Trabajo de Fin de Máster
"Máster Universitario en Microelectónica:
Diseño y Aplicaciones de Sistemas
Micro/Nanométricos"

TIME SERIES FORECASTING WITH DEEP LEARNING FOR COGNITIVE-RADIO APPLICATIONS

Author: Promise I. OKORIE

Advisors: José Manuel DE LA ROSA UTRERA

Luis A. CAMUÑAS MESA

29th November 2021

"Always curious...to learn."

pio

UNIVERSIDAD DE SEVILLA

Abstract

Facultad de Física
Universidad de Sevilla

Máster Universitario en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos

Final Masters Project: Time series forecasting with Deep Learning for Cognitive-Radio Applications

by Promise I. OKORIE

We live in a world where the number of devices that are constantly communicating with each other are growing exponentially, and to keep up with that trend, new communication technologies are being developed at a higher rate than in previous decades. The consequence of all these is the increase in the shared usage of the same electromagnetic spectrum by all these devices. Cognitive Radios (CR) [1] are being proposed as a solution that allows communication systems to efficiently use the frequency spectrum, by dynamically modifying their transceiver specifications according to the information sensed from the electromagnetic environment, where they should be able to develop sensing, decision, sharing and allocation functions. A Software-defined Radio (SDR) acts as the base upon which CR technology can be implemented. Artificial Intelligence (AI) layers, embedded in CR systems can be used to optimize the management of the electromagnetic spectrum and assist the signal processing and performance of IoT nodes equipped with CR technology [2]. In the past few years, improvements on Artificial Neural Networks (ANNs) have led to their usage in trying to solve the spectrum management problem, where, for example, Long Short-term Memory networks (LSTMs), a type of Recurrent Neural Networks (RNNs) have been used in the past to predict temporal evolution of data [3] [4].

This project contributes to this topic by examining the use of several ANNs to predict spectrum occupancy in CR systems. Their performance is compared in terms of system complexity, execution time and accuracy. Five NN architectures are studied and implemented to predict channel occupancy which was envisioned as a time series forecasting/prediction problem and will be used to predict the future evolution of the radioelectric spectrum for Cognitive-Radio applications.

Acknowledgements

Getting to this point was not easy and for that I thank God for giving me the strength, wisdom, knowledge and understanding.

Thank you, Dad, Mum, Prince (Dedek), Precious (Cious-nwa), and Emma (Ogolo).

Thank you to Daniel Rodríguez and Noemi Amengual, ("*No es una carrera de 100 metros, es un maratón*"), to their wonderful family, to Juanpe, to Vanessa, to my Tía, to Olena and family and to my entire Centre Cristia de Mallorca family. The list is endless.

Thank you to Héctor Escobar and Amparo Benitez, to Vladi and Laura and their three cute minions, to David and mi prima Cris and to the rest of my Centro Cristiano El Sembrador family.

Thank you very much to my supervisors José Manuel de la Rosa and Luis A. Camuñas Mesa, for their patience and guidance, and mostly for being better human beings that I could ever have wished for.

Thank you very much to all who lent me a pen, greeted me, asked about my studies, told me a bad unfunny joke, and so on. Thank you *La banda del patio*.

Contents

List of Figures

List of Tables

Acronyms

ADC Analog-to-digital converter

AI Artificial Intelligence

ANNs Artificial Neural Networks

CNNs Convolutional Neural Networks

CR Cognitive Radios

DAC Digital-to-analog converter

DSA Dynamic Spectrum Access

FMP Final Masters Project

FSA Fixed Spectrum Allocation

IoT Internet of Things

IP Internet Protocol

LSTMs Long Short-term Memory networks

MLPs Multi-Layer Perceptrons

NNs Neural Networks

OSA Opportunistic Spectrum Access

PDR Programmable Digital Radio

PUs Primary Users

RKRL Radio Knowledge Representation Language

RL Reinforcement learning

RNNs Recurrent Neural Networks

RMSE Root Mean-Square Error

SDR Software-defined Radio

SS Spectrum Sensing

SUs Secondary Users

Dedicated to all who once believed in me.

Chapter 1

Introduction

1.1 Background and motivation

The rate at which wirelessly connected devices grow is consistently increasing and an indicator is in the number of mostly wireless devices connected to Internet Protocol (IP) networks. Figure 1.1 shows that by 2023, this number is expected to reach 29.3 billion units, up from 18.4 billion in 2018. The functionality of these devices, going from devices that were only capable of voice-only communications, as in the case of mobile phone devices, to devices that require higher data rates as a result of multimedia type applications currently required of them [5].

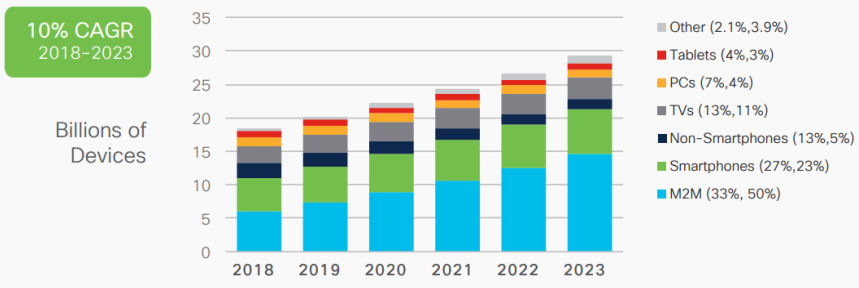


FIGURE 1.1: Global device and connection growth. (Source: *Cisco Annual Internet Report, 2018–2023*)

Also influential is the exponential growth of information traffic as a result of Internet of Things (IoT). IoT describes physical objects (or groups of such objects) that are embedded with sensors, processing ability, software, and other technologies that connect and exchange data with other devices and systems over the Internet or other communications networks [6]. As such, efficient IoT nodes will require hybrid software/hardware platforms, new computation paradigms, communication protocols, as well as highly adaptive and programmable circuitry, very specially at the analog/digital interface of the system [7], [8].

The most pressing issue in all of these is that, for all the exponential growth, both in the number of devices and in the technological innovations in communication systems, there is an excessive demand and shared use of the electromagnetic spectrum. This spectrum, shared by these devices, is both limited and controlled by regulations and recognized authorities, such as the International Telecommunications Union (ITU) for international regulations, the Federal Communications Commission (FCC) in the United States or the Ministerio de Industria, Turismo y Comercio through the Secretaría de Estado de Telecomunicaciones y para la Sociedad de la Información (SETSI) for Spain, which mostly use a Fixed Spectrum Allocation (FSA) policy. The current static allocation schemes by these regulators consist of assigning channels

to specific users with licenses for specific wireless technologies and services, where these *licensed users*, also called Primary Users (PUs), have access to their assigned spectrum portions to transmit/receive their data, while others, *unlicensed users*, are forbidden even when those spectrum portions are unoccupied [9].

When these assigned portions are vacant, *frequency* or *spectrum holes* are created. A spectrum hole, also called white space, is a frequency band assigned to a PU, but it is not being used at a particular time and at a particular location [10]. This highlights an inefficient use of the spectrum that leads to spectrum scarcity for unlicensed users, also called Secondary Users (SUs). These allocation schemes cannot accommodate the requirements of an increasing number of higher data rate devices, resulting in the need for innovative techniques that can offer new ways of exploiting the available spectrum [5], and also help provide spectrum spaces for future technologies [11].

A solution to the spectrum scarcity and allocation issue would be to dynamically manage the spectrum by making available the unoccupied (vacant) channels/portions to SUs, without interfering the PUs signals and to achieve this, the Opportunistic Spectrum Access (OSA), also known as the Dynamic Spectrum Access (DSA), has been proposed. Unlike the FSA, DSA allows the sharing of the spectrum between SUs and PUs, whereby the spectrum is divided into numerous bandwidths assigned to one or more dedicated users [12] [13].

CR [14] was proposed as a solution that allows communication systems to make a more efficient use of the electromagnetic spectrum, by dynamically modifying the transceiver specifications according to the information sensed from their electromagnetic environment. CR-based technologies are *opportunistic* in nature as they provide a way for their users, SUs, to make an opportunistic use of licensed frequency bands when they are not occupied by their owners, i.e. PUs.

1.2 Current project objective

The aim of the final project is to explore different architectures of NNs which can be applied for time series forecasting problems and subsequently be used to predict the future evolution of the radioelectric spectrum for Cognitive-Radio applications.

1.3 Current project tasks

Given the broad aspect of the main project (i.e, NEURO-RADIO project), it is important to list the specific tasks that were carried out in this Final Masters Project (FMP).

1. Generate dataset that represents the temporal evolution of the occupancy level of a communications channel.
2. Implement five different ANNs models.
3. Fit/train the ANNs models.
4. Use the fitted network models to obtain predictions.
5. Compare the results gotten from the time it takes to build, compile and fit the network models and the accuracy of each model.

1.4 Report structure

The present report is divided into 4 chapters:

- Introduction

  Brief introduction that explains the motivation behind the project, and the long-term and short-term objectives that are expected to be met.

- Theoretical concepts

  Basic theoretical concepts that are the foundation of the project.

- Methodology and implementation

  Includes the design details and implementation of the project, from dataset obtention to network predictions.

- Results and conclusions

  Results of the network predictions are provided and conclusions are drawn from them.

1.5 Structure of current project folder

The scripts for this project are included in the compressed zip file called **tfm_Promise**. The contents of this file are:

- **MemoriaTFM**: A memory of the masters project in pdf format.
- scripts: A folder that contains the scripts used in the project.
  - **main**: Main script from where the other scripts are called and all necessary methods are instantiated. This script is run to train the network models and make predictions. The main variables that can be modified to the users discretion are:
    - *cpu_gpu_select*: variable to select either the CPU (0) or the GPU (1).
    - *folder_path*: folder path where to save generated data.
    - *train_percentage*: defines the percentage of the dataset that will be used for training (remaining will be used for prediction/test)
    - *n_input*: defines the total samples to used as input.
    - *n_output*: defines the number of output samples to be predicted.
    - *num_runs*: defines the total number of executions/runs.
    - *net_type*: selects the network type, where 1 = CNN, 2 = LSTM.
    - *lstm_model*: selects the LSTM model type, where 1 = Vanilla, 2 = Encoder-Decoder, 3 = CNN-LSTM, 4 = Conv-LSTM.
  - **generate_data_input**: Script to generate dataset.
  - **cnn_model**: Script where functions for the 1D CNN model are implemented.
  - **lstm_model**: Script where functions for the vanilla LSTM, encoder-decoder, CNN-LSTM, and ConvLSTM models are implemented.

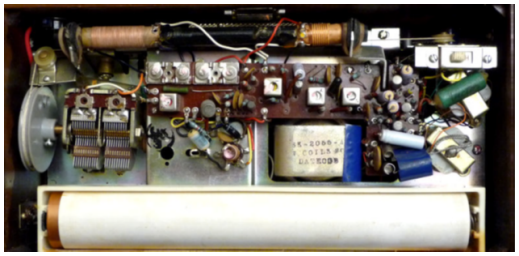
Chapter 2

Concepts and previous works

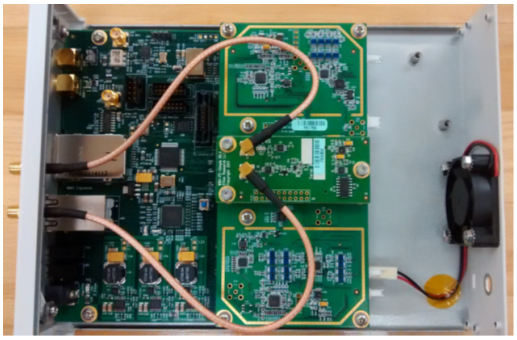
In this chapter, we explain the general concepts that make up the foundation of the project. We look at concepts like cognitive radio, spectrum sensing, neural networks etc

2.1 Software defined radio

The radio communication in traditional hardware radios is all carried out by hardware components that include amplifiers, capacitors, inductors, filters, modulators/demodulators etc. SDR) replaces some or all these physical components with software layers that implement the operating functions of the radio. SDRs include programmable analog-to-digital converters, and a microprocessor, FPGA, or general-purpose computer that acts as the digital processing hardware. An image comparison of the hardware of a traditional radio and an SDR is shown in Figure 2.1a and Figure 2.1b. Earlier mentions of SDRs are attributed to articles by J. Mitola [1][14]



(A) JVC Nivico TH-2770Z Transistor Radio.[1]



(B) (Ettus Research N210 software defined radio.

FIGURE 2.1: Visual comparison between a traditional hardware radio and a SDR.

[1] image retrieved from http://antiqueradio.org/

A block diagram of an ideal SDR transceiver [15], as it was originally conceived by Mitola [1] is shown in Figure 2.2, where the RF signal coming in from the antenna is directly digitized by an Analog-to-digital converter (ADC). The digitization is close to the antenna and most of the processing is performed by a high-speed general-purpose digital signal processor (DSP) [15]. Software run on the DSP can be used to implement functions like frequency tuning and translation, filtering, channel selection, demodulation, etc. On the transmitter side, a power amplifier (PA) is combined with a Digital-to-analog converter (DAC). This ideal SDR transceiver is unrealizable because the required specifications for both the ADC and the DAC are prohibitively rigorous and as such the realistic radio transceiver would contain an analog-signal-processing (ASP) section that includes signal conditioning (frequency translation, amplification, and filtering) [15]. This realistic implementation of the transceiver is shown in Figure 2.3.

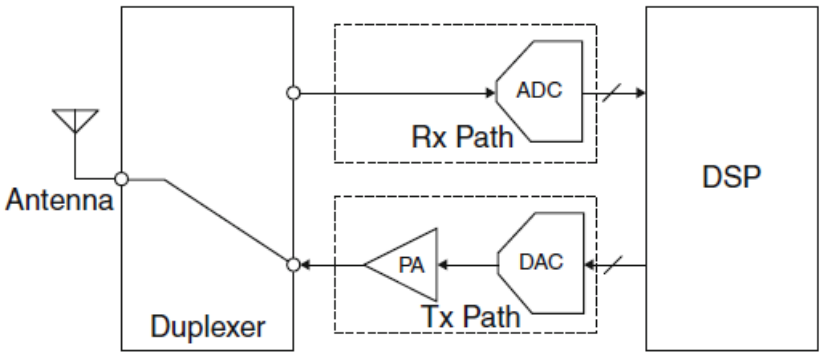


FIGURE 2.2: Ideal SDR transceiver as conceived by Mitola.

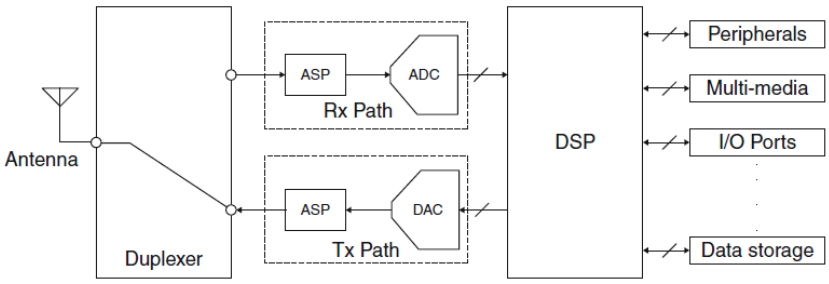


FIGURE 2.3: Realistic SDR transceiver.

2.2 Cognitive radio

First proposed by Mitola [14], it defines a radio that has the ability to sense their environment and make decisions based on the gotten information. The communication system of the radio is defined such that it can adjust its behaviour and adapt to meet its objectives, depending on the information gotten from the environment and its internal state. This ability to sense and make decisions is enabled by the software-defined layers of the radio. A cognitive radio must be self-aware, where

it's knowledge of a minimum set of basic facts about radio is used to communicate with other entities. An ideal CR, as conceived by Mitola, should be able to perform the following tasks: spectrum sensing, spectrum sharing, spectrum decision, and spectrum mobility [11]. A CR framework model is shown in Figure 2.4 [16].

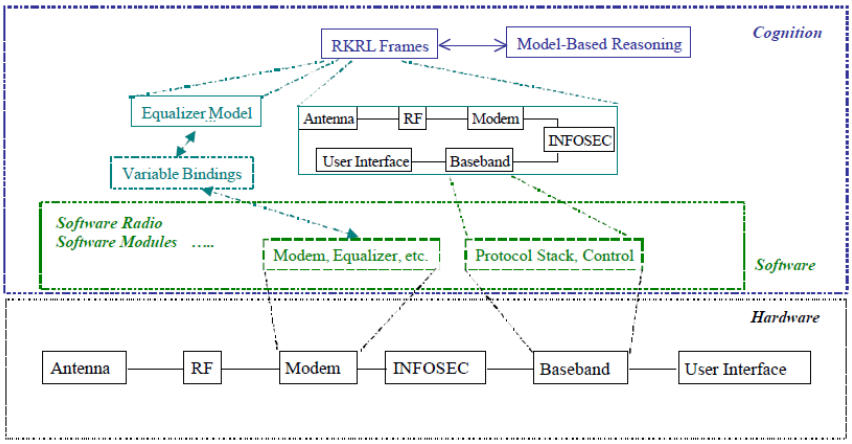


FIGURE 2.4: Cognitive Radio Framework.

The radio hardware (an SDR or Programmable Digital Radio (PDR)) consists of a set of modules: antenna, RF section, modem, information security (INFOSEC) module, baseband/ protocol processor, and user interface. The baseband processor hosts both the protocol and control software. The modem software includes the modem with equalizer, among other things. The framework also showcases how the cognitive radio has an internal model of its own hardware and software structure. The equalizer model would contain the codified knowledge about equalizers, including how the taps represent the channel impulse response and the variable bindings between it and the software equalizer establish the interface between the reasoning capability and the operational software. The model-based reasoning capability that applies these Radio Knowledge Representation Language (RKRL) frames to solve radio control problems gives the radio its "cognitive" ability.

Benefits of CR include usage optimization of assigned and unassigned spectrum, interoperability organization between users, network reconfiguration to meet current needs, etc.

2.3 Spectrum sensing

Spectrum Sensing (SS) is the task of obtaining awareness about the spectrum usage and existence of PUs in a geographical area. This awareness can be obtained by using geolocation and database, by using beacons, or by local spectrum sensing at CRs [5]. The SS model can be formulated as [10]:

$$y(n) = \begin{cases} w(n) & H_0\text{: PU absent} \\ h * s(n) + w(n), & H_1\text{: PU present} \end{cases} \tag{2.1}$$

where:

- $n = 1 \dots N$.
- N is the number of samples.

- $y(n)$ is the signal received by the SU.
- $s(n)$ is the PU signal.
- $w(n)$ is the additive white Gaussian noise (AWGN) with zero mean and variance, δ_w^2.
- h is the complex channel gain of the sensing channel.
- H_0 denotes the absence of the PU signal.
- H_1 denotes the presence of the PU signal.

A SS technique is used to detect PU signals and the detector output (test statistic) is then compared to a threshold in order to make the sensing decision about the PU signal presence. The sensing decision is performed as [10]:

$$\begin{cases} \text{if } T \geq \gamma, & H_1 \\ \text{if } T < \gamma, & H_0 \end{cases} \tag{2.2}$$

where T denotes the test statistic of the detector and γ denotes the sensing threshold. If the PU signal is absent, SUs can access the PU channel, and on the contrary, access is denied.

2.3.1 Spectrum sensing techniques

Many diverse SS techniques have been reported so far and they can be classified into two main categories: *cooperative sensing* and *non-cooperative sensing* [10][17] [18]. In cooperative sensing, SUs collaborate and coordinate with each other taking into account the objectives of each SU to make the final common decision. This cooperation between the different SUs can be divided into two schemes: *centralized* and *distributed* schemes.

In distributed scheme, for a given frequency band, the sensing result of each SU is exchanged with other SUs, where each SU makes its own final decision, basing on the received results of others. For the centralized scheme, all the SUs send their sensing results to a central unit, called fusion centre, that makes the final decision on which SU accesses the spectrum, using the sensing results.

Many proposed SS techniques include those based on matched filter based sensing [19], energy detection [20], cyclostationary detection [21] [22], wavelet detection [5], covariance detection [23], among others.

Energy detection

The received signal energy is compared with a threshold, that depends only on the noise power. The test statistic of the detector is computed from the squared magnitude of the Fast Fourier Transform (FFT) averaged over N samples of the signal the SU received. Different energy detection methods have been proposed, of which includes a method based on adaptive threshold in unknown white Gaussian noise with noise power estimation [24], a double-threshold technique in [25] with the intention of finding and localizing narrowband signals, and a technique in [26] based on wideband spectrum sensing, where the signal strength levels are sensed within several frequency ranges to improve the opportunistic throughput of the SU and decrease the interference to the PU [10]. Some of the challenges with energy detector

based sensing include selection of the threshold for detecting primary users, inability to differentiate interference from primary users and noise, and poor performance under low signal-to-noise ratio (SNR) values [27].

Wavelet detection

Also known as *edge* detection, it is based on the continuous wavelet transform, which allows finding the signal decomposed coefficients with the help of a basis [28] [29]. The continuous wavelet transform of the received signal is computed to perform the power spectral density, where the local maximum of the power spectral density corresponds to the edge, which is then compared to a threshold to decide about the spectrum occupancy.

Matched filter detection

It is based on a coherent pilot sensor that maximizes the Signal-to-Noise Ratio (SNR) at the output of the detector. It is an optimal filter that requires the prior knowledge of the PU signals and it is the best choice when some information about the PU signal are available at the SU receiver [10] [19] [30].

2.3.2 Machine learning applied to spectrum sensing

A CR system must be able to perceive (perception), learn (learning), and reason (reasoning), where perception can be achieved through the sensing measurements of the spectrum that allows the cognitive radio to identify ongoing RF activities in its surrounding environment. The sensing observations/results acquired is used by the CR system to *learn*, where it tries to classify and organize the observations into suitable categories using learning algorithms. The knowledge acquired through learning is applied by the reasoning ability of the CR system to achieve its objectives. This process describes the so-called cognition cycle [14]. An illustration of the cycle is shown in Figure 2.5 [31].

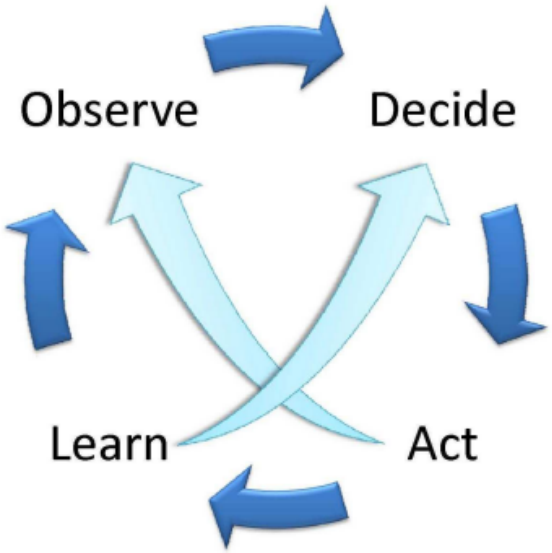


FIGURE 2.5: Illustration of a cognition cycle of a CR.

Learning algorithms for CRs are classified under two main categories: Supervised and unsupervised learning. This classification is depicted in Figure 2.6 [31]. In supervised learning, a supervisor determines if an action carried out by an agent is correct or wrong, whereas in unsupervised learning, there is no presence of such supervisors.

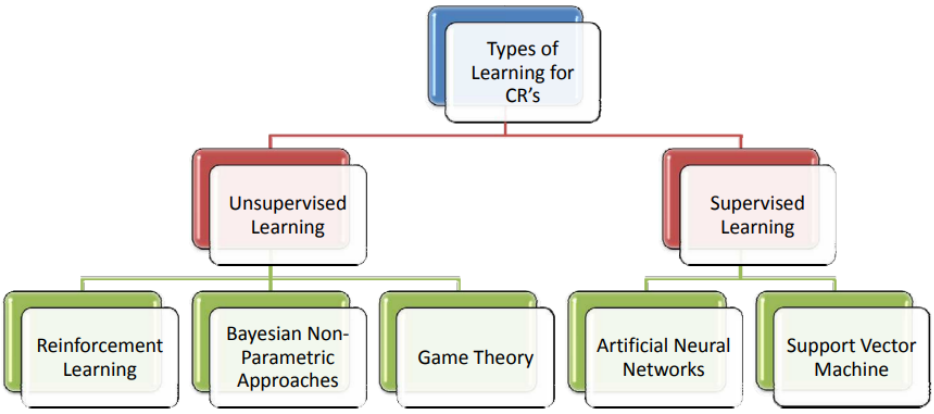


FIGURE 2.6: Classification of learning approaches for CRs.

Unsupervised learning

Reinforcement learning (RL) With Reinforcement learning (RL), an agent can modify its behaviour by interacting with its environment [31]. Agents can learn autonomously without supervision, where they learn from the feedback the receive from their environment after executing an action. Reinforcement learning is characterized by two main features: *trial-and-error* and *delayed reward*. By trial-and-error, it is assumed that an agent does not have any prior knowledge about the environment, and it executes some actions blindly in order to explore the environment. The delayed reward is the feedback signal that an agent receives from the environment after executing each action [31]. These rewards are observed constantly by the agents for each action.

Non-parametric Learning: The Dirichlet Process Mixture Model (DPMM) The Dirichlet process has been used as a framework for non-parametric Bayesian learning in cognitive radios in [32], and in [33], where it was used for identifying and classifying wireless systems.

Game theory-based Learning Game theory [34] is a mathematical tool that implements the behaviour of rational entities in an environment of conflict. In CR applications, it is applied to CR protocols to reduce the complexity of adaptation algorithms in large cognitive networks [31].

Supervised learning

Supervised learning techniques are generally used in known environments, where there is prior information about the characteristics of the environments.

Artificial Neural Network Previous ANN techniques applied to CR include the use of Multi-layered Feedforward Neural Networks (MFNN) as a technique to synthesize performance evaluation functions in cognitive radios [35], the use of an ANN-based cognitive engine that learns how environmental measurements and the status of the network affect its performance on different channels [36], or the use of a Feed-backward ANN in conjunction with cyclostationarity based spectrum sensing [37].

Support Vector Machine The Support Vector Machine (SVM), first proposed by Vapnik [38], is used for many machine learning tasks such as pattern recognition and object classifications. In CR applications, it has mostly been used in performing signal classifications [39].

2.4 Deep learning for time series forecasting

A *time series* data is a sequence of observations taken sequentially in time [40]. *Forecasting*, on the other hand, involves taking models fit on historical data and using them to predict future observations using ANNs.

An approach to solving the spectrum sensing problem of CR is based on translating the identification problem of frequency holes to a time-series prediction problem, and where in the past, RNNs, and more recently, a specific type of RNN called Long Short-Term Memory (LSTM) network has been proposed and found to be suitable in predicting temporal evolution of data [3], [4].

2.4.1 Artificial neural networks for time series forecasting

ANNs, sometimes also referred simply as NNs, are computer systems/models made up of algorithms that carry out tasks like data processing, classification, predictions, function approximations etc., by learning, and which are modelled after the neurons found in the human brain. The neurons in the human brain, working in parallel with one another, receive inputs and their outputs, result of some computational operations on the inputs are passed between one another. ANNs emulate this behaviour of neurons with the so-called units. A unit receives inputs from other units through connections to other units or input values. The connections, analog to synapses in the human brain, act in one way or the other on the units and with varying strengths. The strength of each connection is referred to as its weight. An ANN learns by analysing training examples (labelled or unlabelled data) without being programmed directly with rules to perform a concrete task and the different ways in which it learns include:

- Supervised learning: Involves making use of labelled datasets as the training input data. The algorithm analyses the labelled dataset, and its predictions are checked against the true answer and depending on whether the predictions are true or false, the network weights are modified accordingly until the desired result is achieved.
- Unsupervised learning: The given dataset is not labelled in this method of learning. The algorithm analyses the unlabelled input data and "guesses" the pattern. With a cost function, the prediction is measured to give its accuracy and adjustments are made depending on the results. Some common unsupervised learning algorithms and subsequent applications include clustering, anomaly detection, neural networks for auto-encoders, Deep Belief Nets etc.

- Reinforced(-ment) learning: The network is not told which actions to take but learns by discovering which actions provoke the best results. As it learns what to do, it rewards itself positively or negatively, depending on the result, and maximizes a numerical value that represents a long-term objective.

There are different classes of ANNs in terms of their connective structure of which includes CNNs, RNNs, Feedforward Neural Networks (FNNs), Modular Neural Networks, etc., each covering a wide range of applications. There are other classifications based on different aspects like the number of layers present in the network or the presence of a feedback but given the scope and objectives of the current project, we take a look at only the classification based on their connective structure and, in concrete, first two aforementioned classes, CNNs and RNNs.

Convolutional neural networks (CNNs)

One of the different classes of ANNs, CNNs are feedforward ANNs with alternating convolutional and subsampling layers, where they are based on a reduced set of connections between layers (projective field or convolutional kernel) which reduces the amount of connections and number of different weights, simplifying the computational cost. The work by Fukushima and Miyake in 1982 [41], considered to be the predecessor of CNNs, describes a self-organized, hierarchical network that has the capability to recognize stimulus patterns based on the differences in their appearances (e.g., shapes). With this network, called "Neocognitron", it was evident the need to develop a supervised method to train (or adapt) it for the learning task in hand [42].

The solution would arrive with the so-called "LeNet" by Yann LeCun in 1990, considered to be the first CNN [43]. Earlier models of CNNs were only limited to low-resolution and gray-scale images in small-size datasets. In 2012 came the first deep CNN, called "AlexNet" [44]. A deep network is a feed-forward network with more than one hidden layer (where hidden layers are those different from the input and output layers). In subsequent years, following the success of AlexNet, various deep 2D CNNs were proposed, which includes "ZFnet" (2013) [45], and "GoogLeNet" (2014) [46].

The architecture of LeNet consisted of two interleaved convolutional and pooling layers, followed by three (two hidden and one output) fully-connected layers. The output layer is made up of 10 Radial Basis Function (RBF) neurons, each of which computes the Euclidean distance between the network output and ground truth label for 10 classes. This configuration is shown in Figure 2.7 [47].

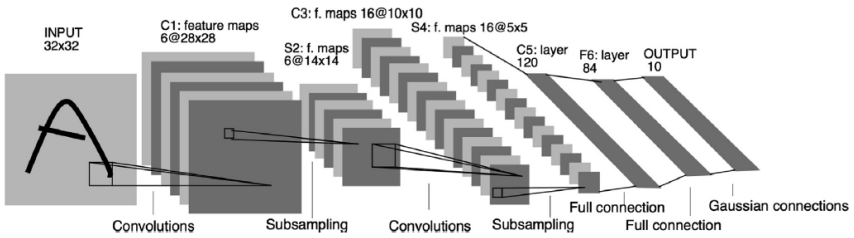


FIGURE 2.7: Architecture configuration of LeNet

In the case of AlexNet, the first deep CNN, 5 convolutional layers and 3 max-pooling layers are followed by three (two hidden and one output) fully-connected

(dense) layers. The output layer neurons implement a softmax loss of the network predictions for 1000 classes. This configuration is shown in Figure 2.8 [44].

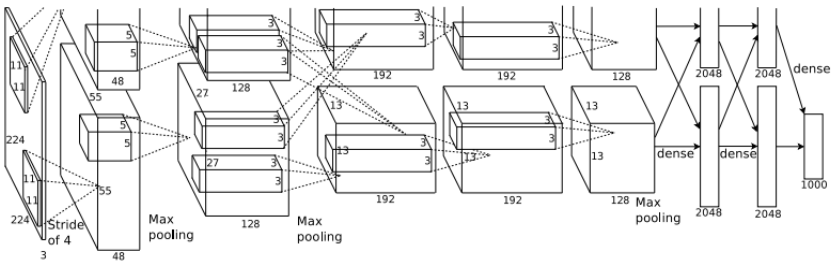


FIGURE 2.8: Architecture configuration of the first deep CNN, AlexNet

To explain how a CNNs works, let us consider the following sample model [42] that classifies a 24 × 24-pixel grayscale image into two categories. The configuration of this sample model is shown in Figure 2.9.

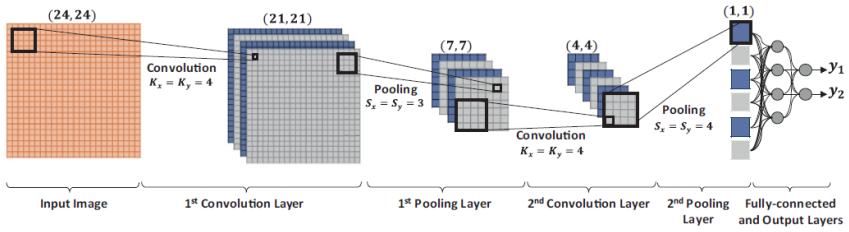


FIGURE 2.9: Configuration of sample CNN with 2 convolution and one fully-connected layers.

This sample network consists of two convolution and two pooling layers with 4 and 6 neurons, respectively. The output of the last pooling layer is processed by a single fully-connected layer and followed by the output layer that produces the classification output. The interconnections feeding the convolutional layers are assigned by weighting filters (w) having a kernel size of (K_x, K_y). The convolution takes place within the image boundaries; therefore, the feature map dimension is reduced by the ($K_x - 1, K_y - 1$) pixels from the width and height, respectively. The subsampling factors (S_x, S_y) are set in advance in the pooling layers. In the sample illustration in the figure, the kernel sizes corresponding to the two convolution layers were set to $K_x = K_y = 4$, while the subsampling factors are set as $S_x = S_y = 3$ for the first pooling layer and $S_x = S_y = 4$ for the second one. Note that these values were deliberately selected so that the outputs of the last pooling layer (i.e. the input to the fully-connected layer) are scalars (1x1). The output layer consists of two fully-connected neurons corresponding to the number of classes to which the image is categorized.

Advantages of CNNs include, as mentioned in [42], the fusion of the feature extraction and feature classification processes into a single learning body, where the models learn to optimize the features during the training phase directly from the raw input. Also, they have the ability to process large inputs with greater computational efficiency compared to the conventional fully-connected Multi-Layer Perceptrons (MLP) networks, due to them being sparsely-connected with tied weights, and

they are unaffected by small transformations, like translation, scaling, skewing and distortion, in the input data.

1D CNNs

Although CNNs were originally designed to efficiently handle and operate on 2D image data by directly extracting features from the raw data, modifications on them has led to the creation of alternatives such as 1D CNNs, that receives and operates on 1D data. Some advantages that they present over conventional 2D CNNs include less computational requirements and complexities, compact architectures with few hidden layers over deep architectures, and being suitable for real-time and low-cost applications [42].

A simple 1D CNNs configuration is formed by the following hyperparameters:

- Number of hidden CNN and MLP layers/neurons.
- Filter (kernel) size in each CNN layer.
- Subsampling factor in each CNN layer.
- Selected pooling and activation functions.

Figure 2.10 shows a sample configuration of a 1D CNN that has three consecutive hidden CNN layers and 2 hidden Multi-Layer Perceptrons (MLPs) [42]. The filter (kernel) size for all the hidden CNN layers is 41, and with a subsampling factor of 4. The raw 1D data is processed by the CNN layers and the features present are learnt and extracted.

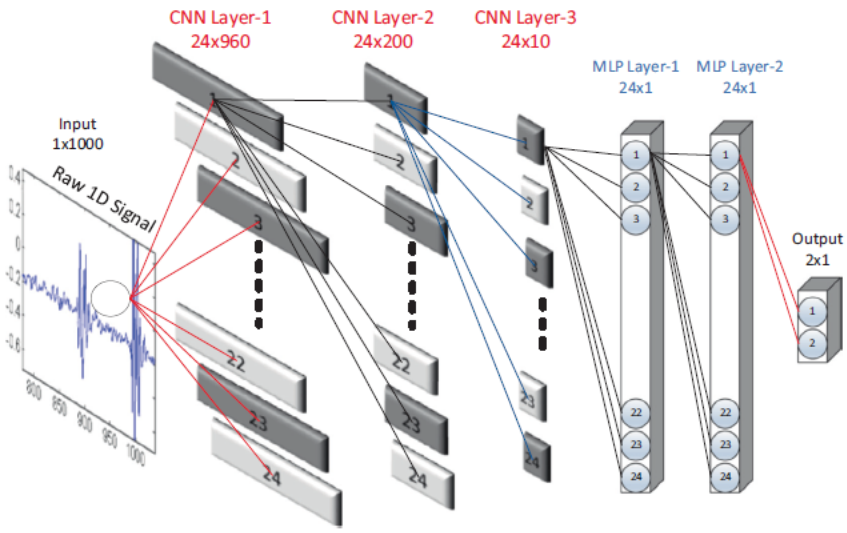


FIGURE 2.10: Configuration of sample 1D CNN with 3 consecutive CNN layers.

Recurrent neural networks (RNNs)

RNNs are a type of artificial neural networks that uses sequential data or time series data, and in such cases, the connections between nodes form a directed or undirected graph along a temporal sequence. The fundamental feature of a RNN is the

presence of at least one feed-back connection, that allows activations to flow round in a loop. This feature differentiates RNNs from feed-forward networks that only include connections from layer i to i+1, since RNNs include connections to previous layers (for example, connection from layer i to i-1).

Their "memory" also differentiates RNNs from their traditional counterparts, as they are capable of extracting information/features from prior inputs to influence the current input and output. They are networks with loops in them, allowing information to persist. Its applications include language translation, natural language processing (nlp), speech recognition, and image captioning.

Figure 2.11 shows an illustration of a rolled RNN [48], where a group of neural network, A, looks at some input x_t and outputs a value h_t and a loop allows information to be passed from one step of the network to the next. In reality, a RNN has a chain-like nature, where it can be thought of as multiple copies of the same network, each passing information to the next in line. This behaviour is represented in Figure 2.12.

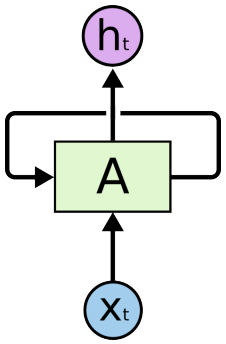


FIGURE 2.11: Rolled RNN.

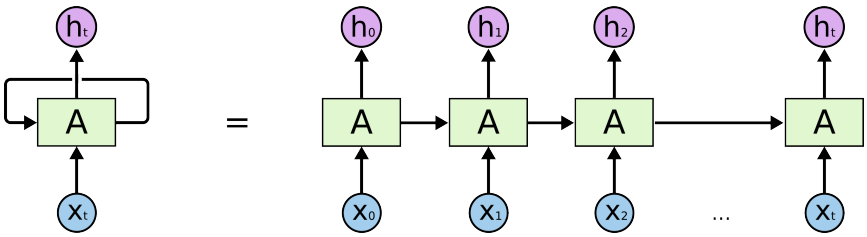


FIGURE 2.12: Unrolled RNN.

Long-term dependencies As earlier mentioned, a characteristic of RNNs is their ability to use information from prior inputs to connect to the present input, but there comes a time when the distance between the information needed from a prior input and the present input is large and the network cannot make the connection. This describes a *"Long-term dependency"* problem that RNNs face. Although, theoretically, they can handle long-term dependencies, they are unable to learn them in practice. This problem is also known as the *"Exploding and vanishing gradient problem"*.

Long short-term memory networks (LSTMs)

LSTMs are a type of RNNs that was introduced by Hochreiter and Schmidhuber [49], to solve the long-term dependency problem, where they remember information for long periods of time. LSTMs are capable of modelling longer term dependencies by having memory cells and gates that controls the information flow along with the memory cell.

All recurrent neural networks have the form of a chain of repeating modules of neural network, and in standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer, represented in Figure 2.13.

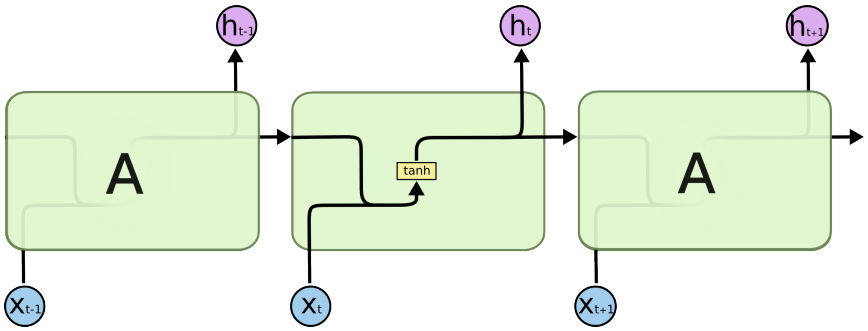


FIGURE 2.13: Repeating module with a single layer in a standard RNN.

In the case of the LSTM, the repeating module is made up of *four* neural network layers, as shown in Figure 2.14.

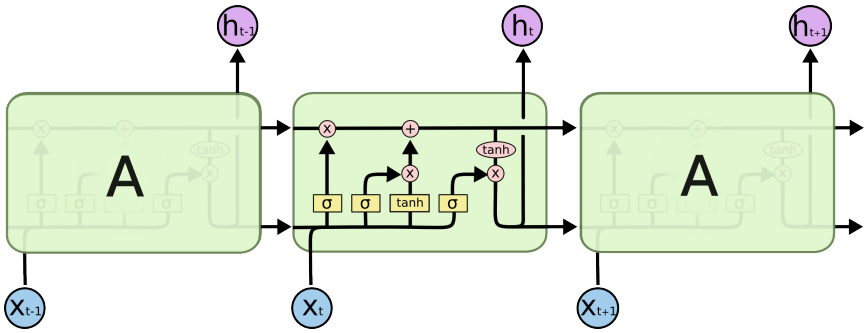


FIGURE 2.14: Repeating module in an LSTM with four interacting layers.

To better understand the figures, Figure 2.15 shows the meaning of the notations used on the figures, where each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The horizontal line that runs through the top of the diagram is known as the *cell state*. This is shown in Figure 2.16.

Gates, shown in Figure 2.17, are used to regulate the information that is added or removed from the cell gate. They are composed of a sigmoid neural net layer and a pointwise multiplication operation, where the sigmoid layer outputs numbers

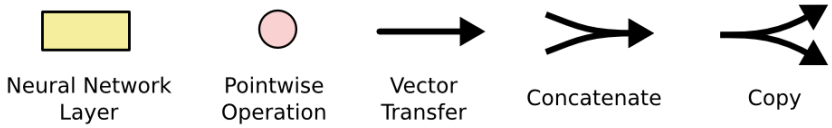


FIGURE 2.15: Notation meanings.

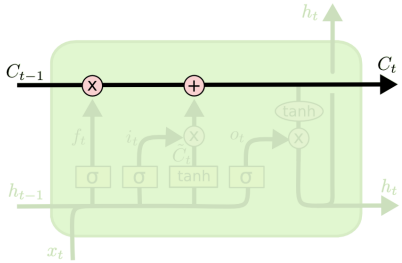


FIGURE 2.16: Cell state.

between zero and one, that describes how much of each component should be let through. A value of zero means nothing is let through while a value of one is the exact opposite. An LSTM has three of these gates that protect and control the cell state.

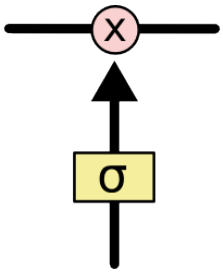
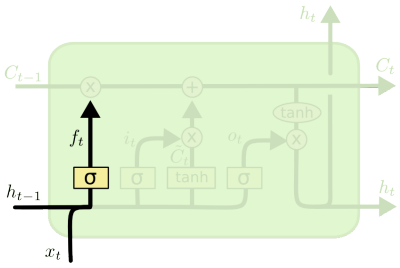


FIGURE 2.17: Cell gate.

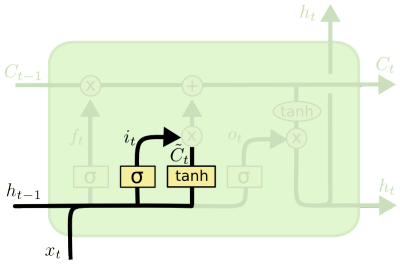
Step-by-Step LSTM Walk Through

- A decision is made by the a sigmoid layer called the *"forget gate layer"* on which information is kept or rejected, i.e, it determines how much contents from previous cell C_{t-1} will be erased. It looks at h_{t-1} and x_t, and outputs a number between 0 (completely eliminate) and 1 (completely keep) for each number in the cell state C_{t-1}. This is represented in Figure 2.18.
- In the next step, represented in Figure 2.19, a decision is made on what information is to be stored in the cell state. In the first part of this process, a sigmoid layer called the "input gate layer", i_t, decides which values are to be updated. Next, a *tanh* layer creates a vector of new candidate values, $\tilde{C}_t$, as a function of h_{t-1} and x_t, that could be added to the state.
- The old cell state C_{t-1} is updated, into the new cell state C_t by using the input and forget gates with new candidate cell states. The old state is multiplied by f_t, to make it to forget information already chosen to be forgotten. The



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

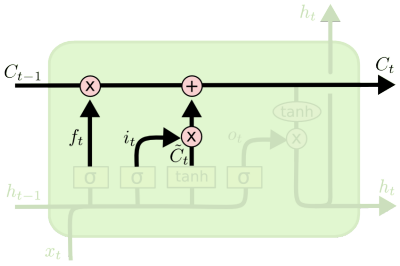
FIGURE 2.18: Forget gate.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

FIGURE 2.19: New candidate cell and input gate.

new candidate values, $i_t * \tilde{C}_t$ are added and scaled by how much is decided to update each state value. Figure 2.20 shows a representation of the update process.

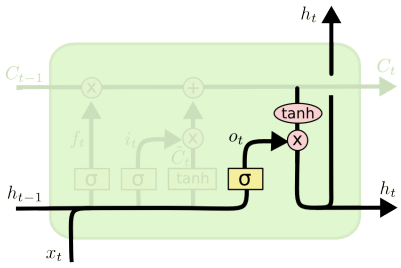


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

FIGURE 2.20: Update cell states.

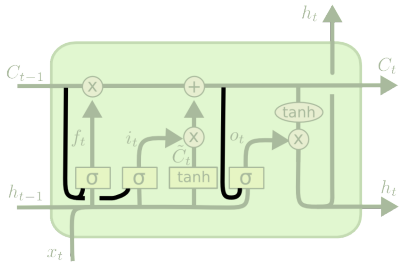
- Finally, a decision is made on the output, which will be based on the cell state, but will be a filtered version. The output gate o_t (sigmoid layer)decides which part of cell state C_t will be in the output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so as to filter the parts that were decided. This process is represented in Figure 2.21.

Various variants of LSTMs have been developed, such as the *peephole connections* variant of Figure 2.22 that allows the gate layers to look at the cell state, another variation that uses coupled forget and input gates (as seen in Figure 2.23), or the Gated Recurrent Unit, or GRU (shown in Figure 2.24), introduced by [50], that combines the forget and input gates into a single "update gate" and also merges the cell state and hidden state.



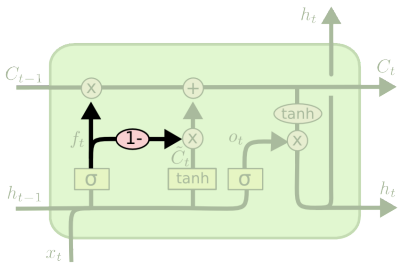
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

FIGURE 2.21: Output generation.



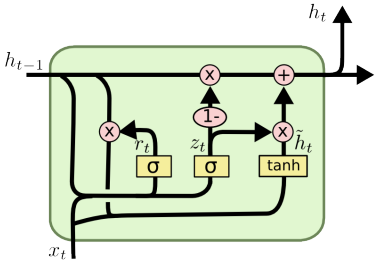
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t]\ +\ b_o\right)$$

FIGURE 2.22: Peephole connections LSTM variant.



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

FIGURE 2.23: Tied forget and input gate LSTM variant.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

FIGURE 2.24: GRU LSTM variant.

Chapter 3

Methodology and Content Development

This chapter deals with the obtention of training data for the different network models we will be evaluating, detailed explanation of the architecture/structure of the networks, and the scripts used in their implementation.

The methodology implemented in this project is based on the described methodology for time series forecasting by J. Brownlee [51]. The design methodology is summarised as follows:

1. Split the dataset into a training and test subsets.
2. Build a network model.
3. Fit the network model on the training dataset.
4. Make predictions on the test set using walk-forward validation method.
5. Calculate and use the Root Mean-Square Error (RMSE) metric to compare the predictions to the expected values.

3.1 Software requirements

The scripts used in this project are all written in Python and the simulations are run with the PyCharm Integrated Development Environment (IDE) (more information can be found in Appendix B). To be able to run the scripts some dependencies or libraries (packages) are required. The list of libraries that are needed for a correct execution of each script are listed at the start of each corresponding script.

PyCharm provides a convenient and easy way to install/download each library/-package, and instructions on how to do so can be found at: https://www.jetbrains.com/help/pycharm/installing-uninstalling-and-upgrading-packages.html#interpreter-settings.

3.2 Dataset

The dataset used for both training and evaluation of the network models is a univariate time series data that models the temporal evolution of the occupancy level of a communications channel. We implement a function, called *generate_data_input(...)* and provided in a script of the same name, that generates a sequence that represents this evolution. This script accepts *four* parameters that modifies the behaviour of the channel, and they are:

- multilevel_sequence = array with time slots that represents the temporal evolution of the occupancy level of a communications channel. The signal value in each of these time slots can be assigned intermediate occupation values using integers of any value and range to represent the multilevel nature of the sequence.
- length_symbol = number of samples in each time slot.
- length_transition = samples that correspond to the transition between adjacent slots.
- noise_prop = noise level added to the sequence.

For the present project, the values selected for these parameters are:

- multilevel_sequence

  [2, 4, 4, 4, 0, 4, 1, 2, 0, 2, 4, 0, 4, 3, 0, 4, 4, 2, 0, 2, 2, 0, 0, 0, 4, 0, 4, 0, 4, 4, 0, 0, 4, 0, 0, 0, 4, 4, 0, 1, 3, 0, 4, 1, 2, 4, 4, 0, 0, 1, 0, 4, 4, 4, 4, 4, 0, 0, 2, 1, 1, 4, 0, 0, 4, 0, 0, 1, 1, 4, 0, 2, 3, 4, 4, 0, 0, 4, 1, 4, 4, 4, 4, 4, 1, 0, 4, 0, 3, 1, 1, 4, 0, 0, 3, 0, 1, 4, 2, 0, 0, 4, 2, 0, 2, 4, 0, 0, 4, 0, 4, 0, 0, 2, 4, 2, 0, 4, 0, 1, 2, 1, 1, 0, 0, 0, 0, 4, 4, 1, 4, 4, 4, 4, 4, 0, 4, 0, 1, 0, 4, 0, 0, 0, 4, 1, 1, 0, 2, 4, 0, 0, 4, 1, 0, 4, 0, 0, 0, 4, 0, 0, 3, 4, 3, 3, 0, 0, 0, 4, 4, 3, 0, 0, 1, 1, 3, 3, 4, 0, 3, 3, 4, 2, 1, 0, 2, 4, 2, 0, 4, 3, 3, 2, 0, 4, 3, 4, 4, 3]

  As seen above, the values that define the time slot go from the range of zero to four (both inclusive), representing a 5-level multilevel sequence. It should be noted that these values are later normalized within the range of 0 to 1, where 0 means that the communications channel is completely free during the corresponding time slot, and 1 means that it is saturated.

- length_symbol = 100
- length_transition = 70
- noise_prop = 0.15

The plotted data is shown in Figure 3.1.

3.2.1 Dataset preparation for network models: Train and test data

With the aim of forecasting multiple timesteps, the dataset is configured to meet such aim. Before that is done, we first split the dataset in two: *training set*, that will be used for training predictive network models, and *test set*, for evaluating the models. The value given to a modifiable parameter named ***train_percentage*** represents the percentage of the total dataset that is selected for training.

With the two parameters ***n_input*** and ***n_output***, we select the number of timesteps that are used as input data for the model and the number of timesteps that are to be predicted by the model, respectively. The *n_output* parameter is also used to divide the dataset into multiple samples of *n_output*-sized subsets.

In this present project, *train_percentage* is set to 75, i.e, 75% of the total dataset is used to train the network models, and the rest, 25%, is used to evaluate the models. *n_input* and *n_output* are either set to 5, 6 or 10. This means that the data passed as input to the models and the forecasting/predictions will be done in multi-steps of size 5, 6 or 10. The *split_dataset(..)* method that implements the division of the

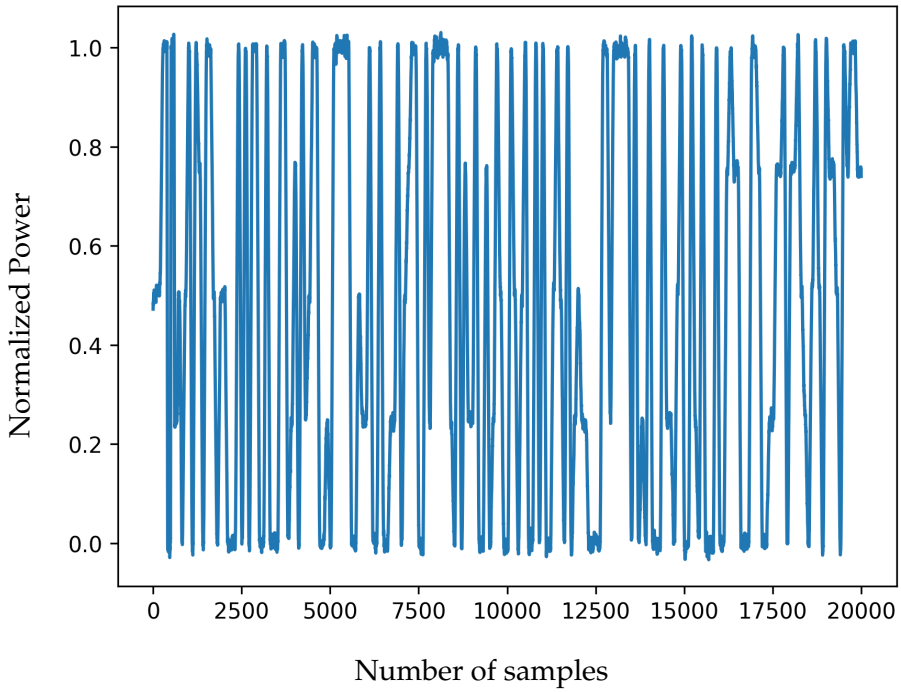


FIGURE 3.1: Plotted data.

dataset into training and test sets, and also configures and organizes the dataset into subsets of size *n_output* is given below in Listing 1.

```
# split a univariate dataset into train / test sets
def split_dataset(data, train_percentage, n_output):
    # split data according to train and test percentages
    limit_train = int((train_percentage/100)*len(data))
    train, test = data[0:limit_train], data[limit_train:len(data) + 1]

    # reshape data
    train = train.reshape(len(train), 1)
    test = test.reshape(len(test), 1)

    # restructure into windows of sequence data
    train = array(split(train, int(len(train)/n_output)))
    test = array(split(test, int(len(test)/n_output)))
    return train, test
```

LISTING 1: Splitting dataset into training and test subsets.

3.2.2 Training data preparation

We prepare the dataset obtained from the split of the previous section for a supervised learning model. This means that the training data needs to be divided into multiple samples that the network models learn from and generalize across. To

achieve this, the *sliding window* method is used to prepare the dataset for a supervised learning model, where the training data is divided into two components: input (X) and output component (Y).

The input component denotes the number of prior observations, for example, the first 10 timesteps, while the output component represents the observations at the current timestep. For example, if the number of timesteps selected is 10, there would be 10 current observations. To understand the *sliding window* concept better, let us imagine we have a dataset as shown below in Table 3.1.

TABLE 3.1: Time series dataset to explain sliding window concept.

Time	**Measure**
1	0.2
2	0.25
3	0.30
4	0.35

As we can see from the dataset, it is a time series dataset and when it is restructured as a supervised learning model by applying the sliding window concept, where the previous timestep observation is used to predict the next timestep observation, we have the following result seen in Table 3.2:

TABLE 3.2: Sliding window concept implementation.

X	**Y**
?	0.2
0.2	0.25
0.25	0.30
0.30	0.35
0.35	?

In the present project, this is implemented with the *to_supervised()* method, which is shown below in Listing 2.

```
# convert history into inputs and outputs
def to_supervised(train, n_input, n_output):
    # flatten data
    data = train.reshape((train.shape[0]*train.shape[1],
    ↪  train.shape[2]))
    x, y = list(), list()
    in_start = 0
    # step over the entire history one time step at a time
    for _ in range(len(data)):
        # define the end of the input sequence
        in_end = in_start + n_input
        out_end = in_end + n_output
        # ensure we have enough data for this instance
        if out_end < len(data):
            x_input = data[in_start:in_end, 0]
            x_input = x_input.reshape((len(x_input), 1))
            x.append(x_input)
            y.append(data[in_end:out_end, 0])
        # move along one time step
        in_start += 1
    return array(x), array(y)
```

LISTING 2: Restructure training dataset into a supervised learning model using the sliding window concept.

Train is the training dataset, and *n_input* and *n_output* both represent the number of inputs (current observations) and number of outputs (predicted observations), respectively.

3.3 Neural networks implementation with Keras

As earlier mentioned, the main aim of this final project is to explore different architectures of ANNs which can be applied for time series forecasting problems and subsequently be used to predict the future evolution of the radioelectric spectrum for Cognitive-Radio applications. These ANN models include CNNs, LSTMs, or hybrid combinations of both CNNs and LSTMs. To achieve this aim of solving the problem of time-series prediction, five NN models are chosen to be studied, and whose performance will be compared, and they are:

- 1D CNN model
- Vanilla LSTM model
- Encoder-decoder LSTM model
- CNN-LSTM model
- ConvLSTM model

The neural networks under study and implemented in the project are all done in Keras. Keras is a deep learning (DL) Application Programming Interface (API)

written in Python [6]. It runs on top of an ML platform named TensorFlow. The core data structures of Keras are layers and models. A layer consists of a tensor-in tensor-out computation function, which is the layer's call method, and some states (the layer's weights), held in TensorFlow variables. For the different layers that make up each ANN model, the two main setting parameters which are tuned, and which determine the performance of the ANNs are: *filters*, also called *units*, and *activation functions*.

A filter (unit in the case of LSTMs) is an integer that specifies the dimensionality of the output space. In the case of convolution layers, it represents the number of output filters in the convolution. For LSTM models, a unit is also used to refer to the dimensionality of the hidden state.

Also known as a transfer function, an activation function defines how the weight -ed sum of the input is transformed into an output from a node or nodes in a layer of the NN. It introduces non-linearity into the output of a neuron. The influence of activation functions is also noted in the convergence ability and speed of the neural network or in blocking the convergence.

The layers which are used in the project are [52]:

- Conv1D: is a 1D convolution layer that creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs.
- MaxPooling1D: is a layer that implements a max pooling operation for 1D temporal data, where it downsamples the input representation by taking the maximum value over a spatial window of variable size. A *pool_size* argument is provided that represents the size of the max pooling window.
- LSTM: a Long Short-Term Memory layer.
- Dense: is a densely-connected NN layer that implements the operation:

$$output = activation(dot(input, kernel) + bias) \tag{3.1}$$

  where *activation* is the element-wise activation function passed as the activation argument, *kernel* is a weights matrix created by the layer, and *bias* is a bias vector created by the layer (only applicable if use_bias flag is set as True).
- TimeDistributed: is a wrapper which allows the application of a layer to every temporal slice of an input, i.e, the layer provided to the wrapper will be applied to each of the input timesteps, independently.
- Flatten: is a layer that flattens the input, and which does not affect the batch size.
- RepeatVector: is a layer that repeats the input n-given times.

There are 3 ways to develop NN models in Keras: The *sequential model*, *functional model*, and *model subclassing*. The Sequential model involves creating a plain stack of layers where each layer has exactly one input tensor and one output tensor, thus, making it prohibitive for multiple input and output models. Functional model facilitates the creation of flexible NN models that are capable of multiple inputs and/or outputs, non-linear topology, and shared layers. Model subclassing is used for creating complex models from scratch. In this current project, we do not implement the model subclassing method of creating NN models.

3.3.1 Input layer configuration

Before going into the rest of the architectures of each of the NN models, we look at how their input layers are configured. The configuration for 1D CNN, vanilla LSTM, encoder-decoder LSTM, and CNN-LSTM are the same. The shape of the input layer for these four models is given by the number of timesteps that is defined for the model and the number of features present in each sample. As mentioned earlier in subsection 3.2.1, the number of timesteps is defined by the *n_output* parameter. The input layer is called together with the first hidden layer of the model. For example, in the case of 1D CNN, the implementation is as seen in Listing 3.

```
# adding the input layer and the first hidden layer
model.add(Conv1D(filters=24, kernel_size=3, activation='relu',
↪  input_shape=(n_timesteps, n_features)))
```

LISTING 3: Input layer instantiation in hidden layer.

From the above example, the input layer is declared with *input_shape=(n_timesteps, n_features)* in the hidden layer that is *Conv1D*.

In the case of ConvLSTM, the *ConvLSTM2D* class which is used to implement the network model, by default, expects input data to have the shape: [samples, timesteps, rows, cols, channels]. As such, the input layer is configured to *input_shape=(n_steps, 1, n_length, n_features)*, where *n_steps* is the numer of timesteps,

As the dataset is a univariate time series, the number of features present in all five models is one.

3.3.2 Sequential model

1D CNN

The 1D CNN model has a convolutional hidden layer, with a kernel size of 3, that operates over a 1D sequence, followed by a pooling layer, that downscales the output of the convolutional layer, highlighting only the most significant features. A flatten layer is then used to reduce the feature maps to a single one-dimensional vector, followed by a fully connected dense layer that interprets the features extracted by the convolutional part of the model. The definition of this model in Keras is shown in the following line of code of Listing 4.

```
# define model
model = Sequential()
# adding the input layer and the first hidden layer
model.add(Conv1D(filters=24, kernel_size=3, activation='relu',
↪  input_shape=(n_timesteps, n_features)))
# adding a second hidden layer
model.add(MaxPooling1D(pool_size=2))
# adding a third hidden layer
model.add(Flatten())
# adding a fourth hidden layer
model.add(Dense(10, activation='relu'))
# adding the output layer
model.add(Dense(n_outputs))
```

LISTING 4: 1D CNN model definition.

A plot of the network model where the number of inputs and outputs is selected to be 10 and the number of filters for the convolution layer is 8, is shown in Figure 3.2. This plot is generated with the following code shown in Listing 5.

```
# Plot and save network model
plot_model(model, show_shapes=True, to_file=folder_path + name_model +
↪  '_network_model.png', dpi=300)
```

LISTING 5: Generate model plot.

model is the complete built layers, *name_model* denotes the network model name and *folder_path* is where the plotted image is stored. The same code snippet is the same for getting the plots of all network models in the project.

The plot in Figure 3.2 provides a schematic view of the built model, where it includes the shape of the inputs and outputs of each layer. An explanation of the model for this present example is as follows:

- The left-hand side of each box represents the layers.
- The right-hand side lists the shape of the input and output of the selected layer.
  - input: The input shape varies depending on the network model and the type of layer. In the example of Figure 3.2, the *InputLayer* has an input shape of [None, 10, 1], where:

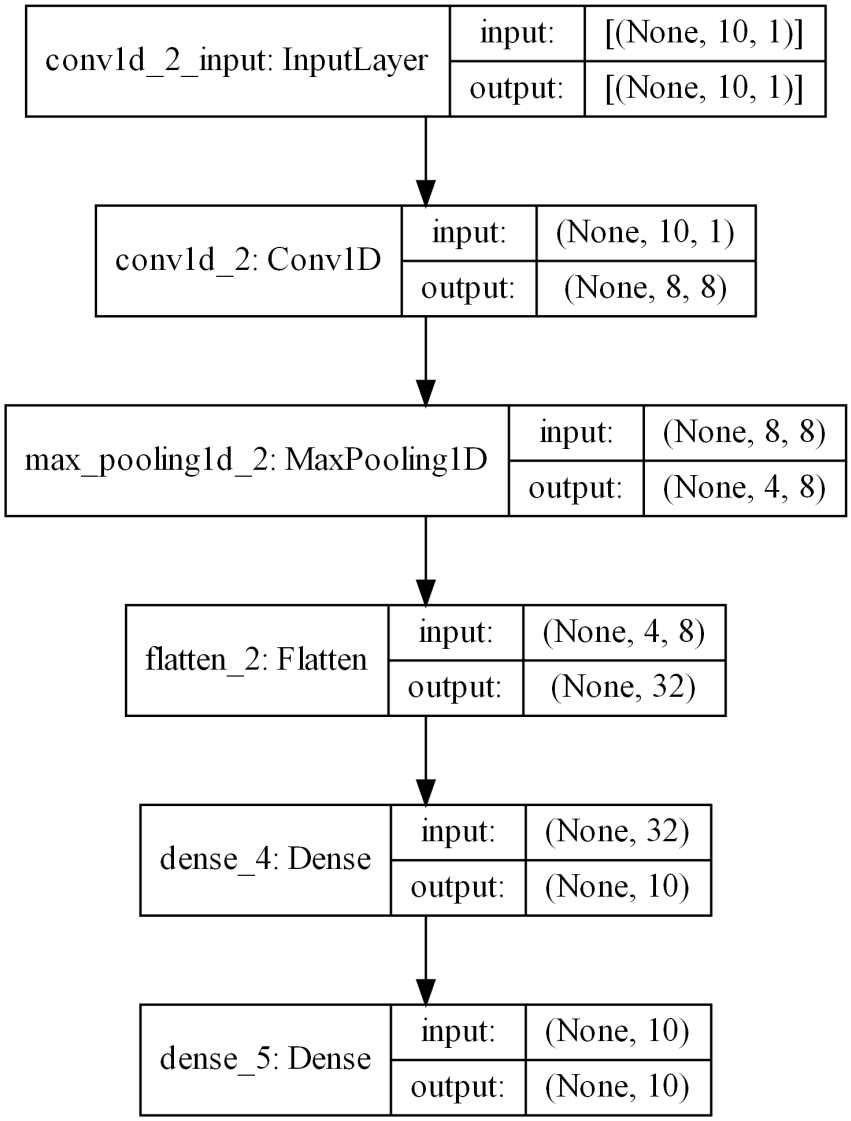


FIGURE 3.2: Model plot of 1D CNN network.

* None: Means that this dimension is variable or dynamic.
* 10: Number of timesteps.
* 1: Number of features, and which for a univariate series data, is one, for one variable.

Vanilla LSTM

The vanilla LSTM is implemented only with a single hidden layer of LSTM units, and a densely connected output layer for the predictions. The implementation is shown in the next code snippet of Listing 6:

```
# define model
model = Sequential()
# adding the input layer and the first hidden layer
model.add(LSTM(224, activation='tanh', input_shape=(n_timesteps,
↪  n_features)))
# adding a second hidden layer
model.add(Dense(10, activation='tanh'))
# adding the output layer
model.add(Dense(n_outputs))
```

LISTING 6: Vanilla LSTM model definition.

For an example with I/O of 10 timesteps and the number of units for the LSTM layer is 32, the built model plot is shown in Figure 3.3.

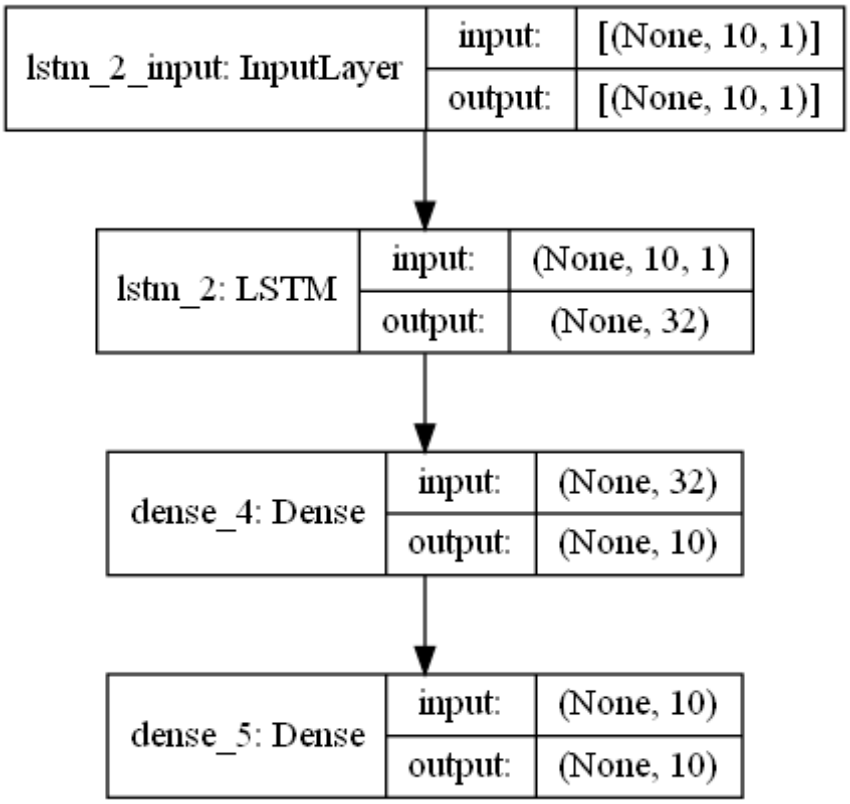


FIGURE 3.3: Model plot of Vanilla LSTM network.

Encoder-decoder LSTM

The encoder layer, made up of the first LSTM layer, reads the input sequence and produces a vector, the size of n-specified units, as an output (one output per unit), that captures features from the input sequence. A RepeatVector layer repeats the internal representation of the input sequence multiple times, once for each timestep in the output sequence. Then the decoder, the second hidden LSTM layer, is defined with the same number of units as the encoder, and it outputs the entire sequence,

not just the output at the end of the sequence as was in the case of the encoder. A fully connected layer is then used to interpret each timestep in the output sequence before the final output layer.

It should be noted that the output layer predicts a single step in the output sequence, which means that the same layers are applied to each step in the output sequence, resulting in the same fully connected layer and output layer being used to process each time step provided by the decoder. This is achieved by wrapping the fully connected interpretation layer and the output layer in a TimeDistributed wrapper, that allows the wrapped layers to be used for each time step from the decoder. This allows the LSTM decoder to figure out the context required for each step in the output sequence and the wrapped dense layers to interpret each timestep separately, yet reusing the same weights to perform the interpretation. The implementation of the network with Keras is shown in the next lines of code of Listing 7:

```
# define model
model = Sequential()
# define encoder
model.add(LSTM(224, activation='tanh', input_shape=(n_timesteps,
↪  n_features)))
# repeat encodering
model.add(RepeatVector(n_outputs))
# define decoder model
model.add(LSTM(160, activation='tanh', return_sequences=True))
# define output model
model.add(TimeDistributed(Dense(10, activation='tanh')))
model.add(TimeDistributed(Dense(1)))
```

LISTING 7: Encoder-decoder LSTM model definition.

An example of the plotted model with number of I/O equal to 10, and the number of units for the two LSTM layers is 32, is shown in Figure 3.4.

CNN-LSTM

The encoder is made up of a first convolutional layer that reads across the input sequence and projects the results onto feature maps. The second convolutional layer reads the input sequences with a kernel size of three timesteps and performs the same operation on the feature maps (of size 64) previously created, attempting to amplify any salient features. The max pooling layer simplifies the feature maps by keeping 1/4 of the values with the largest (max) signal. The distilled feature maps after the pooling layer are then flattened into one long vector that can then be used as input for the decoder. The decoder is the same as was described for the encoder-decoder model, where a TimeDistributed wrapper is also used. The Keras implementation of the model is shown in Listing 8:

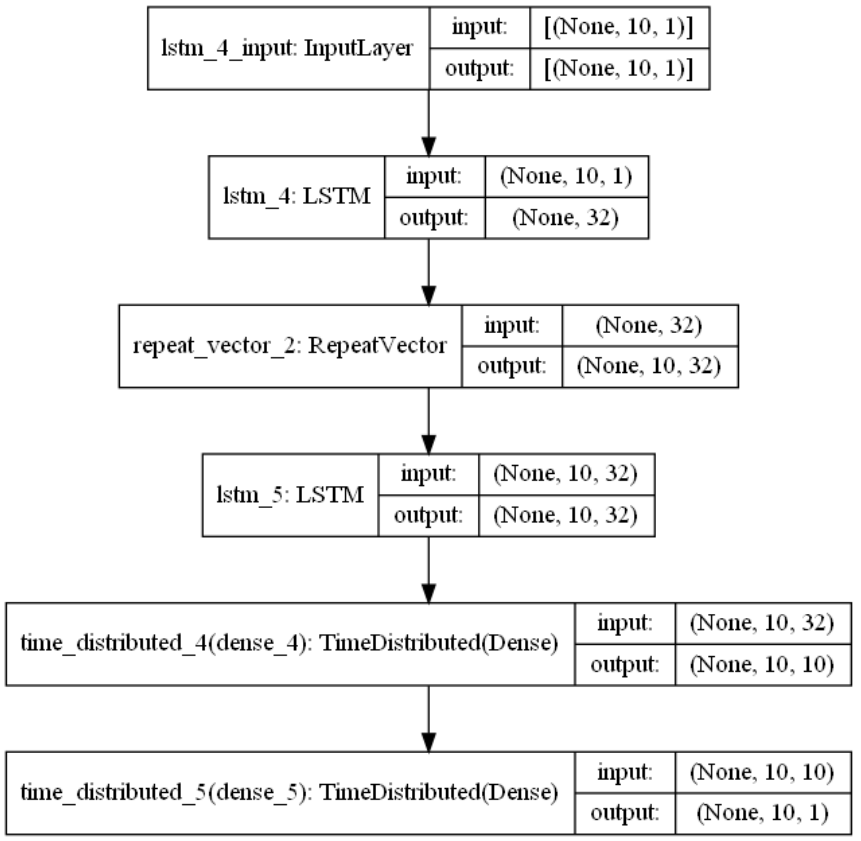


FIGURE 3.4: Model plot of Encoder-decoder LSTM network.

```
# define model
model = Sequential()
# define cnn input model
model.add(Conv1D(filters=8, kernel_size=3, activation='relu',
 ↪  input_shape=(n_timesteps, n_features)))
model.add(Conv1D(filters=8, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
# define decoder model
model.add(RepeatVector(n_outputs))
model.add(LSTM(224, activation='tanh', return_sequences=True))
model.add(TimeDistributed(Dense(10, activation='tanh')))
model.add(TimeDistributed(Dense(1)))
```

LISTING 8: CNN-LSTM model definition.

An example of the plotted model with number of I/O equal to 10, number of filters for the convolution layers equal to 8, and the number of units for the LSTM layer is 32, is shown in Figure 3.5.

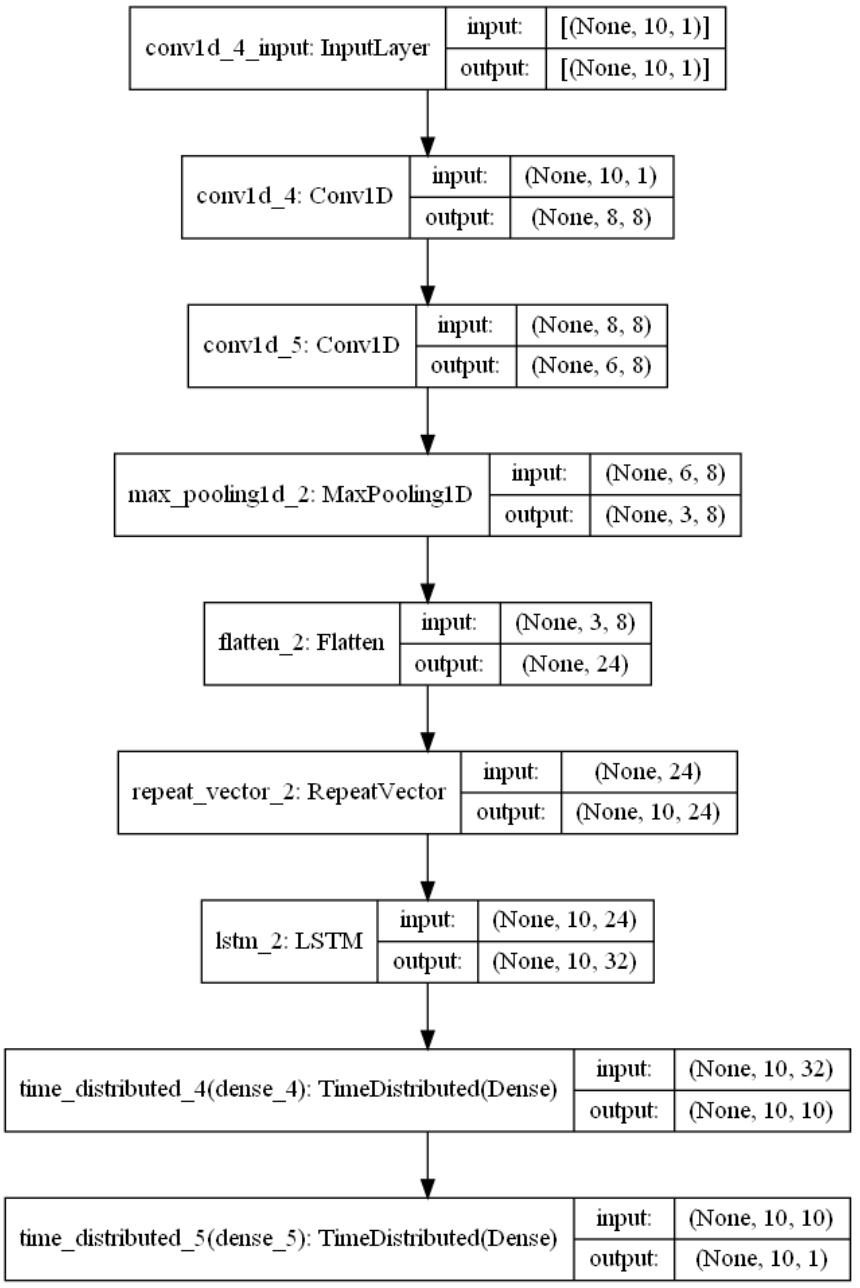


FIGURE 3.5: Model plot of CNN-LSTM network.

ConvLSTM

The ConvLSTM2D class of the Keras library that supports the ConvLSTM model for 2D data is reconfigured and adapted to receive univariate 1D input data. The generated feature maps are flattened before they are repeated and decoded with a LSTM layer. A TimeDistributed wrapper is then used for both the dense interpretation layer and output layer. This is implemented in Listing 9:

```
# define model
model = Sequential()
# define convlstm encoder input model
model.add(ConvLSTM2D(filters=8, kernel_size=(1, 3), activation='relu',
↪  input_shape=(n_steps, 1, n_length, n_features)))
model.add(Flatten())
model.add(RepeatVector(n_outputs))
model.add(LSTM(224, activation='tanh', return_sequences=True))
model.add(TimeDistributed(Dense(10, activation='tanh')))
model.add(TimeDistributed(Dense(1)))
```

LISTING 9: ConvLSTM model definition.

An example of the plotted model, with number of I/O equal to 10, is shown in Figure 3.6. In this example, the number of filters for the ConvLSTM2D layer is 8, and the number of units for the LSTM layer is 32.

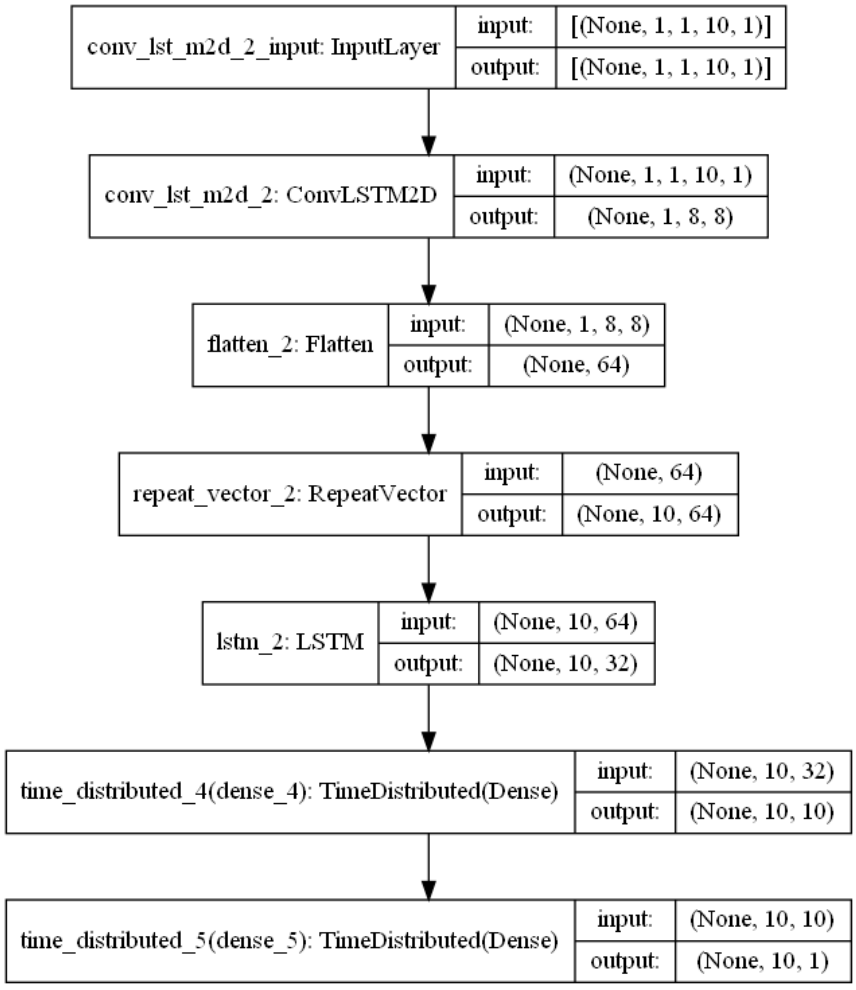


FIGURE 3.6: Model plot of ConvLSTM network.

3.3.3 Network compilation

Each network model is required to be compiled, i.e., configured, after being modelled, and this is done using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error *mse* loss function.

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. Advantages of this method include being computationally efficient and requiring little memory, invariant to diagonal rescaling of gradients, and being well suited for large data/parameter problems [53].

The *mse* loss function is a regression loss function that computes the mean of squares of errors between target variables/labels and predictions. In Keras, this is defined as:

$$loss = square(y_{true} - y_{pred}) \tag{3.2}$$

where y_{true} is the true target value and y_{pred}, the predicted value. Given that the result of the differences is squared, *mse* result is always positive regardless of the sign of the true and predicted values, but it also means that larger mistakes increase by a higher margin the error present than smaller mistakes. This ultimately results in network models being punished for making larger mistakes than smaller ones. A perfect prediction would give a result of 0.

With Keras, the compilation is run with the following code line shown in Listing 10:

```
model.compile(loss='mse', optimizer='adam')
```

LISTING 10: Compiling built network model.

3.3.4 Network training

We will train the different networks for different number of *filters* and *units*, using the CPU first, and later, using the GPU.

The specifications for the CPU are:

- Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz
- RAM: 32 GB
- Windows 10 Pro, 64-bits

The specifications for the GPU are:

- NVIDIA GeForce GTX 1650 SUPER
- CUDA nucleus: 1280
- Available graphic memory: 20438 MB
- Dedicated video memory: 4096 MB GDDR6

The code below, shown in Listing 11, implements the selection of either the CPU or the GPU, determined by the value the variable *cpu_gpu_select* is set to: 0 for the CPU and 1 for the GPU.

```
# select CPU (0) or GPU (1)
cpu_gpu_select = 1

if cpu_gpu_select == 0:
    # Hide GPU from visible devices. That means only the CPU is
    ↪  available.
    tf.config.set_visible_devices([], 'GPU')

else:
    # select GPU
    os.environ["CUDA_VISIBLE_DEVICES"] = "0"
```

LISTING 11: Selection of the CPU or GPU for network training and prediction.

75% of the input data is used for training the network and the remaining 25% is used for prediction. The modelled NNs are trained by running the following line of code shown in Listing 12.

```
model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size,
↪  verbose=0)
```

LISTING 12: Model fitting/training.

The *fit()* method trains the model by slicing the data into "batches" of size *batch_size* and repeatedly iterating over the entire dataset for a fixed number of epochs, given by *epochs*. An epoch is given as one full training cycle or iterations on the entire training dataset. The network weights are updated after every time a slice of the data, of size batch_size, is used to train the model.

For 1D CNN network, *epochs* and *batch_size* are set to 20, and 16, respectively. For the rest of the LSTM network models, their values are set to 50 (epoch) and 16 (batch_size).

3.4 Network model evaluation

After the networks are fitted/trained, we can now use it to make predictions and the validation method used to evaluate the network models is the walk-forward validation.

3.4.1 Walk-forward validation

With the walk-forward validation method of evaluating network models that respects the temporal ordering of observations [51], a given network model is provided the actual data (current timesteps) as the basis for making a prediction on future timesteps. For example, if a model is required to make a prediction of 10 future timesteps, then 10 timesteps data, representing the present time, are provided. For the next 10 future timesteps, the previous 10 current timesteps, in addition with the predicted 10 timesteps, are used to make the prediction. This example is represented in Table 3.3.

TABLE 3.3: Example of walk-forward validation concept.

Input	Predict
[1-10]	[11-20]
[1-10] + [11-20]	[21-30]
[1-10] + [11-20] + [21-30]	[31-40]
...	...

1D CNN

This approach is implemented for a 1D CNN model, in Python, with the *evaluate_model_cnn* method described in the following code of Listing 13:

```
def evaluate_model_cnn(train, test, n_input, n_output, folder_path,
↪  name_model):
    # fit model
    model = build_model(train, n_input, n_output, folder_path,
    ↪  name_model)
    # history is a list of "n_input-sized" samples
    history = [x for x in train]
    # walk - forward validation over each "n_input-sized" samples
    predictions = list()
    for i in range(len(test)):
        # predict the samples of size n_output
        yhat_sequence = forecast(model, history, n_input)
        # store the predictions
        predictions.append(yhat_sequence)
        # get real observation and add to history for predicting the
        ↪  next samples
        history.append(test[i, :])
    # evaluate predictions of each sample
    predictions = array(predictions)
    score, scores = evaluate_forecasts(test[:, :, 0], predictions)

    # reshape the arrays into an easily readable format
    predictions = predictions.reshape((predictions.shape[0] *
    ↪  predictions.shape[1], test.shape[2]))
    original = test[:, :, 0]
    original = original.reshape((original.shape[0] * original.shape[1],
    ↪  test.shape[2]))

    return score, scores, original, predictions
```

LISTING 13: Walk-forward validation for 1D CNN network model.

The train and test datasets, shaped in samples of sizes *n_output*, are provided to the functions as arguments. An additional argument, *n_input*, is provided that is used to define the number of prior observations that the model will use as input in order to make a prediction. Within the method, two functions are called:

build_model(), that builds, compiles and trains the network model, and *forecast()*, that uses the model to make forecasts for each new timestep of size *n_output*.

Previous timestep observations are stored in a list called *history*.

LSTM models: Vanilla, Encoder-decoder, CNN-LSTM

In the case of the LSTM models, the method used for the evaluation is described depending on the LSTM network model being evaluated. Vanilla LSTM, Encoder-decoder LSTM, and CNN-LSTM make use of the same method, *evaluate_model_lstm*, described below in Listing 14.

```
def evaluate_model_lstm(train, test, n_input, n_output, lstm_model,
↪  folder_path, name_model):

    if lstm_model == 1:
        # fit model
        model = build_model_vanilla(train, n_input, n_output,
        ↪  folder_path, name_model)
    elif lstm_model == 2:
        # fit model
        model = build_model_encoder_decoder(train, n_input, n_output,
        ↪  folder_path, name_model)
    elif lstm_model == 3:
        # fit model
        model = build_model_cnn_lstm(train, n_input, n_output,
        ↪  folder_path, name_model)

    # history is a list of "n_input-sized" samples
    history = [x for x in train]
    # walk - forward validation over each "n_input-sized" samples
    predictions = list()
    for i in range(len(test)):
        # predict the samples of size n_output
        yhat_sequence = forecast_lstm(model, history, n_input)
        # store the predictions
        predictions.append(yhat_sequence)
        # get real observation and add to history for predicting the
        ↪  next samples
        history.append(test[i, :])
    # evaluate predictions of each sample
    predictions = array(predictions)

    score, scores = evaluate_forecasts(test[:, :, 0], predictions)

    # plot the original signal versus the predicted signal
    predictions = predictions.reshape((predictions.shape[0] *
    ↪  predictions.shape[1], test.shape[2]))
    original = test[:, :, 0]
    original = original.reshape((original.shape[0] * original.shape[1],
    ↪  test.shape[2]))

    return score, scores, original, predictions
```

LISTING 14: Walk-forward validation for vanilla LSTM, encoder-decoder LSTM, and CNN-LSTM network models.

ConvLSTM

For ConvLSTM model, the method is described below in Listing 15.

```
def evaluate_model_conv_lstm(train, test, n_input, n_output, n_steps,
↪  n_length, folder_path, name_model):

    model = build_model_conv_lstm(train, n_input, n_output, n_steps,
    ↪  n_length, folder_path, name_model)

    # history is a list of "n_input-sized" samples
    history = [x for x in train]
    # walk - forward validation over each "n_input-sized" samples
    predictions = list()
    for i in range(len(test)):
        # predict the samples of size n_output
        yhat_sequence = forecast_conv_lstm(model, history, n_steps,
        ↪  n_length, n_input)
        # store the predictions
        predictions.append(yhat_sequence)
        # get real observation and add to history for predicting the
        ↪  next samples
        history.append(test[i, :])
    # evaluate predictions of each sample
    predictions = array(predictions)

    score, scores = evaluate_forecasts(test[:, :, 0], predictions)

    # plot the original signal versus the predicted signal
    predictions = predictions.reshape((predictions.shape[0] *
    ↪  predictions.shape[1], test.shape[2]))
    original = test[:, :, 0]
    original = original.reshape((original.shape[0] * original.shape[1],
    ↪  test.shape[2]))

    return score, scores, original, predictions
```

LISTING 15: Walk-forward validation for ConvLSTM network model.

3.4.2 Network forecasting

Due to the slow nature of NNs when it comes to training, the preferred usage of the models is to build them once on historical data and to use them to forecast each step of the walk-forward validation. Though the training is generally slow, it should be noted that they are fast to evaluate. The network models are static (i.e. not updated) during their evaluation.

The *forecast()* method for 1D CNN, Vanilla, Encoder-decoder, and CNN-LSTM takes as arguments the model fit on the training dataset (*model*), the history of data observed so far (*history*), and the number of inputs timesteps expected by the model (*n_inputs*).

To be able to make predictions on the test data, the input data is organized into a 3D shape described as: [number of sample, n_output, number of features]. In our case, both the number of samples at a given time, and the number of features in our dataset is 1. So we will always have: [1, n_output, 1]. A prediction of the

future timesteps is made by using the fit model and the input data to call the *predict()* method.

The implemented *forecast()* method for 1D CNN, Vanilla, Encoder-decoder, and CNN-LSTM models is shown below in Listing 16.

```
def forecast(model, history, n_input):
    # flatten data
    data = array(history)
    data = data.reshape(data.shape[0]*data.shape[1], 1)
    # retrieve last observations for input data
    input_x = data[-n_input:, 0]
    # reshape into [1, n_input, 1]
    input_x = input_x.reshape((1, len(input_x), 1))
    # forecast the next sample
    yhat = model.predict(input_x, verbose=0)
    # only the forecast vector is needed
    yhat = yhat[0]
    return yhat
```

LISTING 16: Function for making a multi-step forecast with 1D CNN, Vanilla, Encoder-decoder, and CNN-LSTM network models.

In the case of ConvLSTM, this method, in addition to the three previous arguments, receives two more arguments: *n_steps*, that describes the number of subsequences, and *n_length*, that describes the length of each subsequence. This implementation is shown below in Listing 17.

```
def forecast_conv_lstm(model, history, n_steps, n_length, n_input):
    # flatten data
    data = array(history)
    data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
    # retrieve last observations for input data
    input_x = data[-n_input:, 0]
    # reshape into [samples, timesteps, rows, cols, channels]
    input_x = input_x.reshape((1, n_steps, 1, n_length, 1))
    # forecast the next sample
    yhat = model.predict(input_x, verbose=0)
    # only the forecast vector is needed
    yhat = yhat[0]
    return yhat
```

LISTING 17: Function for making a multi-step forecast with a ConvLSTM network model.

3.4.3 Forecast evaluation

The *n_output*-sized timesteps sample of the forecast are evaluated individually. This evaluation comprises of comparing the real timesteps with the predicted timesteps and getting the error between the two. Different methods exist to achieve this, with the two most common being Mean Absolute Error (MAE) and RMSE. We make use

of RMSE, which, apart from being more punishing of forecast errors, is good match for the *mean square error* loss function previously chosen in the network models, to calculate a score that represents the RMSE across all forecast timesteps. The *evaluate_forecasts* function implements this forecast evaluation and it is described below in Listing 18.

```
# evaluate one or more forecasts against expected values
def evaluate_forecasts(actual, predicted):
    scores = list()
    # calculate an RMSE score for each sample
    for i in range(actual.shape[1]):
        # calculate mse
        mse = mean_squared_error(actual[:, i], predicted[:, i])
        # calculate rmse
        rmse = sqrt(mse)
        # store
        scores.append(rmse)
    # calculate overall RMSE
    s = 0
    for row in range(actual.shape[0]):
        for col in range(actual.shape[1]):
            s += (actual[row, col] - predicted[row, col]) ** 2
    score = sqrt(s/(actual.shape[0] * actual.shape[1]))
    return score, scores
```

LISTING 18: Function for evaluating forecasts/predictions by network models.

To summarize/list the scores, we call the function *summarize_scores()*, described below in Listing 19.

```
# summarize scores
def summarize_scores(name, score, scores):
    # s_scores = ', '.join(['%.1f' % s for s in scores])
    # print(scores)
    # print('%s: [%.3f] %s' % (name, score, s_scores))
    print('%s: [%.3f]' % (name, score))
```

LISTING 19: Function to summarize network model performance.

Note: Due to the large size of timesteps in the dataset, only the overall score across all forecast timesteps are printed. The scores of each individual timestep is not printed and it is commented out.

Chapter 4

Results and conclusions

4.1 Results of network fitting and predictions with network models

Given the stochastic nature of the algorithm, where two simultaneous executions are highly likely to produce two different results, i.e., the RMSE values and network execution time are slightly different, the network is run 3 times and the run with the minimum RMSE value is chosen. Another possible solution to this would have been to compute the average of the results of various runs for a single network model. The five models are trained on the CPU and GPU and the simulation time is gotten after the training and prediction are done.

4.1.1 Training on CPU for I/O = 10

In the first case, 10 timesteps were selected as input and it was also specified that the network predict 10 future timesteps at each instant of time.

The obtained results are shown in Table 4.1, where the ANNs under study are compared in terms accuracy of the prediction – characterized by the Root Mean-Square Error (RMSE), CPU/GPU time required to train/fit each NN model and the subsequent predictions and the number of filters and units. The numbers of filters and units indicated in the table were chose empirically to obtain similar behaviour for all networks. A look at the table shows that, in terms of the accuracy of the network models, they are all similar, with minimal differences in some cases that are considered to be not that influential when comparing one model's accuracy with the other. On the other hand, in terms of CPU time and system complexity, the 1D CNN model presents a better efficiency than the rest. Of the recurrent LSTMs models, the Vanilla model is more efficient than the other three models. The plotted predictions by the different network models can be seen in Appendix A.

4.1.2 Training on GPU for I/O = 10

In the second case where the same number of timesteps as in the first case are selected as input and also predicted, but with the difference that the training is done on the GPU, Table 4.2 compares the result of each of the models.

A look at Table 4.2 shows that when the training/fitting and predictions is done on the GPU, their is an increase in the overall computational time on the 1D CNN, CNN-LSTM, and ConvLSTM model. These three models make use of convolutional layers and as such, it may seem that the increase is due to them. The accuracy of all five models are similar, though it should be noted that the ConvLSTM model has the more accurate than the rest. The plotted predictions by the different network models can be seen in Appendix A.

TABLE 4.1: Comparison of NNs under study for 10 I/O datasets with training done on CPU.

Network models	**Filters**	**8**	**16**	**24**
1D CNN	CPU Time (s)	19.01	19.31	18.96
	RMSE (%)	1.3	1.4	1.5
Vanilla LSTM	**Units**	**32**	**160**	**224**
	CPU Time (s)	98.48	147.71	166.99
	RMSE (%)	1.3	1.3	1.3
Encoder-decoder LSTM	CPU Time (s)	179.14	302.26	404.89
	RMSE (%)	1.2	1.3	1.3
CNN-LSTM	CPU Time (s)	112.17	158.76	190.94
	RMSE (%)	1.3	1.4	1.6
ConvLSTM	CPU Time (s)	134.71	182.91	225.88
	RMSE (%)	1.3	1.2	1.2

TABLE 4.2: Comparison of NNs under study for 10 I/O datasets with training done on GPU.

Network models	**Filters**	**8**	**16**	**24**
1D CNN	CPU Time (s)	50.58	47.77	49.54
	RMSE (%)	1.3	1.3	1.3
Vanilla LSTM	**Units**	**32**	**160**	**224**
	CPU Time (s)	147.71	128.03	130.11
	RMSE (%)	1.3	1.4	1.3
Encoder-decoder LSTM	CPU Time (s)	176.10	193.17	210.00
	RMSE (%)	1.3	1.3	1.3
CNN-LSTM	CPU Time (s)	187.80	185.20	195.64
	RMSE (%)	1.4	1.4	1.4
ConvLSTM	CPU Time (s)	576.82	592.42	596.66
	RMSE (%)	1.2	1.2	1.2

4.2 Conclusions

We recapitulate everything we have done and suggest future implementations that can be carried out by improving the project or using part of it.

4.2.1 Results conclusions

Drawing conclusions from the results presented in both Table 4.1 and Table 4.2, different solutions can be implemented to solve the time-series forecasting problem. The five evaluated models offer similar accuracies, characterized by their RMSE values, in their predictions and as such, for any proposed solution, involving any of the models, its accuracy would slightly be different from other possible solutions based on other models. In this case, a preferred solution would be to choose the ConvLSTM network model as it, slightly, has the best accuracy, both for when the training and fitting are carried out with CPU or the GPU.

Comparing the computational time (combined training and fitting time) of each network model shows much bigger differences than in the case of comparing their accuracies. Generally, the computational time, when carried out with the CPU than the GPU, is smaller for models that include convolutional layers (1D CNN, CNN-LSTM, and ConvLSTM), and as such, the preferred solution would be to choose the 1D CNN that has the less computational time of the five models. The less computational time the model has, the less resources are consumed.

4.2.2 Compendium

Our aim was to explore different architectures of NNs which can be applied for time series forecasting problems and subsequently be used to predict the future evolution of the radioelectric spectrum for Cognitive-Radio applications. First, in chapter 1, we gave an introduction to what the project entails, the motivation and background behind it and the objectives, short-term and long-term, that we hoped to achieve by completing it. In chapter 2, a brief explanation of the basic concepts that helped to fully understand what we were doing and the theories behind them was given. Extensive research carried out online or with books by different authors was done to be able to explain the concepts and summarize each concept in a concise manner. For every author or idea, we cited, a bibliography reference was provided for a complete reading and to give credit where due.

The implementation part of the project starts with chapter 3, where we explained, in much detail, every step of the project that resulted in completing our aim. this included explaining how the dataset we were using was obtained, the modelling of the different NNs with Keras, their fitting and their evaluation based on the predictions they made. The codes scripts for each part were provided where needed. In chapter 4, the five ANNs under study are compared in terms of their prediction accuracy, characterized by the Root Mean-Square Error (RMSE), CPU/GPU time required to train and fitting/predicting the ANN and the number of filters and units. Also, figures of the model predictions are provided. With our aim of comparing the network models achieved, we proceeded to conclude the project.

4.2.3 Personal learning process

The entire project, from the start to the end, have been a wonderful learning journey, from learning to build neural networks and delving into the world of SDR. These

were things I had no knowledge of or only an idea of their most basic concepts. Also, improving Python skills. Modifying and debugging written codes and seeing that they complied without errors was quite satisfying. At the earlier stages of the project, the challenge was to adapt and restructure our signals to the shape accepted by neural networks. Next came the challenge of finding the right parameter values with which to test the different models. Several research were conducted to help understand the influence these parameters have on the different models and, help my decision making. Credit where due was given to the different authors of the original work that helped in my research. In the training part of the project, patience and focus were needed for both the time it took to train the different iterations of the models and writing down the results. Concerning the organization of the project and writing the report, I learnt how to improve the time I dedicate to the project and keeping my concentration while writing the codes or the report. It is to be noted that the timeline mapped out for the completion of this project was exceeded due to circumstances both in and beyond my control.

4.2.4 Future works and applications

While the project was been carried out, we came upon ideas to improve the network models architecture that we couldn't implement due to the system we had, and the short time frame to complete the project. So, we will list some of these improvements/ideas and future applications for the project.

Network model to predict multilevel signals of multiple channels

A 1D CNN model, that accepts multiple channels as input, was implemented. This network model predicts the channel occupancy of the different channels. An implementation where 4 channels are modelled and predicted is shown in Figure 4.1. A correspoding example of predictions made by the model for each of the channels is shown in Figure 4.2, Figure 4.2, Figure 4.4, and Figure 4.5.

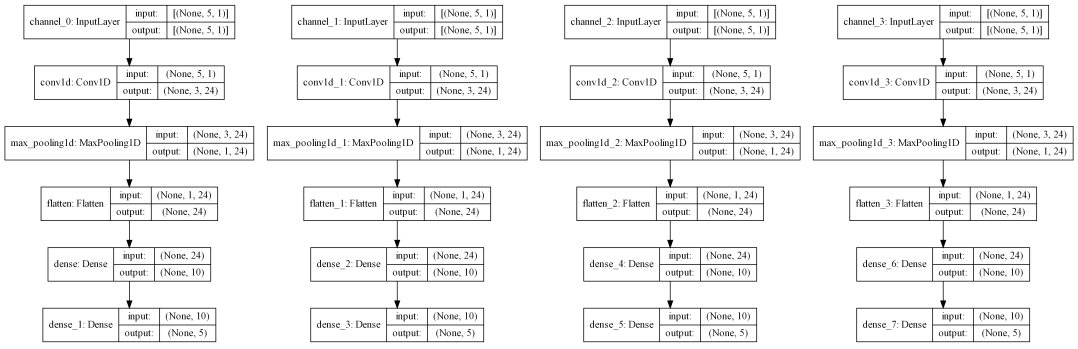


FIGURE 4.1: Multiple channel 1D CNN architecture

The same implementation is being carried out on the other network models seen in the project.

Multilevel signals decision block with multiple thresholds

A decision block that selects the band to be occupied based on multiple thresholds. It receives as input the occupancy signals of several channels which are multilevelled

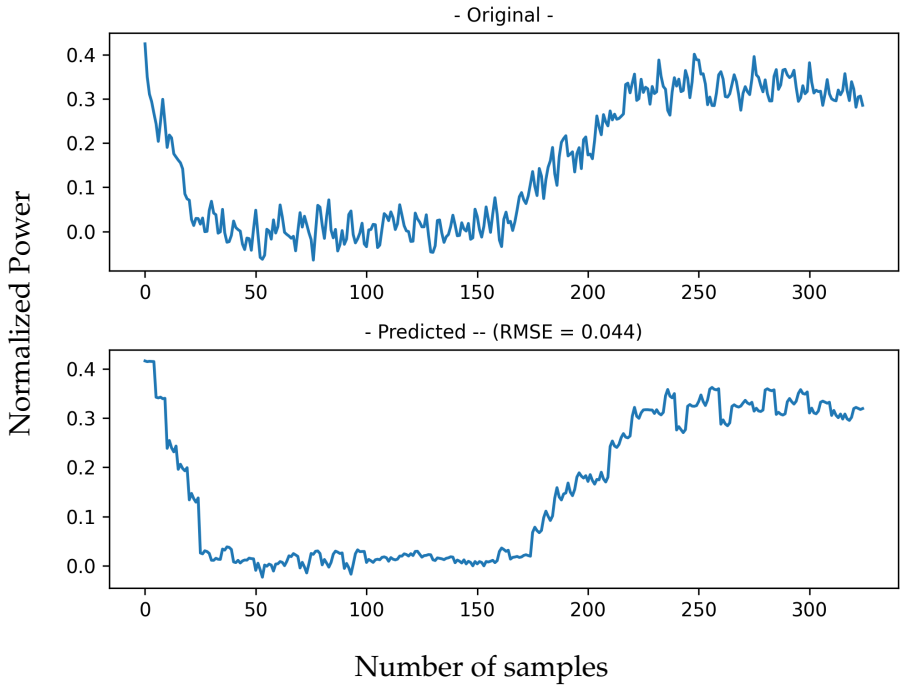


FIGURE 4.2: Prediction for channel 1.

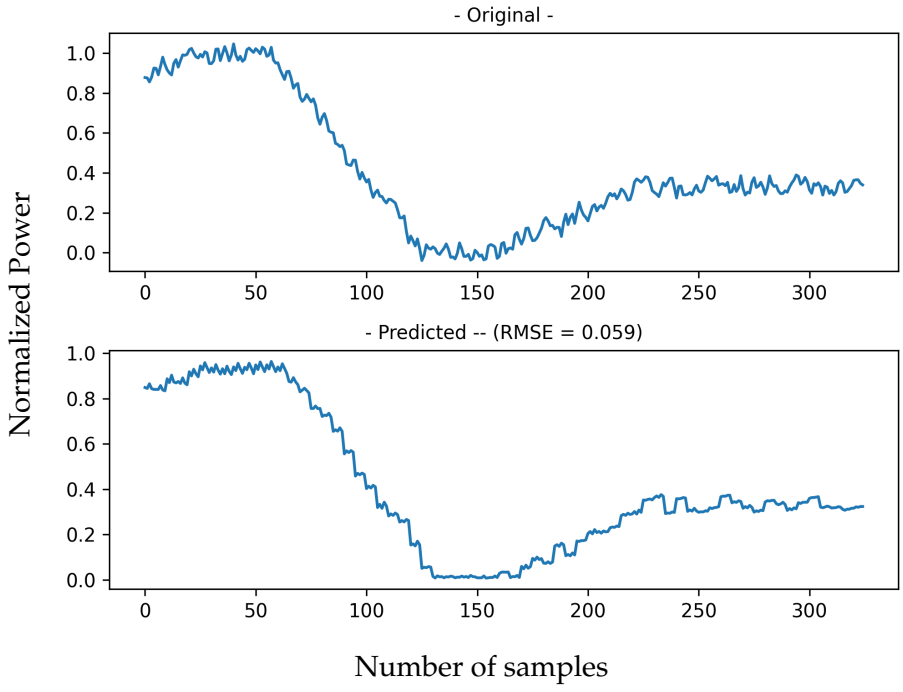


FIGURE 4.3: Prediction for channel 2.

and as such, may require multiple thresholds to be able to make precise decisions. An example of this implementation is shown in Figure 4.6.

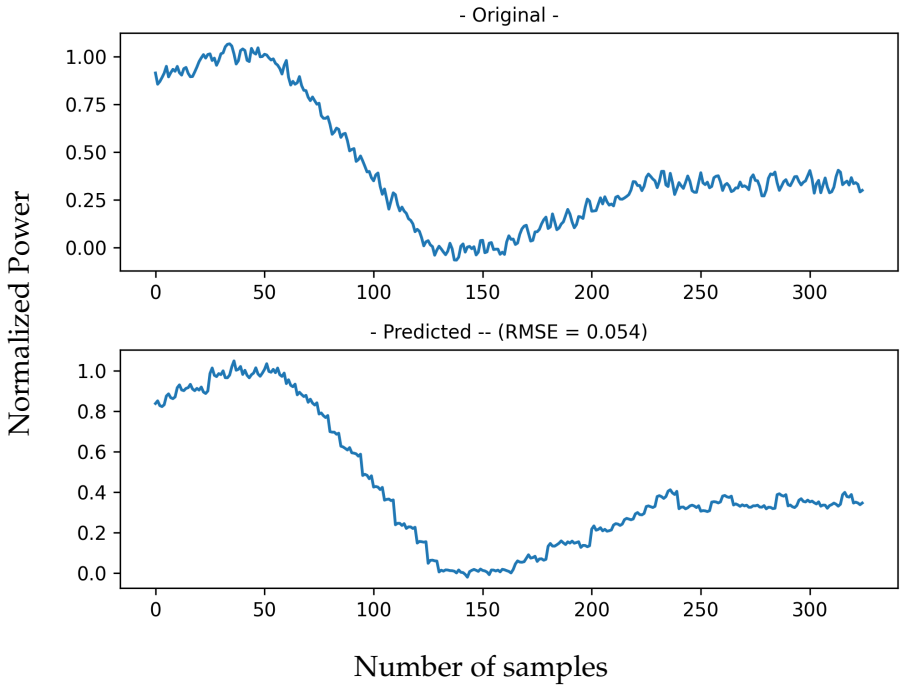


FIGURE 4.4: Prediction for channel 3.

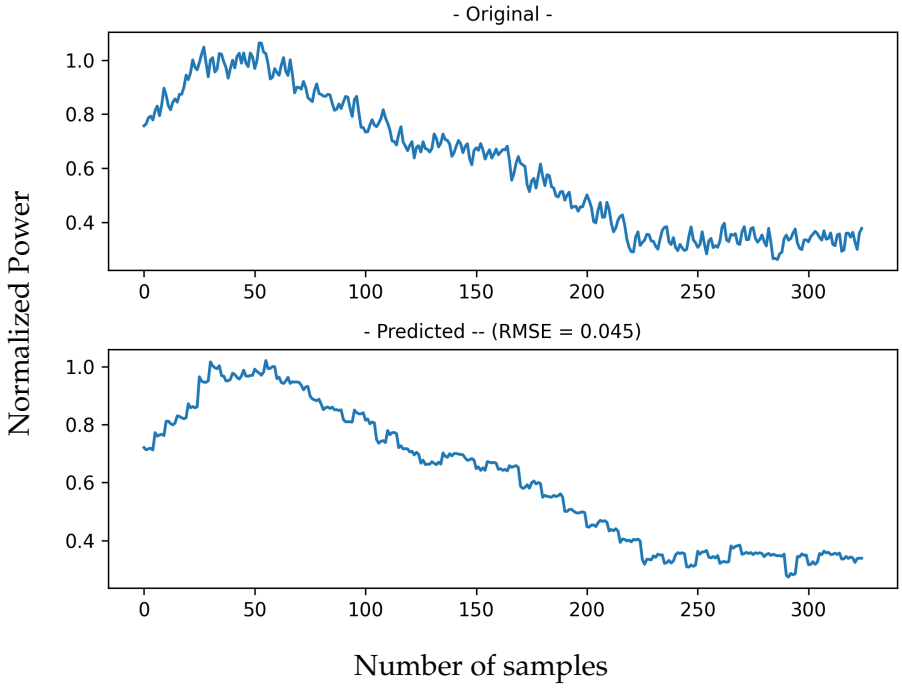


FIGURE 4.5: Prediction for channel 4.

Other applications

- Build the AI module to control the operation of a RF digitizer based on a BP-ΣΔM.

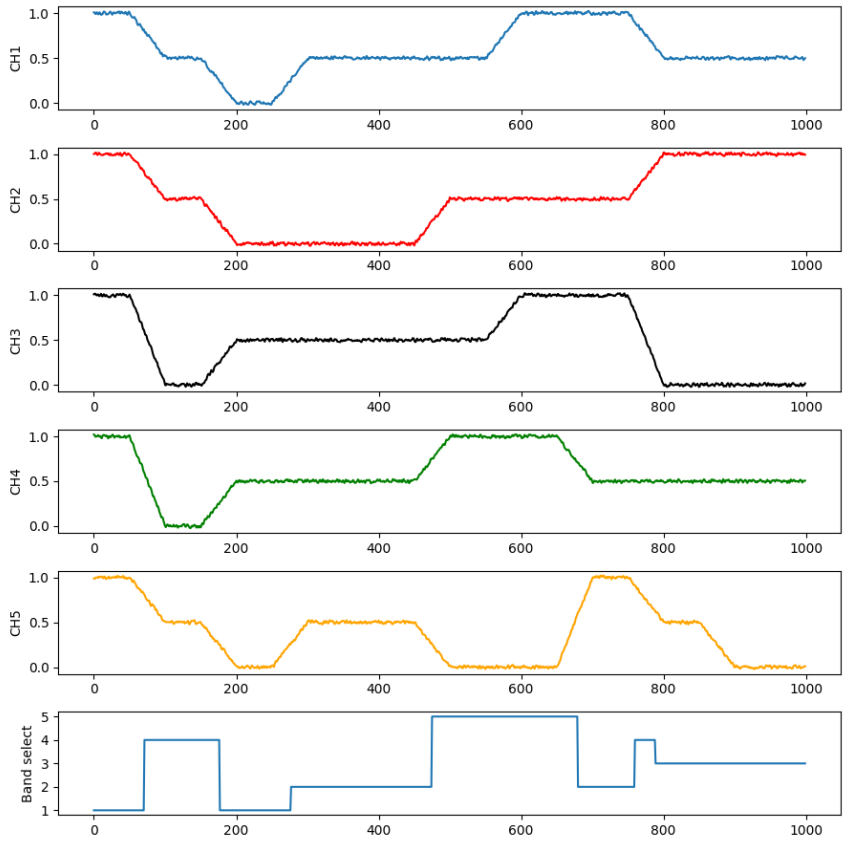


FIGURE 4.6: Evolution of channel occupancy and selection..

Appendix A

Prediction results of the different network models

A.1 Training on CPU for I/O = 10

A.1.1 1D CNN

The plotted prediction results of the 1D CNN model trained on the CPU in comparison with the original dataset, are shown in Figure A.1, Figure A.2, and Figure A.3.

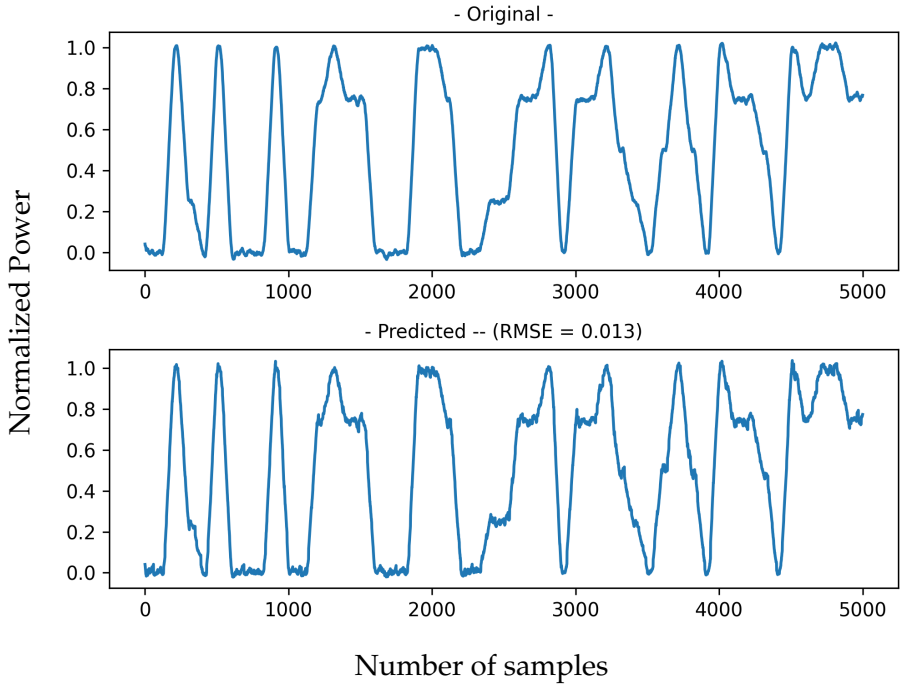


FIGURE A.1: Model of the channel occupancy and prediction of 1D CNN model for filter size of 8.

A.1.2 Vanilla LSTM

The plotted prediction results of the vanilla LSTM model trained on the CPU in comparison with the original dataset, are shown in Figure A.4, Figure A.5, and Figure A.6.

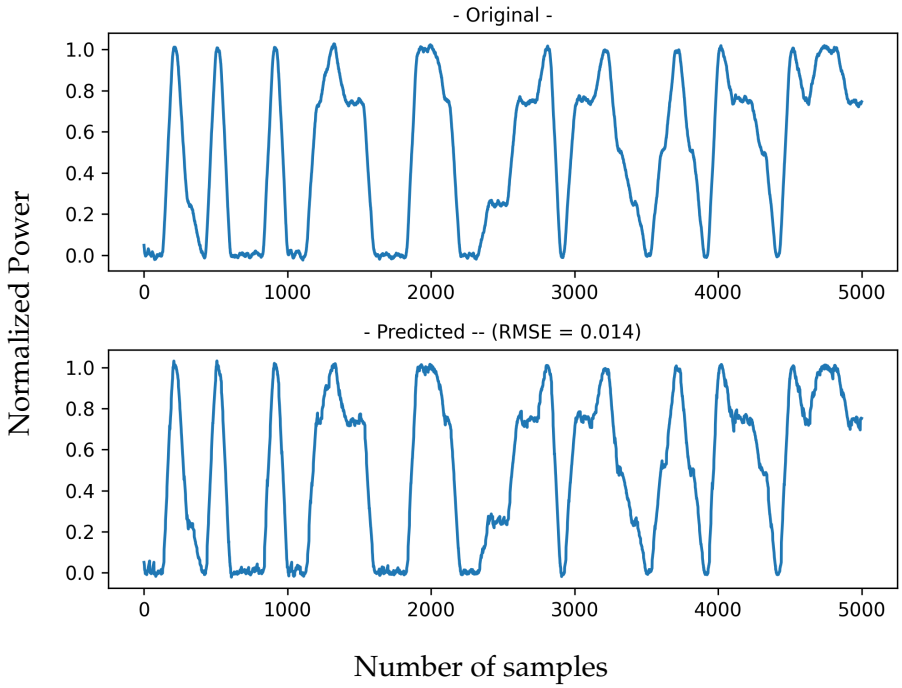


FIGURE A.2: Model of the channel occupancy and prediction of 1D CNN model for filter size of 16.

A.1.3 Encoder-decoder LSTM

The plotted prediction results of the encoder-decoder LSTM model trained on the CPU in comparison with the original dataset, are shown in Figure A.7, Figure A.8, and Figure A.9.

A.1.4 CNN-LSTM

The plotted prediction results of the CNN-LSTM model trained on the CPU in comparison with the original dataset, are shown in Figure A.10, Figure A.11, and Figure A.12.

A.1.5 ConvLSTM

The plotted prediction results of the ConvLSTM model trained on the CPU in comparison with the original dataset, are shown in Figure A.13, Figure A.14, and Figure A.15.

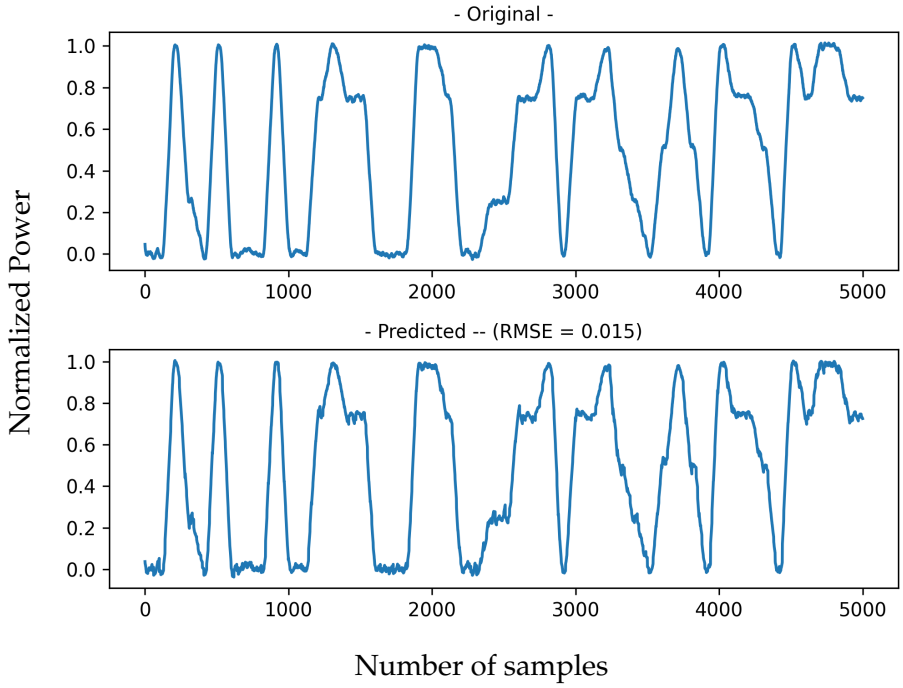


FIGURE A.3: Model of the channel occupancy and prediction of 1D CNN model for filter size of 24.

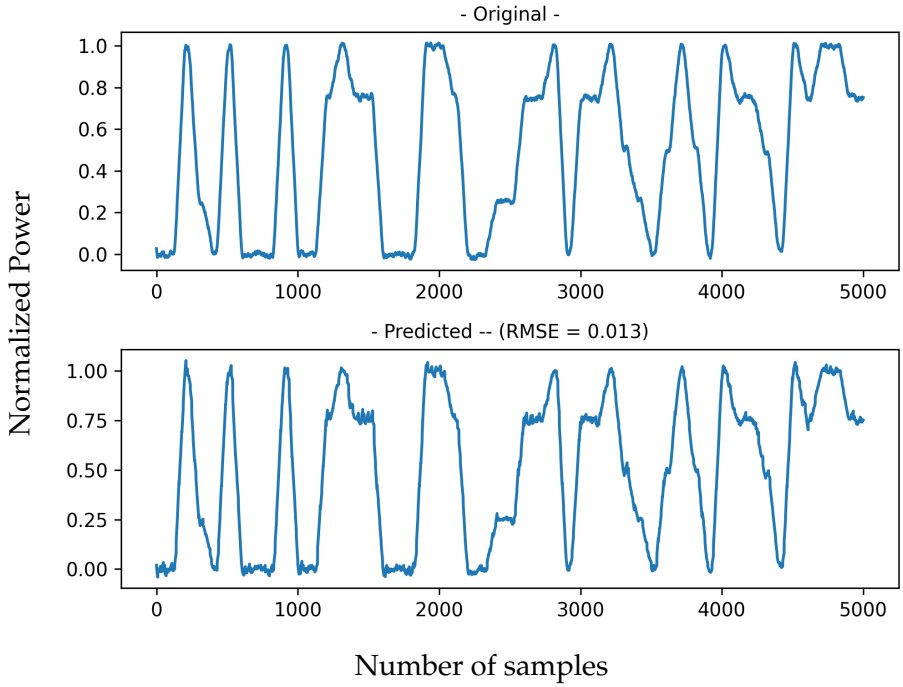


FIGURE A.4: Model of the channel occupancy and prediction of vanilla LSTM model for filter size of 32.

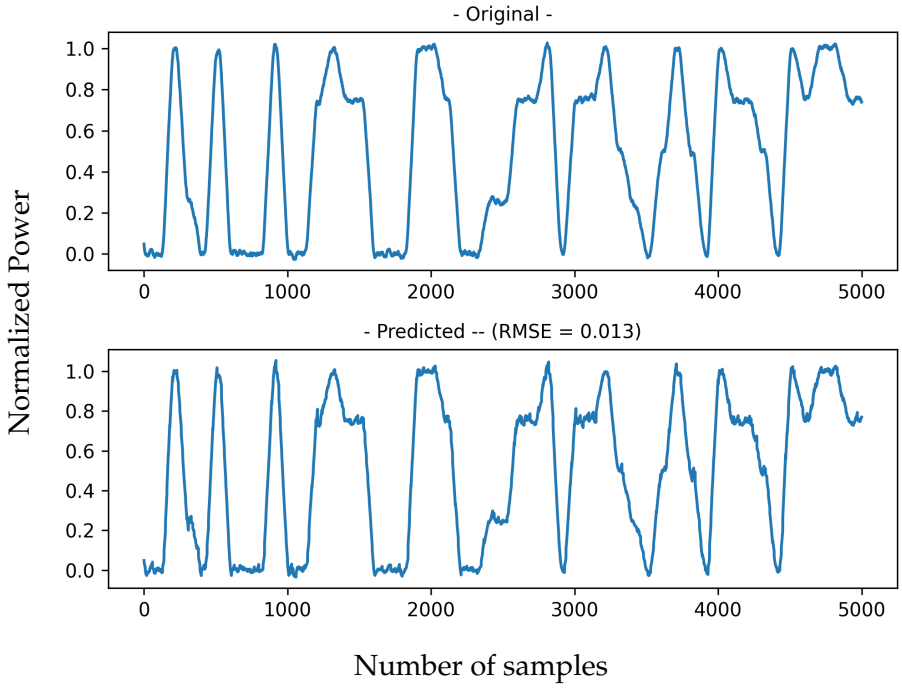


FIGURE A.5: Model of the channel occupancy and prediction of vanilla LSTM model for filter size of 160.

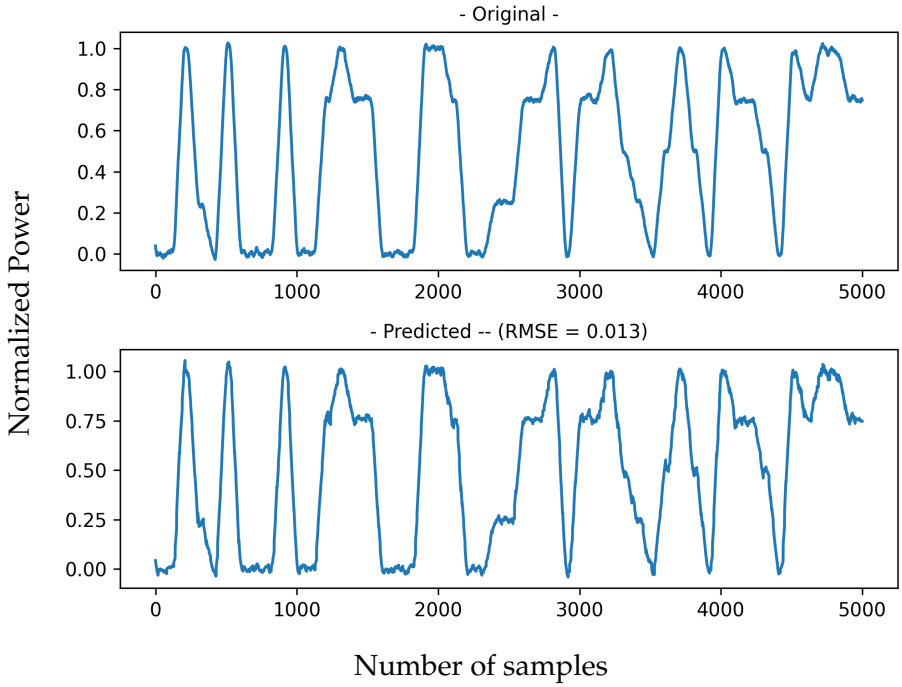


FIGURE A.6: Model of the channel occupancy and prediction of vanilla LSTM model for filter size of 224.

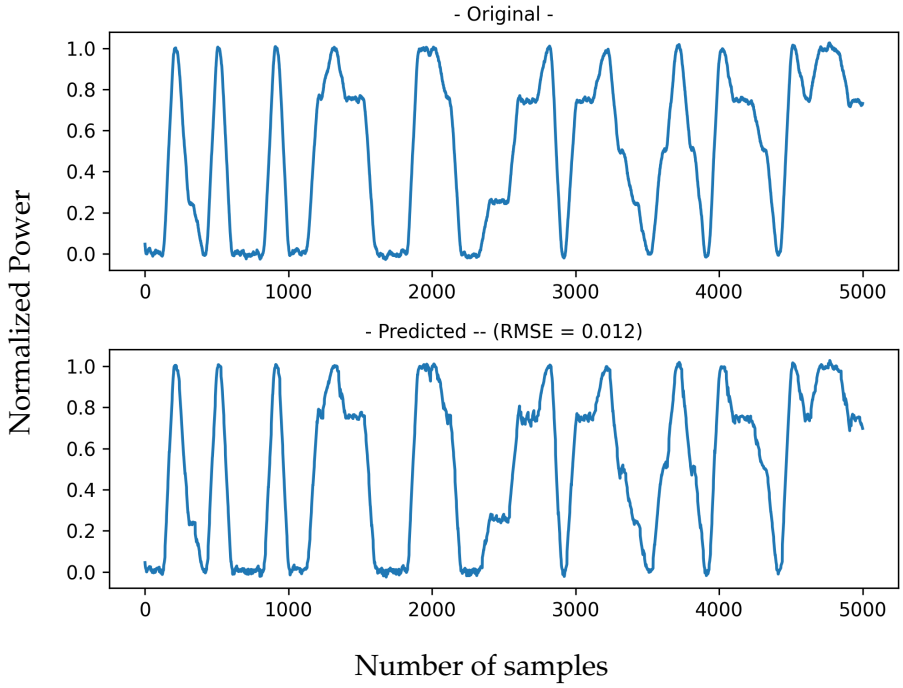


FIGURE A.7: Model of the channel occupancy and prediction of encoder-decoder LSTM model for filter size of 32.

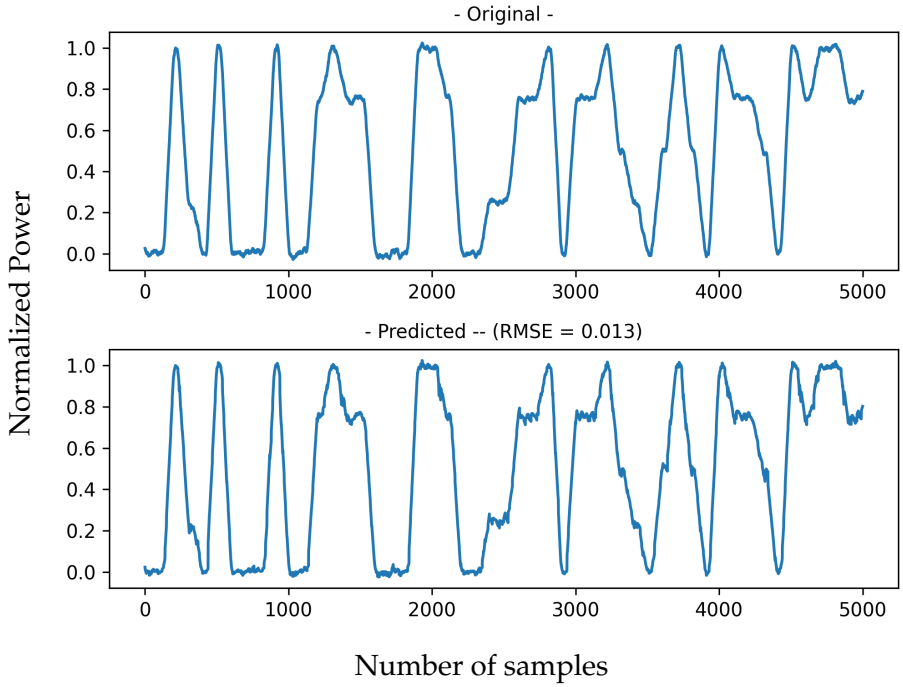


FIGURE A.8: Model of the channel occupancy and prediction of encoder-decoder LSTM model for filter size of 160.

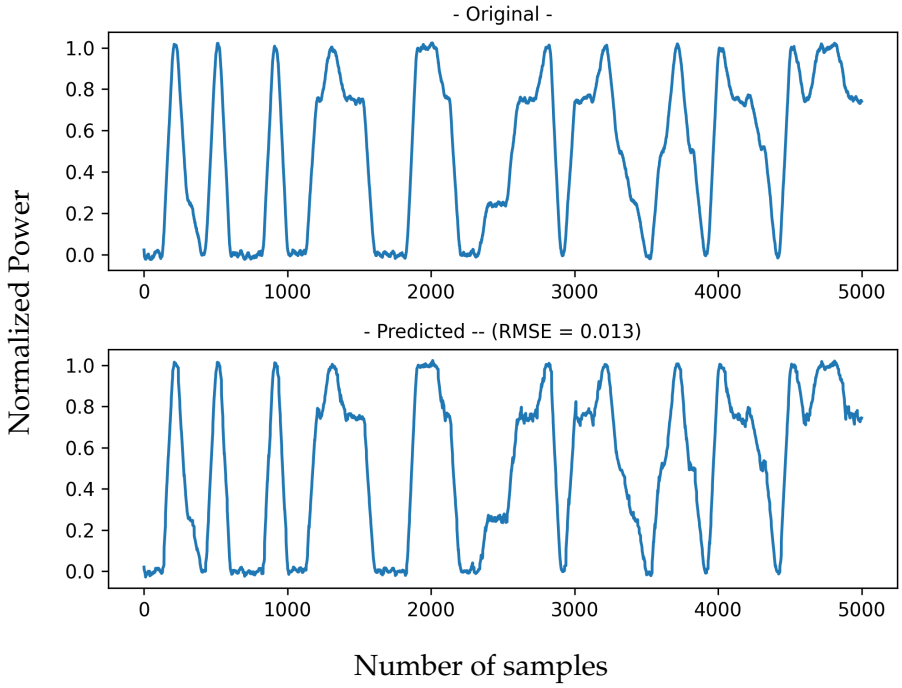


FIGURE A.9: Model of the channel occupancy and prediction of encoder-decoder LSTM model for filter size of 224.

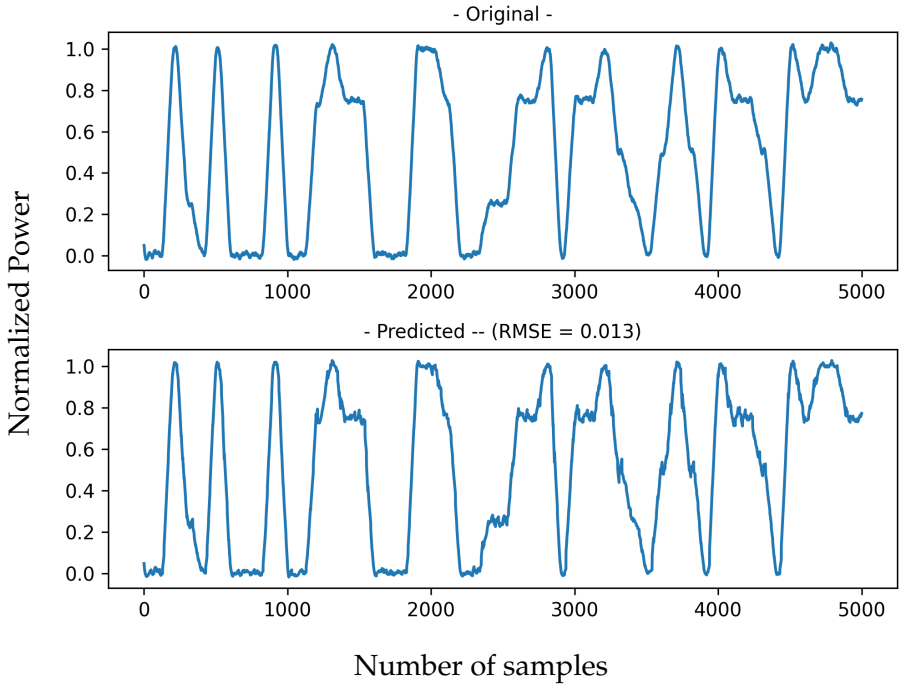


FIGURE A.10: Model of the channel occupancy and prediction of CNN-LSTM model for filter size of 32.

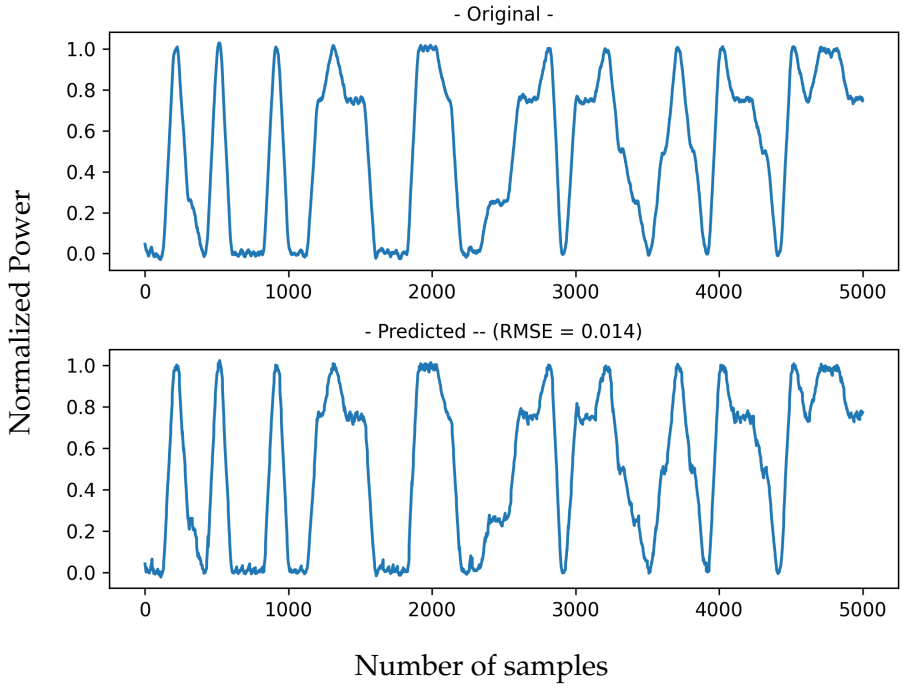


FIGURE A.11: Model of the channel occupancy and prediction of CNN-LSTM model for filter size of 160.

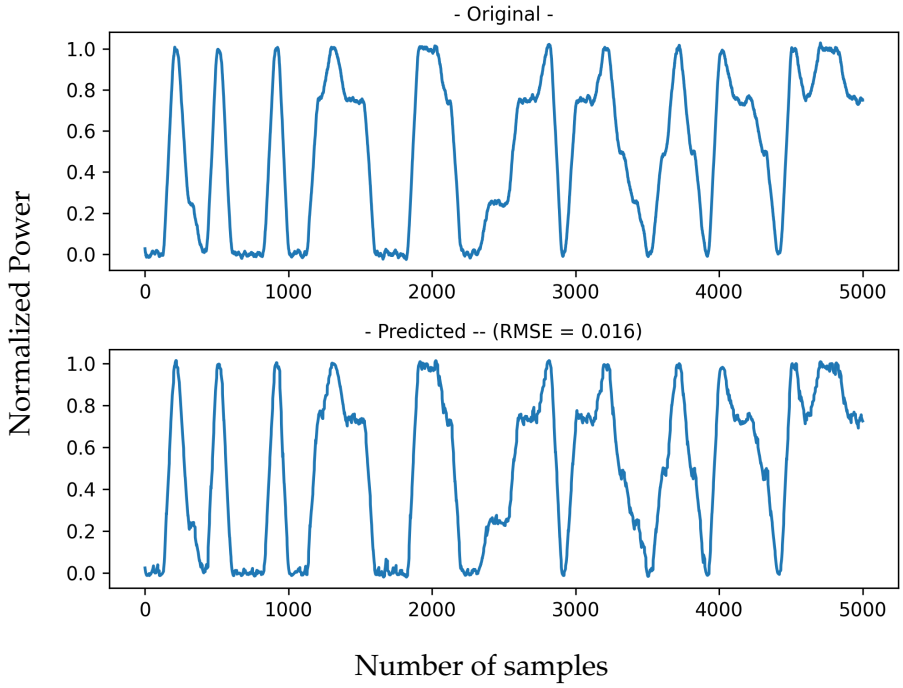


FIGURE A.12: Model of the channel occupancy and prediction of CNN-LSTM model for filter size of 224.

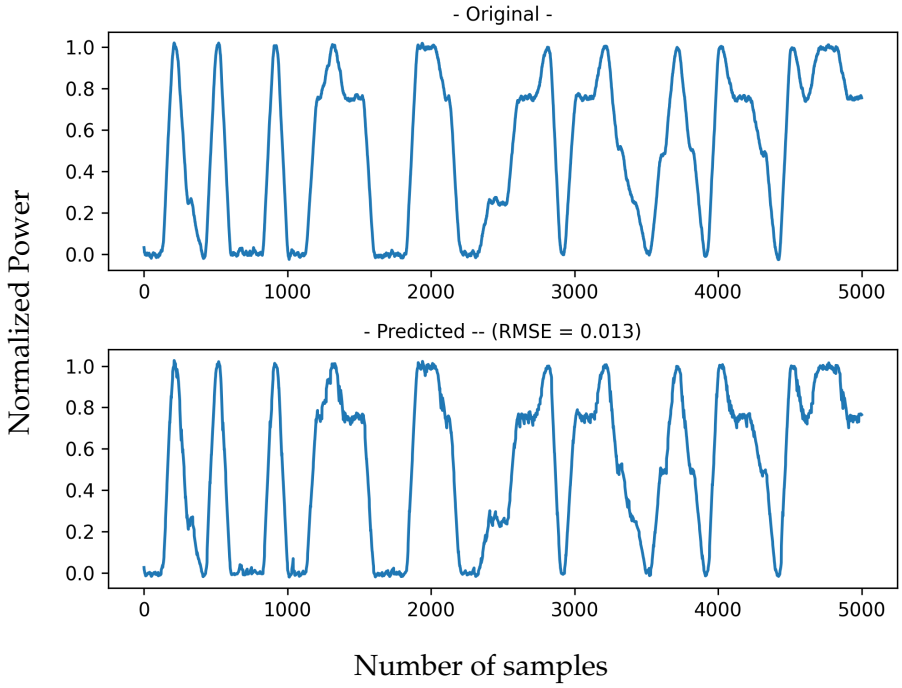


FIGURE A.13: Model of the channel occupancy and prediction of ConvLSTM model for filter size of 32.

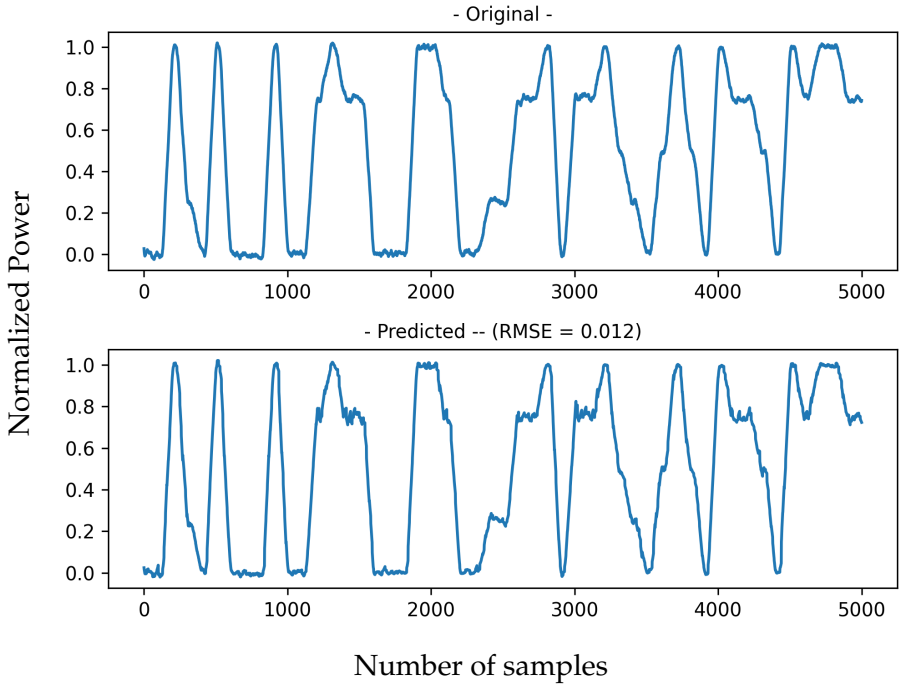


FIGURE A.14: Model of the channel occupancy and prediction of ConvLSTM model for filter size of 160.

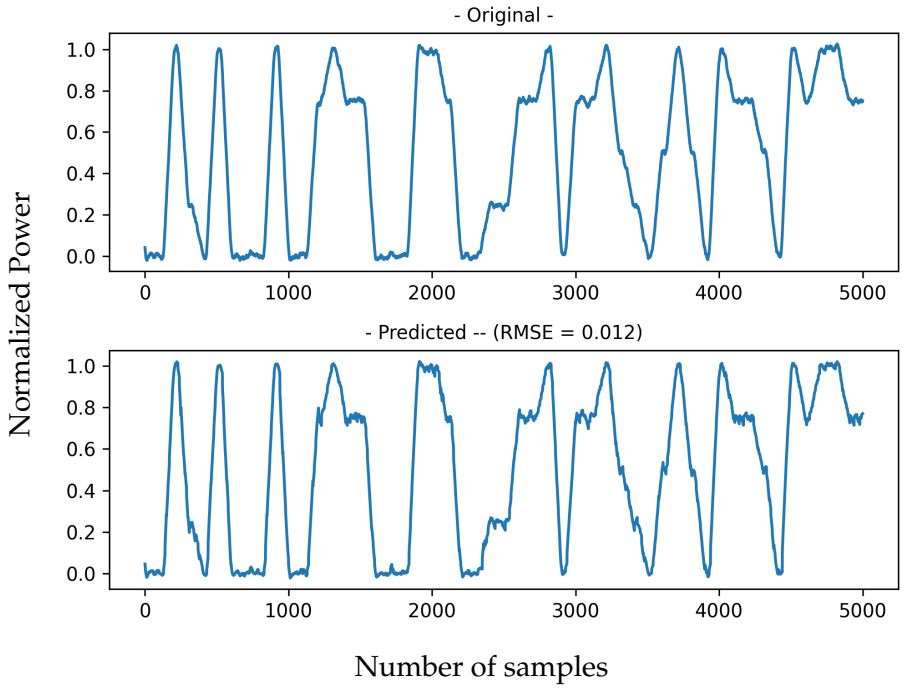


FIGURE A.15: Model of the channel occupancy and prediction of ConvLSTM model for filter size of 224.

Appendix B

Software

B.1 Python - PyCharm IDE

Amongst the most popular computer programming languages, Python is a high-level, general-purpose programming language created by Guido van Rossum and first released in 1991.

Python was conceived in the late 1980s as a successor to the ABC language and since then, 3 major iterations/versions have been released with Python 3 been the latest. More information can be found at: https://www.python.org/

PyCharm is an integrated development environment (IDE) that uses the Python language. With features like code analysis, debugging, testing, built-in console and easier installation of modules/packages makes it a go-to IDE for programmers. More information at: https://www.jetbrains.com/pycharm/

Bibliography

[1] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26–38, 1995. DOI: 10.1109/35.393001.

[2] F. Restuccia and T. Melodia, "Deep learning at the physical layer: System challenges and applications to 5g and beyond," *IEEE Communications Magazine*, vol. 58, pp. 58–64, 2020.

[3] Y. Hua, Z. Zhao, R. Li, X. Chen, Z. Liu, and H. Zhang, "Deep learning with long short-term memory for time series prediction," *IEEE Communications Magazine*, vol. 57, no. 6, pp. 114–119, 2019. DOI: 10.1109/MCOM.2019.1800155.

[4] V. Zuniga, L. Camuñas-Mesa, B. Linares-Barranco, T. Serrano-Gotarredona, and J. Rosa, "Using neural networks for optimum band selection in cognitive-radio systems," Nov. 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294894.

[5] T. Yucek and H. Arslan, "A survey of spectrum sensing algorithms for cognitive radio applications," *IEEE Communications Surveys Tutorials*, vol. 11, no. 1, pp. 116–130, 2009. DOI: 10.1109/SURV.2009.090109.

[6] Wikipedia, the free encyclopedia. "Internet of things." (2021), [Online]. Available: https://en.wikipedia.org/wiki/Internet_of_things.

[7] K.-H. L. Loh, "1.2 fertilizing aiot from roots to leaves," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 15–21. DOI: 10.1109/ISSCC19947.2020.9062950.

[8] M. Liu, "1.1 unleashing the future of innovation," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 9–16. DOI: 10.1109/ISSCC42613.2021.9366060.

[9] F. Salahdine and H. E. Ghazi, "A real time spectrum scanning technique based on compressive sensing for cognitive radio networks," *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pp. 506–511, 2017.

[10] F. Salahdine, "Spectrum sensing techniques for cognitive radio networks," in Oct. 2017.

[11] Y. Molina-Tenorio, A. Prieto-Guerrero, R. Aguilar-Gonzalez, and S. Ruiz-Boqué, "Machine learning techniques applied to multiband spectrum sensing in cognitive radios," *Sensors*, vol. 19, no. 21, 2019, ISSN: 1424-8220. DOI: 10.3390/s19214715. [Online]. Available: https://www.mdpi.com/1424-8220/19/21/4715.

[12] N. Armi, M. Z. Yusoff, and N. M. Saad, *Decentralized cooperative user in opportunistic spectrum access system*, Undetermined. DOI: 10.1109/ICIAS.2012.6306183.

[13] N. Armi, M. Z. Yusoff, and N. M. Saad, "Cooperative spectrum sensing in decentralized cognitive radio system," *Eurocon 2013*, pp. 113–118, 2013.

[14] J. Mitola and G. Q. Maguire, "Cognitive radio: Making software radios more personal," *IEEE Wirel. Commun.*, vol. 6, pp. 13–18, 1999.

[15] A. Morgado, R. Del Río, and J. De la Rosa, *Nanometer CMOS Sigma-Delta Modulators for Software Defined Radio*. Jan. 2012, pp. 1–288, ISBN: 978-1-4614-0036-3. DOI: 10.1007/978-1-4614-0037-0.

[16] J. Mitola, "Cognitive radio an integrated agent architecture for software defined radio," 2000.

[17] X. Zhang, X. Liu, H. Samani, and B. Jalaian, "Cooperative spectrum sensing in cognitive wireless sensor networks," *International Journal of Distributed Sensor Networks*, vol. 11, no. 8, p. 170 695, 2015. DOI: 10.1155/2015/170695. eprint: https://doi.org/10.1155/2015/170695. [Online]. Available: https://doi.org/10.1155/2015/170695.

[18] G. Ganesan and G. Y. Li, "Cooperative spectrum sensing in cognitive radio, part i: Two user networks," *IEEE Transactions on Wireless Communications*, vol. 6, pp. 2204–2213, 2007.

[19] M. Riahi Manesh, S. Subramaniam, H. Reyes, and N. Kaabouch, "Real-time spectrum occupancy monitoring using a probabilistic model," *Computer Networks*, vol. 124, pp. 87–96, 2017, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.06.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128617302463.

[20] M. R. Manesh, M. Apu, N. Kaabouch, and W.-C. Hu, "Performance evaluation of spectrum sensing techniques for cognitive radio systems," Oct. 2016, pp. 1–7. DOI: 10.1109/UEMCON.2016.7777829.

[21] B. deepa, A. Iyer, and C. Murthy, "Cyclostationary-based architectures for spectrum sensing in ieee 802.22 wran," Jan. 2011, pp. 1 –5. DOI: 10.1109/GLOCOM.2010.5683492.

[22] A. Dandawate and G. Giannakis, "Statistical tests for presence of cyclostationarity," *IEEE Transactions on Signal Processing*, vol. 42, no. 9, pp. 2355–2369, 1994. DOI: 10.1109/78.317857.

[23] M. Jin, Y. Li, and H.-G. Ryu, "On the performance of covariance based spectrum sensing for cognitive radio," *IEEE Transactions on Signal Processing - TSP*, vol. 60, pp. 3670–3682, Jul. 2012. DOI: 10.1109/TSP.2012.2194708.

[24] S. Gong, P. Wang, and W. Liu, *Spectrum sensing under distribution uncertainty in cognitive radio networks*, Undetermined. DOI: 10.1109/ICC.2012.6363671.

[25] Z. Bao, B. Wu, P.-H. Ho, and X. Ling, "Adaptive threshold control for energy detection based spectrum sensing in cognitive radio networks," in *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, 2011, pp. 1–5. DOI: 10.1109/GLOCOM.2011.6133659.

[26] C. H. Lim, "Adaptive energy detection for spectrum sensing in unknown white gaussian noise," *IET Commun.*, vol. 6, pp. 1884–1889, 2012.

[27] H. Tang, "Some physical layer issues of wide-band cognitive radio," Dec. 2005, pp. 151 –159, ISBN: 1-4244-0013-9. DOI: 10.1109/DYSPAN.2005.1542630.

[28] Z. Quan, S. Cui, A. Sayed, and H. V. Poor, "Wideband spectrum sensing in cognitive radio networks," May 2008, pp. 901–906. DOI: 10.1109/ICC.2008.177.

[29] E. Abdessamad, R. Saadane, M. El Aroussi, M. Wahbi, and A. Hamdoun, "Spectrum sensing with an improved energy detection," in *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, 2014, pp. 895–900. DOI: 10.1109/ICMCS.2014.6911386.

[30] P. Avinash, R. Gandhiraj, and K. P. Soman, "Spectrum sensing using compressed sensing techniques for sparse multiband signals," *International journal of scientific and engineering research*, vol. 3, 2012.

[31] M. Bkassiny, Y. Li, and S. K. Jayaweera, "A survey on machine-learning techniques in cognitive radios," *IEEE Communications Surveys & Tutorials*, vol. 15, pp. 1136–1159, 2013.

[32] Z. Han, R. Zheng, and H. Poor, "Repeated auctions with bayesian nonparametric learning for spectrum access in cognitive radio networks," English, *IEEE Transactions on Wireless Communications*, vol. 10, no. 3, pp. 890–900, Mar. 2011, Funding Information: The authors would like to thank Mr. Quanyan Zhu of the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign, and Mr. Amir Danak of the Department of Electrical and Computer Engineering, McGill University, Canada, for their constructive comments. This research was supported in part by the Air Force Office of Scientific Research under Grant FA 9550-08-1-0480, by the the National Science Foundation under Grants CNS-0832084, CNS-0953377, CNS-0905556, CNS-0910461, CNS-0546391, CNS-0832089, CNS-0832084, CNS-0905398, and ECCS-1028782, and by the Qatar National Research Fund under Grant NPRP 08-522-2-211., ISSN: 1536-1276. DOI: 10.1109/TWC.2011.010411.100838.

[33] N. Shetty, S. Pollin, and P. Pawelczak, "Identifying spectrum usage by unknown systems using experiments in machine learning," May 2009, pp. 1 –6. DOI: 10.1109/WCNC.2009.4917741.

[34] D. Fudenberg and J. Tirole, *Game Theory*, ser. Mit Press. MIT Press, 1991, ISBN: 9780262061414. [Online]. Available: https://books.google.es/books?id=pFPHKwXro3QC.

[35] N. Baldo and M. Zorzi, "Learning and adaptation in cognitive radios using neural networks," *2008 5th IEEE Consumer Communications and Networking Conference*, pp. 998–1003, 2008.

[36] N. Baldo, T. B. Reddy, B. S. Manoj, R. R. Rao, and M. Zorzi, "A neural network based cognitive controller for dynamic channel selection," *2009 IEEE International Conference on Communications*, pp. 1–5, 2009.

[37] Y. Tang, Q. Zhang, and W. Lin, "Artificial neural network based spectrum sensing method for cognitive radio," *2010 6th International Conference on Wireless Communications Networking and Mobile Computing (WiCOM)*, pp. 1–4, 2010.

[38] V. Vapnik, *The Nature of Statistical Learning Theory*, ser. Information Science and Statistics. Springer New York, 1999, ISBN: 9780387987804. [Online]. Available: https://books.google.es/books?id=sna9BaxVbj8C.

[39] Z. Yang, Y. dong Yao, S. Chen, H. He, and D. Zheng, "Mac protocol classification in a cognitive radio network," *The 19th Annual Wireless and Optical Communications Conference (WOCC 2010)*, pp. 1–5, 2010.

[40] G. Tunnicliffe Wilson, "Time series analysis: Forecasting and control,5th edition, by george e. p. box, gwilym m. jenkins, gregory c. reinsel and greta m. ljung, 2015. published by john wiley and sons inc., hoboken, new jersey, pp. 712. isbn: 978-1-118-67502-1," *Journal of Time Series Analysis*, vol. 37, n/a–n/a, Mar. 2016. DOI: 10.1111/jtsa.12194.

[41] K. Fukushima and S. Miyake, "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position," *Pattern Recognition*, vol. 15, no. 6, pp. 455–469, 1982, ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(82)90024-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0031320382900243.

[42] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1d convolutional neural networks and applications: A survey," *Mechanical Systems and Signal Processing*, vol. 151, p. 107 398, 2021, ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2020.107398. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888327020307846.

[43] Y. Lecun, B. Boser, J. Denker, *et al.*, "Handwritten digit recognition with a back-propagation network," English (US), in *Advances in Neural Information Processing Systems (NIPS 1989), Denver, CO*, D. Touretzky, Ed., vol. 2, Morgan Kaufmann, 1990.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84 –90, 2012.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, 1929–1958, Jan. 2014, ISSN: 1532-4435.

[46] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[48] C. Olah. "Understanding lstm networks." (Aug. 2015), [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[49] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[50] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014. arXiv: 1406.1078. [Online]. Available: http://arxiv.org/abs/1406.1078.

[51] J. Brownlee, *Deep Learning for Time Series Forecasting: Predict the Future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 2018. [Online]. Available: https://books.google.es/books?id=o5qnDwAAQBAJ.

[52] Keras, *Keras API Reference*. (2021), [Online]. Available: https://keras.io/api/.

[53] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].