate with the consecutive value of hour on the observation timestamp measured from the time series starting point (*age-covariate*), and a scalar time series identifier which is passed as a static covariate for the model, then conformed into a learnable embedding which provides a finite-dimensionality space where time series with similar behavior tend to group. The `electricity` and `traffic` datasets were divided into training, validation, and test sets following the procedure described in [Yu et al., 2016], and later applied by [Salinas et al., 2020], [Li et al., 2019b], and [Lim et al., 2021]. Min-max normalization to $[0, 1]$ was applied to both datasets, in order to compensate for the large difference in time series values in `electricity` and to speed up the model convergence in `traffic`. Proper normalization scalers were acquired only from training data to avoid leaking information from the future into the model. Once preprocessed and separated, data was persisted to cloud storage as sequences of binary strings (`TFRecords`) in accordance with best-TPU-performance guidelines. A data



(a) Training.

(b) Inference.

Figure 7.3: Data setup for BSCTRFM at training and inference time. In the first row, a single univariate time series to forecast z_t. In the second and third rows, two exemplary sine-based positional encodings applied on the hour of the day $x_{1,t}$ and the day of the week $x_{2,t}$ of the observation timestamp.

ingestion pipeline for training the model was defined on the basis of the `tf.data.dataset` TensorFlow API. Prefetching was also used for parallel preprocessing, that means allowing the pipeline to start preparing the next batch while still training the current batch. To ensure good throughput during ingestion from cloud storage, individual `TFRecord` files were prepared for each time series identifier, resulting in 370 50-MB files for `electricity` and 963 33-MB files for `traffic`.

7.2.2 Architecture Description

BSCTRFM architecture for local forecasting, as discussed in Subsection 6.2.2 was also applied to the global forecasting case, after implementing two important modifications (see Figure 7.4). First, two dimensions were added to the date-time encoding $datetime_{enc}$ as day-of-the-year sine- and cosine-based functions. These additional dimensions provide an *age covariate* that allows aligning time series with different sizes, thus different starting points. As a result, each timestep date-time value is encoded in a 6D vector $datetime_{enc}$ obtained from the following equation:

$$datetime_{enc} = (\sin(\frac{2\pi h_D}{24}), \cos(\frac{2\pi h_D}{24}), \sin(\frac{2\pi d_W}{7}), \cos(\frac{2\pi d_W}{7}), \sin(\frac{2\pi d_Y}{365}), \cos(\frac{2\pi d_Y}{365})) \tag{7.1}$$

where h_D is a consecutive integer that identifies the hour of the day when the observation was acquired (0-23), d_W is a consecutive integer that identifies the day of the week when the observation was acquired (0-6), and d_Y is a consecutive integer that identifies the day of the year when the observation was acquired (0-365). Second, a time series identifier was used as a static covariate $\mathbf{s}_i$. It was formed as a consecutive integer (1 to 370 for `electricity`, 1 to 963 for `traffic`), then it was embedded in an output embedding dimension for concatenation, along the time axis, with the time series value $\mathbf{z}_{i,t}$ and the dynamic covariates $\mathbf{x}_{i,t}$.

7.2.3 Model Parametrization

Table 7.1 offers a global summary of the experiment setup, including the dataset statistics and the model's architecture, training, and testing parameters. As in the local forecasting case, I trained BSCTRFM using the Adam optimizer [Kingma and Ba, 2015] with $\beta_1 = 0.9$, $\beta_2 = 0.98$, and $\epsilon = 10^{-8}$, and a custom learning rate $lrate = d_{model}^{exp_1} \cdot \min(step_num^{exp_2}, step_num \cdot warmup_steps^{exp_2 - 1})$ which results in increasing the learning rate linearly

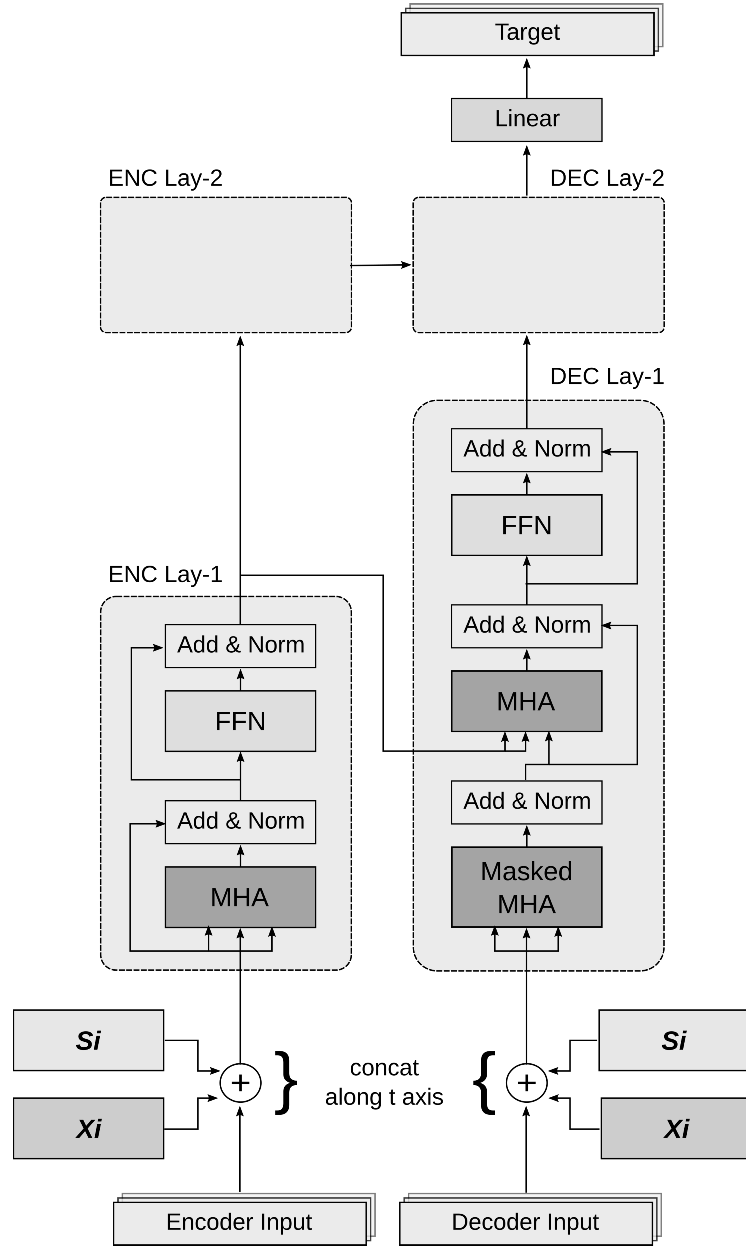
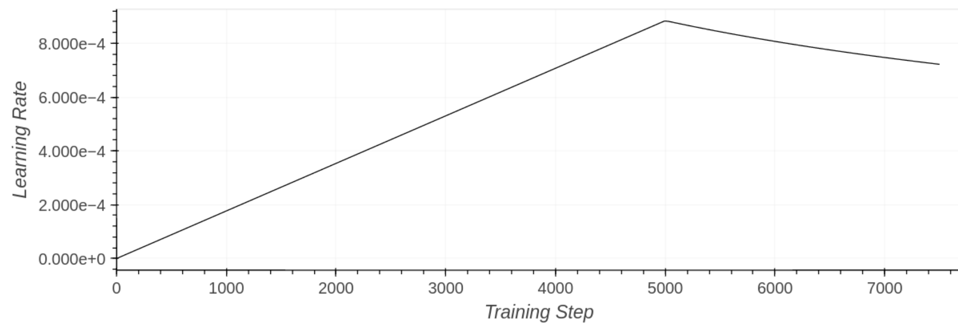


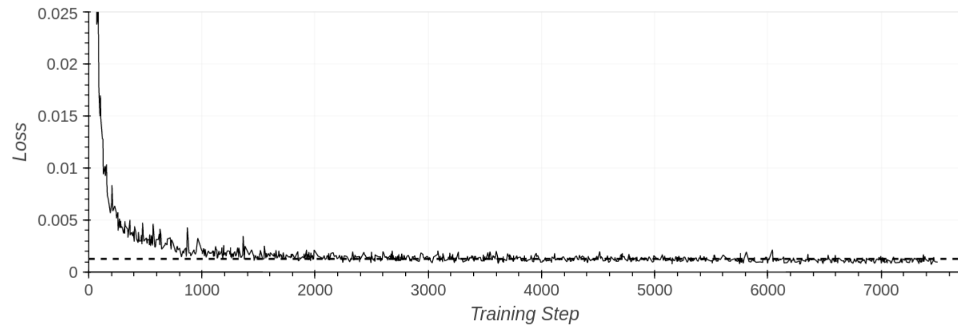
Figure 7.4: BSCTRFM Network Architecture for Global Forecasting. Adapted from Figure 2.9. The gray boxes labeled Encoder Layer 2 and Decoder Layer 2 have identical internal structures as their corresponding downstream layers.

Table 7.1: Dataset statistics and BSCTRFM parameters.

	`electricity`	`traffic`
dataset statistics		
# of time series	370	963
time granularity	hour	hour
domain	$\mathbb{R}^+$	$[0, 1]$
architecture parameters		
encoder length	168	168
decoder length	168	168
input embedding dimension	370	963
output embedding dimension	24	24
# of layers	2	2
# of heads	4	4
model dimension	256	256
feed-forward dimension	512	512
# of trainable parameters	2.7 M	2.7 M
training parameters		
# of training samples	1.8 M	3.1 M
batch size	[128, 256, 512, 1024]	[128, 256, 512, 1024]
warmup steps	[1250, 2500, 5000, 10000]	[1250, 2500, 5000, 10000]
# of epochs	1	1
testing parameters		
forecasting horizon	[24, 168]	[24, 168]

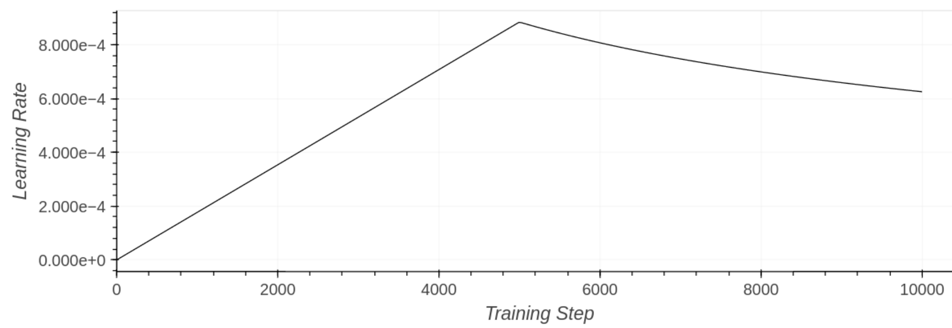


(a) Learning rate schedule.

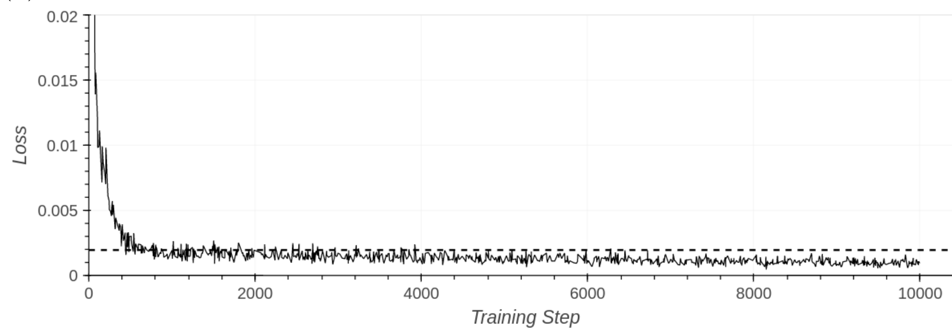


(b) Training loss.

Figure 7.5: BSCTRFM learning rate schedule and training loss on the `electricity` dataset.



(a) Learning rate schedule.



(b) Training loss.

Figure 7.6: BSCTRFM learning rate schedule and training loss on the `traffic` dataset.

for the first *warmup_steps* training steps, and decreasing it thereafter proportionally to the inverse of the $\frac{1}{exp_2}$ root of the step number [Vaswani et al., 2017]. The resulting learning rate schedules are shown in the top part of Figure 7.5 for `electricity` and Figure 7.6 for `traffic`. The MSE was used as the training and evaluation loss function. The RMSE was used as an additional custom metric, via the `metric_fn` component of the TPUEstimator API.

BSCTRFM's main architecture parameters were fixed, to keep a known number of trainable parameters as low as possible, then a grid search was used to tune a couple of training hyperparameters: learning rate and training batch size. As previously stated, the learning rate is given by a schedule that is a function of the number of warmup steps, the model dimension, and the current training step, then the number of warmup steps was the variable that defined the grid search. The number of epochs was fixed to 1 by modifying the batch size in accordance with the total training steps, approximately $\times 1.5$ the warmup steps for `electricity`, and $\times 2$ the warmup steps for `traffic`.

BSCTRFM was trained 10 different times, or executions, for each parameter set. The bottom parts of Figure 7.5 and Figure 7.6 show the behavior of the main loss metric (MSE) for `electricity` and `traffic`, respectively, over the training steps for one of the executions of the selected parametrization. The dotted line shows the value of the error metric for the model evaluation at the end of the training stage. In order to achieve high efficiency, training a model on TPUs is not compatible with frequent evaluation, as it regularly happens with CPU-based training. The reason is starting the evaluation stage requires completely stopping the training loop, including shut down the TPU. As the time required to shut down and restart the TPU is usually higher than the time consumed by one training loop, it is a good practice to schedule evaluation, whenever possible, only at the check-pointing time, when the TPU is required to shut down in order to save the model training progress.

7.2.4 Inference

After training BSCTRFM, it was tested using 24 hours (one day) and 168 hours (one week) as forecasting horizons. To exactly match the specific inference intervals reported by the reference models, 7 rolling-day predictions of one day ahead were produced on each dataset. Following the convention in literature, I labeled these inference results as $\texttt{electricity}_{1d}$ and $\texttt{traffic}_{1d}$.

In order to study the predictive performance of BSCTRFM on a longer horizon, I also produced one-time predictions seven days ahead for each dataset, and labeled them as $\texttt{electricity}_{7d}$ and $\texttt{traffic}_{7d}$. Normalized Deviation (ND) and Normalized Root Mean Squared Error (NRMSE) were used as metrics for testing the forecasting accuracy of the model, and are defined as follows:

$$ND = \frac{\sum_{(i,t)\in\Omega_{test}} |\hat{z}_{i,t} - z_{i,t}|}{\sum_{(i,t)\in\Omega_{test}} |z_{i,t}|} \tag{7.2}$$

$$NRMSE = \frac{\sqrt{\frac{1}{|\Omega_{test}|} \sum_{(i,t)\in\Omega_{test}} |\hat{z}_{i,t} - z_{i,t}|^2}}{\frac{1}{|\Omega_{test}|} \sum_{(i,t)\in\Omega_{test}} |z_{i,t}|} \tag{7.3}$$

where Ω_{test} is the collection of readings for all t timesteps in all i time series in the test dataset, and $|\Omega_{test}|$ is the number of readings in that collection.

7.3 Results and Discussion

This section presents the results of experiments with BSCTRFM on global forecasting. First, I introduce the benchmark for assessing the model's performance. Next, forecasting accuracy and computation times obtained are compared with the models in the benchmark.

7.3.1 Benchmark Models

To assess BSCTRFM's performance on the global, multi-horizon TSF problem, experimental results obtained by state-of-the-art models on the selected datasets, inference setups, and error metrics were collected from the literature. Following is the description of the benchmark models (Transformer-based models have already been described in Section 6.1):

1. `AST` [Wu et al., 2020b],
2. `TRMF` [Yu et al., 2016] is a temporal regularized matrix factorization framework for high-dimensional time series problems (including TSF) with missing values,

3. `DeepAR` [Salinas et al., 2020] is an autoregressive model for probabilistic forecasting with an RNN-based architecture (this model represents a useful comparison for BSCTRFM because it is the only one that explicitly provides computation times),

4. `LogSparse` Transformer [Li et al., 2019b],

5. `ARIMA` [Box and Jenkins, 1968] (see Subsection 2.2.2) and

6. `ETS` [Gardner Jr, 1985] (see Subsection 2.2.1) are two of the most prominent statistical methods applied to TSF,

7. `DeepState` [Rangapuram et al., 2018] is an approach to probabilistic TSF that integrates a linear state space model with a jointly-learned RNN,

8. `Informer` [Zhou et al., 2021],

9. `TFT` [Lim et al., 2021],

10. `Seq2Seq` is the simple Seq2Seq [Sutskever et al., 2014] TSF model with a global context, and

11. `MQRNN` [Wen et al., 2017] is a framework for probabilistic TSF that combines Seq2Seq neural networks, quantile regression, and multi-horizon direct forecasting.

7.3.2 Forecasting Accuracy

Table 7.2 shows the forecasting accuracy comparison of BSCTRFM with the benchmark models. Most of these models report ND values for $\texttt{electricity}_{1d}$, $\texttt{electricity}_{7d}$, $\texttt{traffic}_{1d}$, and $\texttt{traffic}_{7d}$. NRMSE values are only reported by `DeepAR` and `TRMF` for $\texttt{electricity}_{1d}$ and $\texttt{traffic}_{1d}$. `Informer` is evaluated on `electricity` using a very particular procedure: it takes a two-year, hourly-resolution dataset and splits it into train/val/test sets of 15/3/4 months, and although the model is global, only the MT-320 power consumption time series is used for testing. This evaluation is atypical, so BSCTRFM was not directly compared with `Informer`. However, the comparison that `Informer` established with `DeepAR`, `LogSparse` and `ARIMA` (MAE and MSE) is presented as a valuable reference.

Table 7.2: Forecasting accuracy of BSCTRFM compared to state-of-the-art models. The last row shows how it ranks among the models that provide a metric value for a given combination of dataset and inference type.

Dataset	`electricity`					`traffic`		
Inference	1d	1d	7d	7d	7d	1d	1d	7d
Metric	ND	NRMSE	ND	MAE	MSE	ND	NRMSE	ND
AST	**0.042**[1]	-	**0.057**[1]	-	-	**0.093**[1]	-	**0.125**[1]
TRMF	0.084[3]	1.150[2]	0.087[3]	-	-	0.186[3]	0.430[2]	0.202[3]
DeepAR	0.075[3]	1.000[2]	0.082[3]	**0.398**[4]	**0.295**[4]	0.161[3]	**0.420**[2]	0.179[3]
LogSparse	0.059[3]	-	0.070[3]	0.481[4]	0.410[4]	0.122[3]	-	0.139[3]
ARIMA	0.154[3]	-	0.283[3]	1.014[4]	0.797[4]	0.223[3]	-	0.492[3]
ETS	0.101[3]	-	0.121[3]	-	-	0.236[3]	-	0.509[3]
DeepState	0.083[3]	-	0.085[3]	-	-	0.167[3]	-	0.168[3]
Informer	-	-	-	0.466[4]	0.408[4]	-	-	-
TFT	0.055[5]	-	-	-	-	0.095[5]	-	-
Seq2Seq	0.067[5]	-	-	-	-	0.105[5]	-	-
MQRNN	0.077[5]	-	-	-	-	0.117[5]	-	-
BSCTRFM	0.070	**0.546**	0.082	-	-	0.167	0.422	0.203
Rank	5/11	1/3	3/8	-	-	7/11	2/3	6/8

[1]Results from [Wu et al., 2020b].
[2]Results from [Salinas et al., 2020].
[3]Results from [Li et al., 2019b].
[4]Results from [Zhou et al., 2021].
[5]Results from [Lim et al., 2021].

BSCTRFM consistently reaches convergence at a fast pace, after training it for roughly one epoch on the complete datasets, that is 1.8 million example rows for `electricity` and 3.1 million example rows for `traffic`. On $\texttt{electricity}_{1d}$, BSCTRFM ranks fifth among eleven models for ND, and ranks first among the three available models for NRMSE. Concerning $\texttt{traffic}_{1d}$, BSCTRFM ranks seventh among eleven models for ND, and second of three for NRMSE. The fact that BSCTRFM achieves good predictive performance on `traffic` even though the dataset was not entirely passed to the model during training, supports the assumption that it can learn patterns across time series that exhibit similar behavior. Regarding the seven-day forecasting horizon, with only ND metrics available, BSCTRFM ranks third of eight for `electricity` and sixth of eight for `traffic`. This predictive performance degradation on $\texttt{traffic}_{7d}$ can be explained by the error accumulation in the iterative inference process, and suggests the need for training for more epochs if longer forecasting horizons are planned for this dataset.

7.3.3 Computation Time

The number of trainable parameters in the selected configuration of BSCTRFM is approximately 2.7 million, which makes it a reasonably light architecture, regardless of the quadratic complexity in its complete self-attention components. The lightweight condition of the network architecture in conjunction with the massive capacity of TPUs for matrix multiplication and accumulation operations explains the remarkable speed BSCTRFM reaches during the training stage.

Concerning computation time, BSCTRFM outperforms all the models in the benchmark, as shown in Table 7.3. BSCTRFM takes only 89 seconds of training wall-time[10] on `electricity` and 119 seconds on `traffic` to consistently reach the accuracy metrics reported above. This represents an outstanding improvement over the 7 hours and 3 hours of total running time reported by `DeepAR`, respectively. The extreme reduction in training computation time achieved by BSCTRFM does not exclusively apply to `DeepAR`, a DL architecture that exhibits comparable predictive performance to BSCTRFM. The `Temporal Fusion Transformer` (TFT) [Lim et al., 2021] reported a training computation time on GPU-accelerated hardware, that is slightly

[10]Elapsed real time, or wall-clock time is the time taken to complete the training process. Training wall time is reported by TensorBoard and does not include the time spent on evaluation or checkpoint-related operations.

Table 7.3: Computation time comparison.

Dataset	DeepAR	TFT	LogSparse	Informer	BSCTRFM
`electricity`	420 min[1]	360 min[2]	120 min[3]	120 min[3]	**1.5 min**
`traffic`	180 min[1]	-	-	-	**2.0 min**

[1]Results from [Salinas et al., 2020]. Reported as total running time, it includes training and inference stages. Trained on a single p2.xlarge AWS EC2 compute instance containing 4 CPUs and 1 GPU (not specified).

[2]Results from [Lim et al., 2021]. Trained using a single NVIDIA Tesla V100 GPU.

[3]Results from [Zhou et al., 2021]. A total training time of 480 minutes (on a not specified number of NVIDIA Tesla V100 SXM2 GPUs with 32GB memory) is reported for this dataset and three similar datasets when the encoder length is set to 168, as in my main experiment.

over 360 minutes for the `electricity` dataset. In the same line, training times close to 480 minutes are reported for both `LogSparse Transformer` and `Informer` in [Zhou et al., 2021], when these models are trained with the encoder length set to 168 (as in my main experiment) on `electricity` and three similar datasets using GPU-accelerated hardware. Inference computation times for BSCTRFM are highly dependent on the infrastructure where prediction is served and, due to the iterative nature of the model, this inference process is not suitable for TPU-based implementation. Instead, a parallel execution based on separated time series processing was prototyped to obtain inference computation times comparable in magnitude to those required by training.

Summary

This Chapter presents in detail the implementation of BSCTRFM for the case of global, multi-horizon TSF. BSCTRFM was tested on two standard and predominant datasets referred to in the literature as `electricity` and `traffic`. The benchmark used for evaluating BSCTRFM performance is a collection of state-of-the-art models that, in recent years, have reported experiments on the datasets mentioned above with equivalent inference procedures and error metrics.

Some important results from the experimental study are:

- When compared to state-of-the-art models using standard inference procedures and error metrics, BSCTRFM achieves rankings that go from $1^{\text{st}}/3$ for NRMSE on `electricity`$_{1d}$, to $7^{\text{th}}/11$ for ND on `traffic`$_{1d}$. This supports the good predictive performance of my implementation among contemporary, state-of-the-art TSF solutions.
- BSCTRFM consistently reaches the reported forecasting accuracy after training it for just 1.1 epochs on `electricity` and 0.9 epochs on `traffic`.
- The selected configuration of BSCTRFM was consistently trained in 89 seconds of wall-time for `electricity` and in 119 seconds of wall-time for `traffic`.
- The models in the benchmark that reported speed metrics require computation times in the range of $\times 80$ to $\times 280$ the training time required by BSCTRFM on `electricity`, and $\times 90$ the training time required by BSCTRFM on `traffic`. This supports the fact that this implementation is the fastest TSF solution available for standard experiments on the selected datasets.

The next Chapter presents the conclusions of this Thesis, as well as important research lines for future investigation.

Chapter 8

Conclusions and Future Work

This Chapter presents the conclusions of the Thesis, summarizes its contributions, and points out directions for future work.

8.1 Conclusions

The aim of this Thesis was to improve the performance of proven ML-based forecasting models at increased speed, to make them a suitable option for TSF in Big Data environments. In order to comply with the best practices for using TPU accelerators, I designed and implemented a robust cloud-computing platform and integrated it with an advanced, production-level software stack. This integration allowed selected TSF models to be trained in an *all-reduce*, synchronous, data-parallel distributed computing strategy.

I leveraged this cloud-based, distributed ML implementation to investigate three contemporary DL-based models for point, univariate, multi-horizon TSF. On this basis, I tested the three models for local forecasting on an extensive electric power distribution dataset. I also tested one of the models for global forecasting on two standardized datasets that are predominantly used for benchmarking in TSF research.

I investigated three model architectures for local forecasting: the Deep Multi-sequence Long Short-term Memory DMSLSTM, the Encoder-Decoder Stacked LSTM with attention EDSLSTM, and the Basic Transformer BSC-TRFM. These models were evaluated against a robust, cloud-based ARIMA implementation that is part of BigQuery, the data warehousing service of Google Cloud. BigQuery ARIMA was parametrized to keep its training

times under 4 seconds, nevertheless, that configuration was enough for the system to fit and evaluate up to 42 forecasting models in parallel, then select the best one as the baseline. On this basis, I tested the models for short-term forecasting accuracy (24-hour ahead, daily predictions over a week), training computation times, and predictive performance degradation (rolling-day predictions over four weeks and rolling-hour predictions over four weeks). Both DMSLSTM and EDSLSTM overperformed the baseline model on all of the tests. BSCTRFM suffered from instability and overperformed the baseline only in the short-term forecasting accuracy and training computation time tests. I was able to train the final configuration of DMSLSTM on the same computing infrastructure without TPU acceleration. That experiment reported a training time $\times 99$ the training time of the TPU-accelerated experiment. Conversely, I was not able to train the final configurations for EDSLSTM or BSCTRFM without using TPUs. I opted for training these models for up to one epoch and then projecting those results to the original experiment extensions. I calculated training times of these models without TPU as $\times 369$ and $\times 968$ the time required by the cloud-based, DML implementation. In practical terms, this is equivalent to saying these architectures are impossible to train, for the selected dataset and parametrization, without hardware accelerators.

I investigated BSCTRFM also for global forecasting. The model was evaluated against eleven models in the literature, including those that report the state-of-the-art. BSCTRFM achieved forecasting accuracy rankings from first to seventh on the available combinations of inference type and loss metric for the selected datasets. At the same time, it overperformed all of the models in the benchmark that reported speed metrics by achieving training computation times under two minutes.

8.2 Contributions

Following I present the contributions of this Thesis.

Cloud Computing

I designed, implemented, and tested a complete cloud-computing platform in accordance with the best practices for the operation of TPU accelerators. This platform integrates standard deployments of cloud-based storage,

networking, computing, and hardware-based acceleration services.

Production-level Software Stack

Following the best practices for the operation of TPUs, I implemented all of the ML models using the TensorFlow 1.1X framework alongside its TPU-specific Estimator API. This is a software stack with a very steep learning curve that provides a basis for day-to-day, production-level ML operations at large-size companies such as Google, its original developer, and main contributor. Even though this is not a scientific contribution, it is indispensable for producing the experiments and allows scaling further stages of the implementation to real-life, production scopes that are far beyond those achieved by most research projects.

Contemporary ML Architectures

This research leveraged the cloud-based platform and the software stack mentioned above to implement three ML architectures characterized by a sizable number of trainable parameters. Due to their state-of-the-art results, these architectures account for most of the recent research on TSF globally.

First TPU-based TSF Transformer

To the extent of my knowledge, this research proposes the first TPU-based implementation of a canonical (multi-layer, encoder-decoder, full self-attention, dense-matrix computation) Transformer-based architecture for TSF.

Fastest TSF Transformer in the Literature

The Transformer-based architecture mentioned above reports the lowest training times found in the literature on two predominant, standardized datasets for global, multi-horizon TSF. At the same time, it provides good predictive performance on those datasets when compared to state-of-the-art models.

Open Source-based Development

Even though the low-level libraries for using TPU accelerators are Google-proprietary, my complete implementation is based on open-source software

and is available to other research teams upon approved request.

8.3 Future Work

Directives for future work on this research are grouped into two categories: actions concerning the improvement of the implemented model architectures, and actions related to integrating this research work into ML pipelines.

Model Architecture Improvement

The primary objective of this research was not to develop TSF models with the best possible accuracy metrics, but to implement architectures already proven in the literature on a robust DML framework so that they could rival strong baselines with drastic reductions in computation times. Nevertheless, I am now in possession of an ultra-fast cloud-computing framework that can be leveraged for further improvement of the TSF models studied in this research, as well as other architectures not revised here.

The DMSLSTM network can be improved by increasing the length of the context and forecasting windows (up to 2- or even 3-digit figures) to better capture data dependencies in the longer term, which can prevent predictive performance degradation. Additionally, an age-covariate can be used along with the one-hot timestamp encoding to better align the context and forecasting windows on longer training datasets.

The EDSLSTM network can be improved by integrating different types of attention on a weighted basis, as only Luong's multiplicative type was used in this stage. In addition, a configurable inter-layer wiring layout can be used to produce and evaluate context vectors at different depth levels of the encoder and the decoder.

Needless to say, both DMSLSTM and EDSLSTM can be extended by adding a global forecasting functionality, which was not implemented in this Thesis to avoid the increased training times that would be generated by their multiple, sequential-calculation LSTM layers.

The BSCTRFM architecture for local forecasting can be improved by adding functionalities aimed to stabilize its training process, such as learning rate schedule optimization, adaptive model initialization [Liu et al., 2020], or modified residual connections [Parisotto et al., 2020].

The BSCTRFM architecture for global forecasting can be improved by adding sparsity and enhanced locality to its self-attention mechanism via convolutional filters [Li et al., 2019b], different activation functions [Wu et al., 2020b], or sequence-downsampling inside self-attention layers [Zhou et al., 2021]. These modifications have already proved to increase the Transformer's accuracy on TSF problems, therefore they constitute an excellent way to evaluate the impact that moving the TPU-accelerated Transformer outside the dense-matrix-computation domain will have on its speed.

Integration into ML Pipelines

Since the beginning of this Thesis, I opted for developing the ML models on a production-level software stack comprised by the TensorFlow 1.1X numerical computing framework and its TPU-specific Estimator API. I selected this coding solution over the typical, notebook-based, interactive environments because of its unique capability to manage mid- to high-level operations that are specific to TPU accelerators, like the fundamental decoupling of `input`, `model`, and `predict` functions. This software stack is recognized to have a very steep learning curve, which hinders its application in academic research institutions, that mostly opt in favor of easier, nevertheless robust, tools such as Keras or PyTorch.

Besides the more flexible TPU management it provides, another significant advantage of this software stack is a straightforward integration of ML artifacts, such as datasets, models, hyperparameter configurations, and metric results, into dedicated production-level pipelines. The reason behind this advantage is that TensorFlow 1.1X is still the only computing framework fully supported for training tasks inside TensorFlow Extended (TFX) [Baylor et al., 2017], the production-scale ML platform used at Google.

ML pipelines enable research institutions to efficiently manage the complex set of operations an ML model is involved with, at scale. ML production faces all of the challenges in traditional software development (testing, code review, monitoring, etc.) as well as new problems that are specific to its strong dependency on massive, dynamic data. Among these challenges, [Zaharia et al., 2018] identify the following:

- multiple software tools are available at each stage of ML development and the resulting combinations are difficult to register and compare,

- tracking the lineage of a given experiment becomes hard as many parametrization variants lie on top of the software stack combinations,
- reproducibility, a significant requirement of ML development, strongly depends on effective experiment tracking, and
- production-level deployment of ML solutions is required (even in academic scopes) in order to schedule and execute a huge number of experiments, even by people that did not code the application.

Moving this research work into a dedicated ML pipelining environment would amplify its scientific impact by allowing us to extensively experiment on a significant number of combinations, while efficiently recording and comparing all of the resulting variants. It also would allow us to better share artifacts like datasets, preprocessing directives, model architectures, hyperparameters, and evaluation procedures with other research teams. This situation could facilitate collaborations with important universities abroad that have already adopted these development standards in their research departments. Lastly, the expertise produced by leveraging production-level ML environments in this research would greatly contribute to closing the gap between academic and industrial operations in our professional context.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

Mohamed Abdel-Nasser and Karar Mahmoud. Accurate photovoltaic power forecasting models using deep LSTM-RNN. *Neural Computing and Applications*, 31(7):2727–2740, 2019.

Ismail Alaoui Abdellaoui and Siamak Mehrkanoon. Deep multi-stations weather forecasting: explainable recurrent convolutional neural networks, 2020. URL `https://arxiv.org/abs/2009.11239`.

Ismail Alaoui Abdellaoui and Siamak Mehrkanoon. Symbolic regression for scientific discovery: an application to wind speed forecasting. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 01–08. IEEE, 2021.

Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

Sultan Alrowili and K Vijay-Shanker. BioM-transformers: building large biomedical language models with BERT, ALBERT and ELECTRA. In *Proceedings of the 20th Workshop on Biomedical Language Processing*, pages 221–227, 2021.

Berkay Aydin. Parallel algorithms on nearest neighbor search. *Survey paper, Georgia State University*, 2014.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014. URL `https://arxiv.org/abs/1409.0473`.

Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

Filippo Maria Bianchi, Enrico Maiorino, Michael C Kampffmeyer, Antonello Rizzi, and Robert Jenssen. *Recurrent Neural Networks for Short-Term Load Forecasting: An Overview and Comparative Analysis*. Springer, 2017.

Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. Machine learning strategies for time series forecasting. In *European business intelligence summer school*, pages 62–77. Springer, 2012.

Salah Bouktif, Ali Fiaz, Ali Ouni, and Mohamed Adel Serhani. Optimal deep learning LSTM model for electric load forecasting using feature selection and genetic algorithm: Comparison with machine learning approaches. *Energies*, 11(7):1636, 2018.

George EP Box and Gwilym M Jenkins. Some recent advances in forecasting and control. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 17(2):91–109, 1968.

George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. Time series analysis: Forecasting and control. 2008.

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

Robert G Brown and Richard F Meyer. The fundamental theorem of exponential smoothing. *Operations Research*, 9(5):673–685, 1961.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mengmeng Cai, Manisa Pipattanasomporn, and Saifur Rahman. Day-ahead building-level load forecasts using deep learning vs. traditional time-series techniques. *Applied energy*, 236:1078–1088, 2019.

Karanbir Singh Chahal, Manraj Singh Grover, Kuntal Dey, and Rajiv Ratn Shah. A hitchhiker's guide on distributed training of deep neural networks. *Journal of Parallel and Distributed Computing*, 137:65–76, 2020.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015. URL `https://arxiv.org/abs/1512.01274`.

Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D Sculley, Jamie Smith, David Soergel, et al. TensorFlow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1763–1771, 2017a.

Shou-Hsiung Cheng, Shyi-Ming Chen, and Wen-Shan Jian. Fuzzy time series forecasting based on fuzzy logical relationships and similarity measures. *Information Sciences*, 327:272–287, 2016.

Yao Cheng, Chang Xu, Daisuke Mashima, Vrizlynn LL Thing, and Yongdong Wu. PowerLSTM: Power demand forecasting using long short-term memory neural network. In *International Conference on Advanced Data Mining and Applications*, pages 727–740. Springer, 2017b.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Yagmur Gizem Cinar, Hamid Mirisaee, Parantapa Goswami, Eric Gaussier, Ali Aït-Bachir, and Vadim Strijov. Position-based content attention for

time series forecasting with sequence-to-sequence rnns. In *International conference on neural information processing*, pages 533–544. Springer, 2017a.

Yagmur Gizem Cinar, Hamid Mirisaee, Parantapa Goswami, Eric Gaussier, Ali Ait-Bachir, and Vadim Strijov. Time series forecasting using rnns: an extended attention mechanism to model periods and handle missing values. *arXiv preprint arXiv:1703.10089*, 2017b.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

A Shobana Devi, G Maragatham, K Boopathi, and AG Rangaraj. Hourly day-ahead wind power forecasting with the EEMD-CSO-LSTM-EFG deep learning technique. *Soft Computing*, pages 1–21, 2020.

Shyam Kumar Doddavula, Ira Agrawal, and Vikas Saxena. Cloud computing solution patterns: Infrastructural solutions. In *Cloud Computing*, pages 197–219. Springer, 2013.

Shengdong Du, Tianrui Li, Yan Yang, and Shi-Jinn Horng. Multivariate time series forecasting via attention-based encoder–decoder framework. *Neurocomputing*, 388:269–279, 2020.

Adrian Eisenmann, Tim Streubel, and Krzysztof Rudion. Power quality prediction by way of parallel computing-a new approach based on a long short-term memory network. In *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, pages 1–5. IEEE, 2019.

Chenyou Fan, Yuze Zhang, Yi Pan, Xiaoyue Li, Chi Zhang, Rong Yuan, Di Wu, Wensheng Wang, Jian Pei, and Heng Huang. Multi-horizon time series forecasting with temporal attention learning. In *Proceedings of the 25th ACM SIGKDD International conference on knowledge discovery & data mining*, pages 2527–2535, 2019.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

Joao Canas Ferreira and Jose Fonseca. An fpga implementation of a long short-term memory neural network. In *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*, pages 1–8. IEEE, 2016.

Juan J Flores, Jose L Garcia-Nava, Jose R Cedeno Gonzalez, Victor M Tellez, Felix Calderon, and Arturo Medrano. A machine-learning pipeline for large-scale power-quality forecasting in the mexican distribution grid. *Applied Sciences*, 12(17):8423, 2022.

Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

Alex Galakatos, Andrew Crotty, and Tim Kraska. Distributed machine learning. *Encyclopedia of Database Systems*, 2017.

Everette S Gardner Jr. Exponential smoothing: The state of the art. *Journal of forecasting*, 4(1):1–28, 1985.

Everette S Gardner Jr. Exponential smoothing: The state of the art—part ii. *International journal of forecasting*, 22(4):637–666, 2006.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014. URL `https://arxiv.org/abs/1410.5401`.

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

Guang Gu and James L Anderson. Deseasonalized state-space time series forecasting with application to the us salmon market. *Marine Resource Economics*, 10(2):171–185, 1995.

Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 629–634. IEEE, 2017.

Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. Attention-based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 922–929, 2019.

Ameen Abdel Hai and Babak Forouraghi. On scalability of distributed machine learning with big data on apache spark. In *International Conference on Big Data*, pages 209–219. Springer, 2018.

Markus Hauru, Alan Morningstar, Jackson Beall, Martin Ganahl, Adam Lewis, and Guifre Vidal. Simulation of quantum physics with tensor processing units: brute-force computation of ground states and time evolution, 2021. URL `https://arxiv.org/abs/2111.10466`.

Pradeep Hewage, Ardhendu Behera, Marcello Trovati, Ella Pereira, Morteza Ghahremani, Francesco Palmieri, and Yonghuai Liu. Temporal convolutional neural (tcn) network for an effective weather forecasting using time-series data from the local weather station. *Soft Computing*, 24(21): 16453–16482, 2020.

Hansika Hewamalage, Christoph Bergmeir, and Kasun Bandara. Global models for time series forecasting: A simulation study. *Pattern Recognition*, page 108441, 2021.

Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

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

Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12): 58–65, 2021.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

Tim Januschowski, Jan Gasthaus, Yuyang Wang, David Salinas, Valentin Flunkert, Michael Bohlke-Schneider, and Laurent Callot. Criteria for classifying forecasting methods. *International Journal of Forecasting*, 36(1): 167–177, 2020.

Tim Januschowski, Yuyang Wang, Kari Torkkola, Timo Erkkilä, Hilaf Hasson, and Jan Gasthaus. Forecasting with trees. *International Journal of Forecasting*, 2021.

Runhai Jiao, Tianming Zhang, Yizhi Jiang, and Hui He. Short-term non-residential load forecasting based on multiple sequences LSTM recurrent neural network. *IEEE Access*, 6:59438–59448, 2018.

Ameet V Joshi. Amazon's machine learning toolkit: Sagemaker. In *Machine learning and artificial intelligence*, pages 233–243. Springer, 2020.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a Tensor Processing Unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Fpga-accelerated dense linear machine learning: A precision-convergence

trade-off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167. IEEE, 2017.

Jayashree Katti, Jony Agarwal, Swapnil Bharata, Swati Shinde, Saral Mane, and Vinod Biradar. University admission prediction using google vertex ai. In *2022 First International Conference on Artificial Intelligence Trends and Pattern Recognition (ICAITPR)*, pages 1–5, 2022. doi: 10.1109/ICAITPR51569.2022.9844176.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

Kang Ke, Sun Hongbin, Zhang Chengkang, and Carl Brown. Short-term electrical load forecasting method based on stacked auto-encoding and GRU neural network. *Evolutionary Intelligence*, 12(3):385–394, 2019.

Elizaveta Kharlova, Daniel May, and Petr Musilek. Forecasting photovoltaic power production using a deep learning sequence to sequence model with attention. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2020.

Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA,May 7-9, 2015, Conference Track Proceedings*, 2015.

Douglas M Kline. Methods for multi-step time series forecasting neural networks. In *Neural networks in business forecasting*, pages 226–250. IGI Global, 2004.

Weicong Kong, Zhao Yang Dong, David J Hill, Fengji Luo, and Yan Xu. Short-term residential load forecasting based on resident behaviour learning. *IEEE Transactions on Power Systems*, 33(1):1087–1088, 2017a.

Weicong Kong, Zhao Yang Dong, Youwei Jia, David J Hill, Yan Xu, and Yuan Zhang. Short-term residential load forecasting based on LSTM recurrent neural network. *IEEE Transactions on Smart Grid*, 10(1):841–851, 2017b.

Sumit Kumar, Lasani Hussain, Sekhar Banarjee, and Motahar Reza. Energy load forecasting using deep learning approach-LSTM and GRU in Spark cluster. In *2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT)*, pages 1–4. IEEE, 2018.

HT Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics, 1979.

Matthias Langer, Zhen He, Wenny Rahayu, and Yanbo Xue. Distributed training of deep learning models: A taxonomic perspective. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2802–2818, 2020.

Pedro Lara-Benítez, Manuel Carranza-García, José M Luna-Romera, and José C Riquelme. Temporal convolutional networks applied to energy-related time series forecasting. *applied sciences*, 10(7):2322, 2020.

Gangqiang Li, Huaizhi Wang, Shengli Zhang, Jiantao Xin, and Huichuan Liu. Recurrent neural networks based photovoltaic power forecasting approach. *Energies*, 12(13):2538, 2019a.

Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in Neural Information Processing Systems*, 32:5243–5253, 2019b.

Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A*, 379(2194): 20200209, 2021.

Bryan Lim, Sercan Ö Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 2021.

Chang Liu, Zhijian Jin, Jie Gu, and Caiming Qiu. Short-term load forecasting using a long short-term memory network. In *Innovative Smart Grid*

Technologies Conference Europe (ISGT-Europe), 2017 IEEE PES, pages 1–6. IEEE, 2017.

Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. *arXiv preprint arXiv:2004.08249*, 2020.

Chi-Jie Lu, Tian-Shyug Lee, and Chih-Chou Chiu. Financial time series forecasting using independent component analysis and support vector regression. *Decision support systems*, 47(2):115–125, 2009.

Xiaoyi Lu, Haiyang Shi, Rajarshi Biswas, M. Haseeb Javed, and Dhabaleswar K. Panda. Dlobd: A comprehensive study of deep learning over big data stacks on hpc clusters. *IEEE Transactions on Multi-Scale Computing Systems*, 4(4):635–648, 2018. doi: 10.1109/TMSCS.2018.2845886.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015. URL `https://arxiv.org/abs/1508.04025`.

Peng Lv and Lili Yue. Short-term wind speed forecasting based on non-stationary time series analysis and arch model. In *2011 International Conference on Multimedia Technology*, pages 2549–2553. IEEE, 2011.

Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36(1):54–74, 2020.

Daniel L Marino, Kasun Amarasinghe, and Milos Manic. Building energy load forecasting using deep neural networks. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 7046–7051. IEEE, 2016.

Francisco Martínez, María Pilar Frías, María Dolores Pérez, and Antonio Jesús Rivera. A methodology for applying k-nearest neighbor to time series forecasting. *Artificial Intelligence Review*, 52(3), 2019.

Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

Usha Manasi Mohapatra, Babita Majhi, and Suresh Chandra Satapathy. Financial time series prediction using distributed machine learning techniques. *Neural Computing and Applications*, 31(8):3369–3384, 2019.

Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. Sparknet: Training deep networks in spark, 2015. URL `https://arxiv.org/abs/1511.06051`.

Sana Mujeeb, Nadeem Javaid, Mariam Akbar, Rabiya Khalid, Orooj Nazeer, and Mahnoor Khan. Big data analytics for price and load forecasting in smart grids. In *International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 77–87. Springer, 2018.

Chris Olah and Shan Carter. Attention and augmented recurrent neural networks. *Distill*, 2016. doi: 10.23915/distill.00001. URL `http://distill.pub/2016/augmented-rnns`.

Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11): 1–4, 2015.

Zhixin Pan and Prabhat Mishra. Hardware acceleration of explainable machine learning. In *2022 Design, Automation And Test in Europe Conference And Exhibition DATE*, pages 1127–1130, 2022. doi: 10.23919/DATE54114.2022.9774739.

Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training, 2021. URL `https://arxiv.org/abs/2104.10350`.

C Carl Pegels. Exponential forecasting: Some new variations. *Management Science*, pages 311–315, 1969.

Lu Peng, Qing Zhu, Sheng-Xiang Lv, and Lin Wang. Effective long short-term memory with fruit fly optimization algorithm for time series forecasting. *Soft Computing*, pages 1–21, 2020.

Diego Peteiro-Barral and Bertha Guijarro-Berdiñas. A survey of methods for distributed machine learning. *Progress in Artificial Intelligence*, 2(1): 1–11, 2013.

Fotios Petropoulos, Daniele Apiletti, Vassilios Assimakopoulos, Mohamed Zied Babai, Devon K Barrow, Souhaib Ben Taieb, Christoph Bergmeir, Ricardo J Bessa, Jakub Bijak, John E Boylan, et al. Forecasting: theory and practice. *International Journal of Forecasting*, 2022.

Daniel Pop, Gabriel Iuhasz, and Dana Petcu. Distributed platforms and cloud services: Enabling machine learning for big data. In *Data Science and Big Data Computing*, pages 139–159. Springer, 2016.

Martin Popel and Ondřej Bojar. Training tips for the transformer model. *arXiv preprint arXiv:1804.00247*, 2018.

Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. *Advances in neural information processing systems*, 31, 2018.

Alaa Sagheer and Mostafa Kotb. Time series forecasting of petroleum production using deep lstm recurrent networks. *Neurocomputing*, 323:203–213, 2019.

David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

Ljubisa Sehovac and Katarina Grolinger. Deep learning for load forecasting: Sequence to sequence recurrent neural networks with attention. *IEEE Access*, 8:36411–36426, 2020.

Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018. URL `https://arxiv.org/abs/1802.05799`.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

Alex Sheng and Jun Yi Derek He. Distributed evolution strategies using tpus for meta-learning. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 721–728. IEEE, 2020.

Shun-Yao Shih, Fan-Keng Sun, and Hung-yi Lee. Temporal pattern attention for multivariate time series forecasting. *Machine Learning*, 108(8):1421–1441, 2019.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

Slawek Smyl. A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, 36(1):75–85, 2020.

Antti Sorjamaa and Amaury Lendasse. Time series prediction using dirrec strategy. In *Esann*, volume 6, pages 143–148. Citeseer, 2006.

Aravind Srinivas, Tsung-Yi Lin, Niki Parmar, Jonathon Shlens, Pieter Abbeel, and Ashish Vaswani. Bottleneck transformers for visual recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16519–16529, 2021.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

Souhaib Ben Taieb, Gianluca Bontempi, Antti Sorjamaa, and Amaury Lendasse. Long-term prediction of time series by combining direct and mimo strategies. In *2009 International Joint Conference on Neural Networks*, pages 3054–3061. IEEE, 2009.

Souhaib Ben Taieb, Gianluca Bontempi, Amir F Atiya, and Antti Sorjamaa. A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition. *Expert systems with applications*, 39(8):7067–7083, 2012.

AzureML Team. Azureml: Anatomy of a machine learning service. In *Conference on Predictive APIs and Apps*, pages 1–13. PMLR, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *Acm computing surveys (csur)*, 53(2):1–33, 2020.

Renzhuo Wan, Shuping Mei, Jun Wang, Min Liu, and Fan Yang. Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting. *Electronics*, 8(8):876, 2019.

Hai-Kun Wang, Ke Song, and Yi Cheng. A hybrid forecasting model based on cnn and informer for short-term wind power. *Frontiers in Energy Research*, page 1041, 2022.

Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. When edge meets learning: Adaptive

control for resource-constrained distributed machine learning. In *IEEE INFOCOM 2018-IEEE conference on computer communications*, pages 63–71. IEEE, 2018a.

Yi Wang, Dahua Gan, Mingyang Sun, Ning Zhang, Zongxiang Lu, and Chongqing Kang. Probabilistic individual load forecasting using pinball loss guided LSTM. *Applied Energy*, 235:10–20, 2019a.

Yixing Wang, Meiqin Liu, Zhejing Bao, and Senlin Zhang. Short-term load forecasting with multi-source data using gated recurrent unit neural networks. *Energies*, 11(5):1138, 2018b.

Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019b.

Ruofeng Wen, Kari Torkkola, Balakrishnan Narayanaswamy, and Dhruv Madeka. A multi-horizon quantile recurrent forecaster, 2017. URL `https://arxiv.org/abs/1711.11053`.

Billy M Williams. Multivariate vehicular traffic flow prediction: evaluation of arimax modeling. *Transportation Research Record*, 1776(1):194–200, 2001.

Billy M Williams and Lester A Hoel. Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results. *Journal of transportation engineering*, 129(6):664–672, 2003.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2019. URL `https://arxiv.org/abs/1910.03771`.

Arissa Wongpanich, Hieu Pham, James Demmel, Mingxing Tan, Quoc Le, Yang You, and Sameer Kumar. Training efficientnets at supercomputer scale: 83% imagenet top-1 accuracy in one hour. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 947–950. IEEE, 2021.

Gerald Woo, Chenghao Liu, Doyen Sahoo, Akshat Kumar, and Steven Hoi. Etsformer: Exponential smoothing transformers for time-series forecasting, 2022. URL `https://arxiv.org/abs/2202.01381`.

Neo Wu, Bradley Green, Xue Ben, and Shawn O'Banion. Deep transformer models for time series forecasting: The influenza prevalence case, 2020a. URL `https://arxiv.org/abs/2001.08317`.

Sifan Wu, Xi Xiao, Qianggang Ding, Peilin Zhao, Ying Wei, and Junzhou Huang. Adversarial sparse transformer for time series forecasting. *Advances in Neural Information Processing Systems*, 33, 2020b.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yuteng Xiao, Hongsheng Yin, Yudong Zhang, Honggang Qi, Yundong Zhang, and Zhaoyang Liu. A dual-stage attention-based conv-lstm network for spatio-temporal correlation and multivariate time series prediction. *International Journal of Intelligent Systems*, 36(5):2036–2057, 2021.

Eric P Xing, Qirong Ho, Pengtao Xie, and Dai Wei. Strategies and principles of distributed machine learning on big data. *Engineering*, 2(2):179–195, 2016.

Haoran Xu, Benjamin Van Durme, and Kenton Murray. BERT, mBERT, or BiBERT? a study on contextualized embeddings for neural machine translation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6663–6675, 2021a.

Jianlong Xu, Kun Wang, Che Lin, Lianghong Xiao, Xingshan Huang, and Yufeng Zhang. Fm-gru: a time series prediction method for water quality based on seq2seq framework. *Water*, 13(8):1031, 2021b.

Jiehui Xu, Jianmin Wang, Mingsheng Long, et al. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. *Advances in Neural Information Processing Systems*, 34, 2021c.

Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Fast deep neural network training on distributed systems and cloud tpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2449–2462, 2019.

Hsiang-Fu Yu, Nikhil Rao, and Inderjit S Dhillon. Temporal regularized matrix factorization for high-dimensional time series prediction. *Advances in neural information processing systems*, 29:847–855, 2016.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

Bing Zhang, Jhen-Long Wu, and Pei-Chann Chang. A multiple time series-based recurrent neural network for short-term load forecasting. *Soft Computing*, 22(12):4099–4112, 2018.

G Peter Zhang, B Eddy Patuwo, and Michael Y Hu. A simulation study of artificial neural networks for nonlinear time-series forecasting. *Computers & Operations Research*, 28(4):381–396, 2001.

Ge Zhang, Xiaoqing Bai, and Yuxuan Wang. Short-time multi-energy load forecasting method based on cnn-seq2seq model with attention mechanism. *Machine Learning with Applications*, 5:100064, 2021.

Liang Zhang, Gang Wang, and Georgios B Giannakis. Power system state forecasting via deep recurrent neural networks. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8092–8096. IEEE, 2019.

Sixin Zhang. Distributed stochastic optimization for deep learning (thesis), 2016. URL `https://arxiv.org/abs/1605.02216`.

WenJie Zhang, Jian Qin, Feng Mei, JunJie Fu, Bo Dai, and WenWu Yu. Short-term power load forecasting using integrated methods based on long short-term memory. *Science China Technological Sciences*, pages 1–11, 2020.

Yiwei Zhang, Chao Wang, Lei Gong, Yuntao Lu, Fan Sun, Chongchong Xu, Xi Li, and Xuehai Zhou. Implementation and optimization of the accelerator based on fpga hardware for lstm network. In *2017 IEEE international symposium on parallel and distributed processing with applications and 2017 IEEE international conference on ubiquitous computing and communications (ISPA/IUCC)*, pages 614–621. IEEE, 2017.

Huasha Zhao and John Canny. Sparse allreduce: Efficient scalable communication for power-law data, 2013. URL `https://arxiv.org/abs/1312.3020`.

Long Zhao, Zizhao Zhang, Ting Chen, Dimitris Metaxas, and Han Zhang. Improved transformer for high-resolution GANs. *Advances in Neural Information Processing Systems*, 34:18367–18380, 2021.

Huiting Zheng, Jiabin Yuan, and Long Chen. Short-term load forecasting using EMD-LSTM neural networks with a Xgboost algorithm for feature importance evaluation. *Energies*, 10(8):1168, 2017a.

Jian Zheng, Cencen Xu, Ziang Zhang, and Xiaohua Li. Electric load forecasting in smart grids using long short-term memory based recurrent neural network. In *Information Sciences and Systems (CISS), 2017 51st Annual Conference on*, pages 1–6. IEEE, 2017b.

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of AAAI*, 2021.