POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Artificial Neural Networks applied to Quality Prediction of a Wi-Fi link

Relatori
Dr. Stefano Scanzio
Dr. Gianluca Cena

Candidato
Francesco Xia

Dicembre 2021

Contents

List of Figures

Acknowledgements

This thesis is dedicated to my parents for their constant love and support.
A special thanks to my two supervisors for their endless patience and selflessness with which they guided me during the course of this thesis.

Summary

Artificial intelligence, and in particular machine learning, is one of the enabling technologies of Industry 4.0. It is successfully used in many application contexts and with different scopes, including, for example, in preventive maintenance, automated inspections, and optimization of communication processes. This thesis aims to evaluate whether and to what extent artificial neural networks (ANN), a particular application of machine learning, can be profitably used to predict the quality of a Wi-Fi channel in terms of frame delivery ratio. Specifically, we defined two approaches for this purpose: one based on ANNs and the other based on a more traditional approach that mimics current adaptive solutions. Then we tested the ability of each solution to predict the values of a target, which represents the state of a channel over time. Both mechanisms try to infer the future state of a wireless channel by analyzing its conditions in its recent past. For this purpose, data streams transmitted over multiple Wi-Fi channels have been sampled periodically, and these represent the starting data shared by both methods. The results obtained are encouraging, with the ANN models showing superior performance to the traditional method in almost all performance indicators examined. Our analysis also shows that the current methodology has ample room for improvement, especially in terms of performance achievable by ANN models. From our point of view, this approach has vast real-world applicability, with the primary goal of improving the reliability and resilience of a wireless system.

Chapter 1

Introduction

Machine learning is profoundly changing our everyday lives and the ways we interact with new (and old) technologies. It has been radically transforming every sector of the economy for more than a decade, from medicine to finance, from transportation to agriculture, from marketing to manufacturing, and so on. If we look at any cutting-edge technology, such as image recognition [1], self-driving cars [2], or virtual assistants [3], we most likely are witnessing one form or another of machine learning. The algorithms are designed to analyze vast amounts of data, extract useful information and use it, for example, to make decisions or predict the outcome of an event. This gives the machines the ability to perform and complete tasks without explicitly being programmed, i.e., without human intervention. In general, machine learning is part of a larger family called artificial intelligence (AI), which encompasses a wide range of technologies that enable computers to behave in an "intelligent" manner, allowing them to adapt and respond to new situations by approximating human behavior. They represent one of the focal technologies of Industry 4.0, along with the fields of IIoT, big data, cloud computing, robotics, and additive manufacturing. These elements represent the enablers of the *smart factory* concept - highly digitized factories that continuously collect and share data through sensors, machines, and manufacturing systems, which in turn are connected to one or more IT subsystems. Then, this data is used by the organization, or even directly by the "smart" devices, to improve existing processes and optimize the responses generated. Today's manufacturing facilities are pushing and adapting more and more towards this new paradigm in order to increase the overall efficiency, productivity, and reliability of their activities [4]. In this context, machine learning techniques provide decision-making ability to specific problems by processing and analyzing data collected, for example, from sensors. These methods are generally better than conventional techniques that require human intervention, as they can discern patterns, even complex ones, faster and with less effort. Then, these patterns can be profitably exploited in various tasks in the automated industrial scenario, e.g., in discovering possible failures before they occur (*preventive*

maintenance) [5, 6], for automated inspection, and in *cobots* (robots designed to collaborate with human workers) [7, 8]. In general, industrial systems are becoming increasingly automated, intelligent, and connected. However, they are also becoming more distributed in nature in order to cope with an increasing demand for higher flexibility and resilience. This implies the growing of highly heterogeneous system, especially in communications, which requires many wireless extensions with different technological characteristics in addition to a more traditional wired infrastructure [9]. For example, among the most relevant wireless technologies, we can find: IEEE 802.15.4 (WirelessHART, ZigBee, 6TiSCH) [10], IEEE 802.11 (Wi-Fi) [11], 5G for ultra-reliable low-latency communication (URLLC) [12], LoRaWAN for IoT [13], Bluetooth Low Energy (BLE) for connecting field devices (IO-Link wireless) [14]. In industrial environments, generally, wireless networks are required to satisfy two critical properties: reliability and timeliness, which denote respectively the ability of a system to operate correctly over time and its ability to respect the limits, usually time-based, imposed by the application. A wireless network is a shared medium that is intrinsically vulnerable to non-deterministic external phenomena that may cause huge variability in the spectrum conditions of a network. Hence, it is not always possible to guarantee a satisfactory quality of service for the applications or devices involved in the communication. However, several mechanisms have been developed over the years to ensure that a network meets specific quality requirements, such as, for example, access control techniques, proactive or reactive mechanisms. The former control which devices or applications can access a particular service, in our specific case a wireless channel, denying use to unauthorized entities, hence ensuring a higher quality service. Proactive mechanisms are designed to anticipate possible challenges rather than take corrective action after they have occurred. For example, time slotted channel hopping (TSCH) continuously changes the transmission frequency to minimize packet loss [15], while seamless redundancy techniques simultaneously send the same packet on different channels to increase the reliability of a communication [16]. Reactive mechanisms, on the other hand, act reactively to events or triggers, i.e., they take action after the event has happened, such as, for example, changing the transmission channel whenever the quality drops below a predefined threshold. In this category, we find, for instance, frequency agility in ZigBee, rate adaptation in Wi-Fi. These techniques, especially the last two, are designed to work in environments with fast dynamics (from a few milliseconds to seconds). The idea behind these methods can be translated and applied to the application layer. For example, if the conditions of a communication channel are expected to worsen, we can decrease the speed of a group of automated vehicles controlled over the air accordingly. Another example is roaming, where switching from one base station to another can be optimized by knowing a priori estimates of the qualities of each station. The dynamics displayed in these examples are much slower because mechanical parts are involved, which increases the interval from seconds to minutes. Another use case is to optimize the

use of a Wi-Fi channel used to interconnect two different and separate devices that are part of the same logical system. Over this high-speed communication channel, other slower data streams, such as data collected by sensors, can transit. If the communications are expected to deteriorate, we can accordingly enlarge the sampling period or disable some sensors, to reduce traffic and thus congestion on the link. The mechanisms mentioned above estimate the future state of a channel from measurements made on recent past conditions. In this regard, there are several metrics that we can use, the most popular of which are based on simple or weighted averages of successful transmission attempts. In this thesis, the primary goal is to evaluate whether and to what extent it is feasible to use artificial neural networks, a particular application of machine learning, to predict the future condition of a wireless channel [17]. The future condition is expressed similarly to the past condition, i.e., through arithmetic averages. To this end, we periodically sampled the state of multiple Wi-Fi channels operating in the 2.4 GHz band. Then we defined a target, i.e., the frame delivery ratio over a window, and compared the performances of two different approaches, one based on neural networks and the other on a heuristic method, on their ability to predict the target correctly over time given the same starting information (*features*). The target represents what we called above the spectrum condition of a channel.
The thesis is organized as follows:

- Chapter 2 presents a basic introduction of the fundamentals concepts regarding artificial neural networks
- Chapter 3 describes the channel prediction methodology from the theoretical standpoint
- Chapter 4 describes the procedure used to acquire the initial database
- Chapter 5 and 6 describe the software applications used to perform the experiments and the results obtained from them
- Chapter 7 contains the conclusions.

Chapter 2

Neural networks

2.1 Introduction

The human brain is the most intricate and fascinating organ in the human body. It is the cause of our every thought, action, and memory. It gives us the ability to easily differentiate and categorize objects in pictures, to understand and process languages, things that machines were not good at just a few years ago until the *renaissance* of artificial intelligence [18]. What is the relationship between a human brain and neural networks? Moreover, what are precisely neural networks?
A brain consists of a set of basic interconnected units, called *neurons*, that can receive, process, and transmit information to the other units. Different aspects of information are processed hierarchically in different parts of the brain. So, each level of neurons provides a unique insight that gets passed on to the next level. That is the structure that neural networks are trying to mimic.
In order for a neural network to make information-based decisions, it has to go through a process called *learning*. Its main steps remember what happens when we need to make a decision based on experience. In general, it goes along the lines:

1. remember similar situations that occurred previously
2. create a general rule out of it
3. use the rule to predict what may happen.

Similarly, we provide a massive amount of information to a neural network, which attempts to extract relevant patterns from the data. Once the training is over, the derived model can be used to predict the outcome of an event as a result of new data [19].
The information provided to the network is divided into two parts: inputs and labels. The former are the information to be processed by the neural network, while the latter are the values we wish to predict. The learning consists in iterating on the inputs more times and comparing the predicted values, that is, those

generated by the network, with the desired ones, trying each time to improve the predictions by adjusting the internal parameters of the neural network. In general, a neural network can be viewed as a simple input-output model that approximates an unknown function that outputs one or more values given a set of inputs. The closer these values are to the labels, the better the model is considered to be. It is worth noting that most intelligent tasks performed by humans can be expressed in terms of mathematical functions, even if we do not know a priori the specific characteristics for most of them. All we have at our disposal is a set of inputs on which the outputs are known, collected in a collection called *dataset*.

2.2 General aspect

Artificial neural networks (ANNs), also known as neural networks (NNs), is a computational model made of simple processing units called *artificial neurons*. Neurons are grouped in a series of layers and interconnected to each other through links that constitute the edges of a network. Information from the outside is fed to the input units, processed and transformed by intermediary "hidden" units, and eventually supplied to the output units to produce the final outputs. Each edge of the network has a corresponding weight, a scalar representing the degree of influence of one unit over another. Many different neural network architectures are available in the literature, each designed to solve a different type of problem. They can range from simple networks made of a single node (a perceptron network) to a highly complex one consisting of millions of nodes and just as many links. In Section 2.5 two commonly used neural network architectures relevant to our case study are discussed.

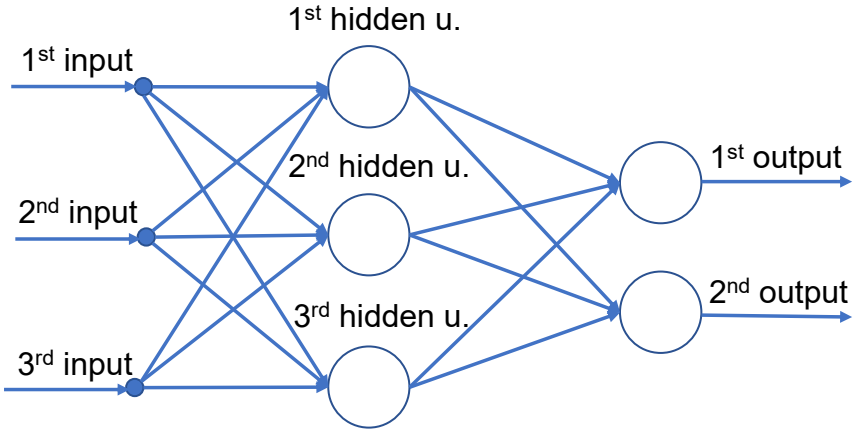


Figure 2.1: Topology of an artificial neural network

2.3 Artificial neuron

An *artificial neuron*, also called *neuron* or *unit*, is the basic unit of a neural network that is connected to other similar units (nodes) via links (edges). Each neuron can receive, process, and forward information in the form of numeric values to the other neurons of the network. Each link has an associated weight, which determines the "strength" of one's node influence on another. Neurons are organized in a series of *layers* (see Fig. 2.7).
Assuming that the network's neurons are uniquely numbered, the output of neuron k is dictated by the following equations:

$$net_k = \sum_{i=1}^{m} w_{ik} x_i + w_{0k} \tag{2.1}$$

$$\hat{y}_k = f(net_k) \tag{2.2}$$

where k is a unique identifier of the neuron, $\hat{y}_k$ denotes the output value of the neuron, f is an activation function (more on Section 2.4), x_i is node k's i-th input and w_{ik} is the weight associated to the link between the input and the node itself, and w_{0k} represents a bias term. x_0 has value 1 and here it was omitted as per convention. The weighted sum $\sum_{i=1}^{m} w_{ik} x_i + w_{0k}$ is called the **net input**, typically abbreviated as net_k.
An alternative form of (2.2), using matrices, is:

$$\hat{y} = f(\boldsymbol{W}^T \boldsymbol{X} + w_0) \tag{2.3}$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$

2.4 Activation functions

The function $f(\cdot)$ introduced in (2.2) is called **activation function**. It decides how the net input is transformed into the output of the neuron. The activation functions can be essentially divided into two main groups:

- linear activation functions: they are functions that can be expressed as a polynomial function of degree zero
- non-linear activation functions: all other functions that are not linear functions.

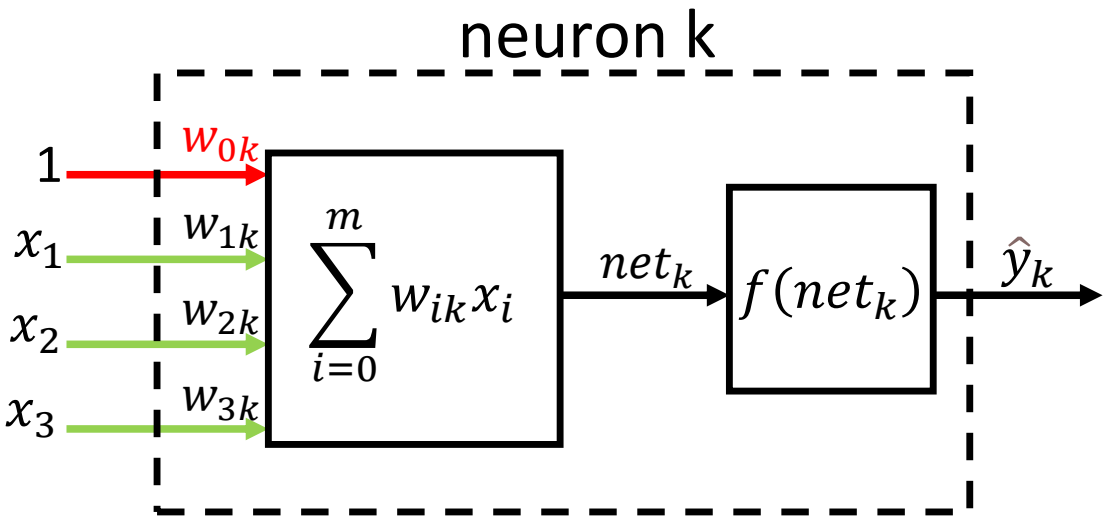


Figure 2.2: Visualization of a neuron's anatomy

The choice of proper activation functions is an essential task that needs to be carefully tuned. It directly impacts a neural network's ability to reach convergence, i.e., to estimate the function it wants to learn with an error lower than a given threshold and the speed with which it does so. As we will see later, inappropriate selection of activation functions can cause a number of problems known as *vanishing/exploding gradients*, which prevent neural networks from reaching the convergence state [20]. Although linear transformations make neural networks simpler, they are less powerful and cannot learn complex patterns from the data. Our studies will concentrate mainly on non-linear activation functions with a short mention of the identity function. There are several benefits to use non-linear activation functions:

- they restrict the output value of a neuron to a particular range. It has been proven empirically that if the output value of a neuron is unbounded, it can lead to computational issues and delayed convergence
- they introduce non-linearity into the neural networks, and consequently improve its ability to learn complex problems.

The non-linearity property is essential to solving a multitude of different types of tasks, which would not have been possible just by relying on linear transformations unless approximations are made. For example, let us imagine we are given the task to divide the data points in Fig. 2.3 into two groups using a curve, where each group must contain only points of the same color. If we choose to use only linear activation functions, the output value of a neuron would be a linear combination of its inputs, which in turn are also a linear combination of another node's inputs,

and the same applies to the output values. Thus the problem would be equivalent to trying to perfectly divide the data points in Fig. 2.3 using just a single straight line. We can take a step further by saying that any multi-layer neural network that uses linear activation functions is equivalent in terms of capabilities to single-layer neural networks using the same type of activation functions.

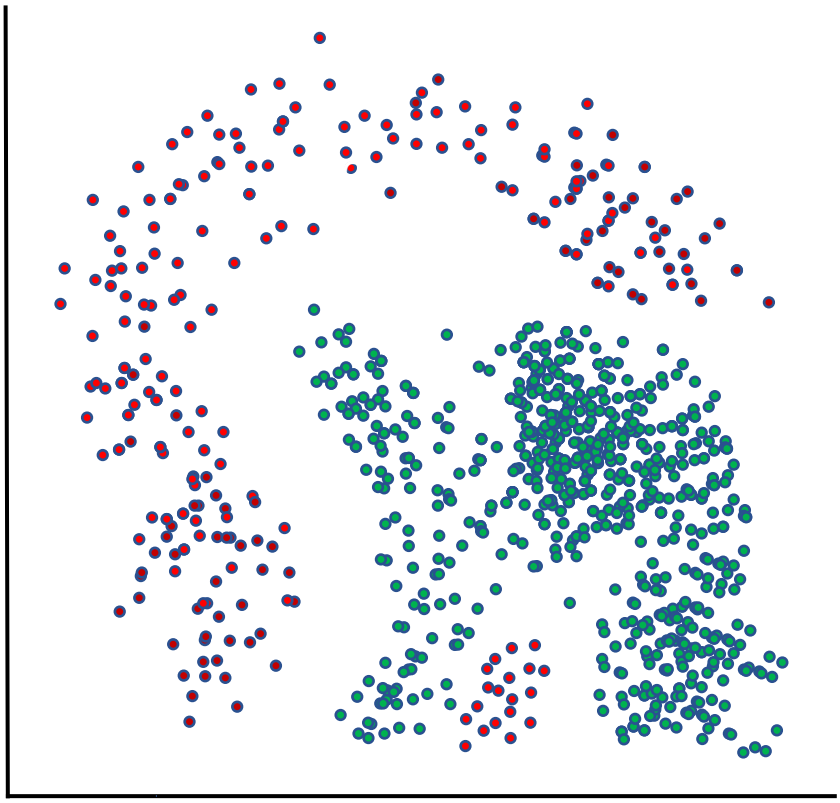


Figure 2.3: A plane of red and green dots

Intuitively we can see that it is impossible to have two perfect groups separated by a line. Even though it is a well-constructed example, most fundamental problems cannot be solved with desirable accuracy using only linear activation functions. That is the principal reason why we need to introduce non-linearity into most of the neural networks so that they can **approximate arbitrary complex functions**. That is what makes neural networks extraordinary powerful. Activation functions are generally required to be differentiable, which means its first-order derivative must exist at each point in its domain. This property is expected because most neural networks are typically trained using a learning algorithm called backpropagation that requires the derivative of the activation functions in one of its steps (more on Section 2.6.2). Among the many different activation functions available in the literature, the most used and common ones are:

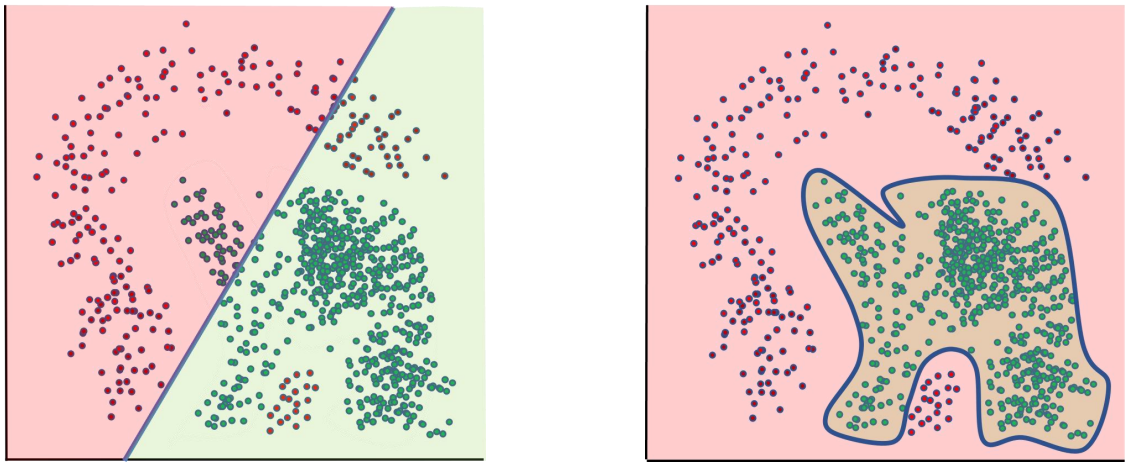


Figure 2.4: A dataplane split using a linear function (left picture) and a non-linear function (right picture)

- ReLu. It is possibly the most popular activation function used in deep neural networks. Its equation is:

$$y(x) = max(0, x) \tag{2.4}$$

  It became increasingly popular in recent years because of its simple shape and it counteracts well the issue of vanishing gradients that prevents model from converging.

- Sigmoid. It is a function with a S-shape that takes as input any real values and outputs values in the range 0 to 1.

$$y(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

  where e is the Euler's number.

- Tanh. It is a function that it is very similar to the sigmoid activation function. It transforms any numerical value received as input to a value in the range -1 to 1.

$$y(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.6}$$

  where e is the Euler's number.

- Identity. It is a function that outputs the value received as input. It is part of a larger family called linear activation functions, whose functions can be expressed as:

$$y(x) = cx \tag{2.7}$$

where c denotes the slope of the line. For the identity function, c assumes the value 1

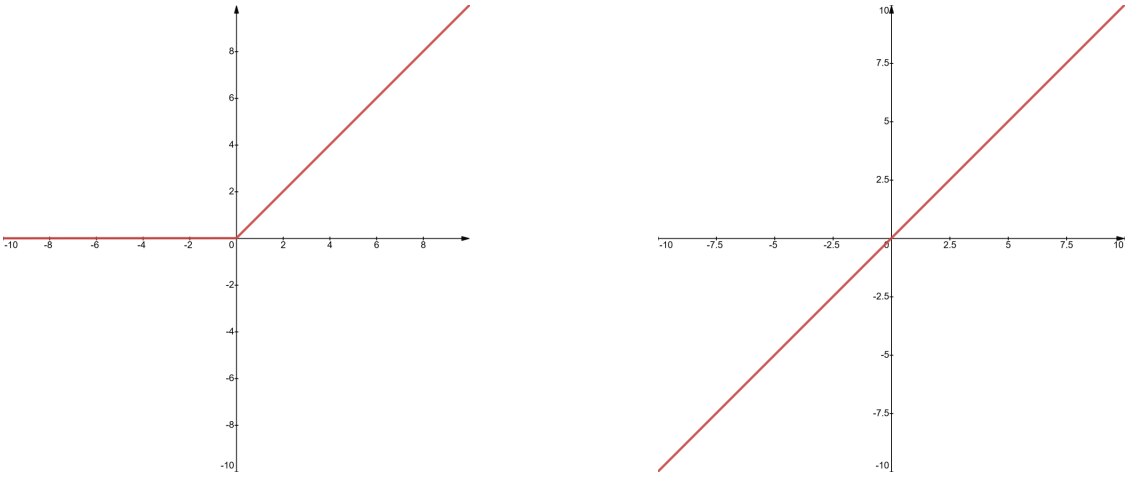


Figure 2.5: A plot of the ReLu function (left picture) and the identity function (right picture)

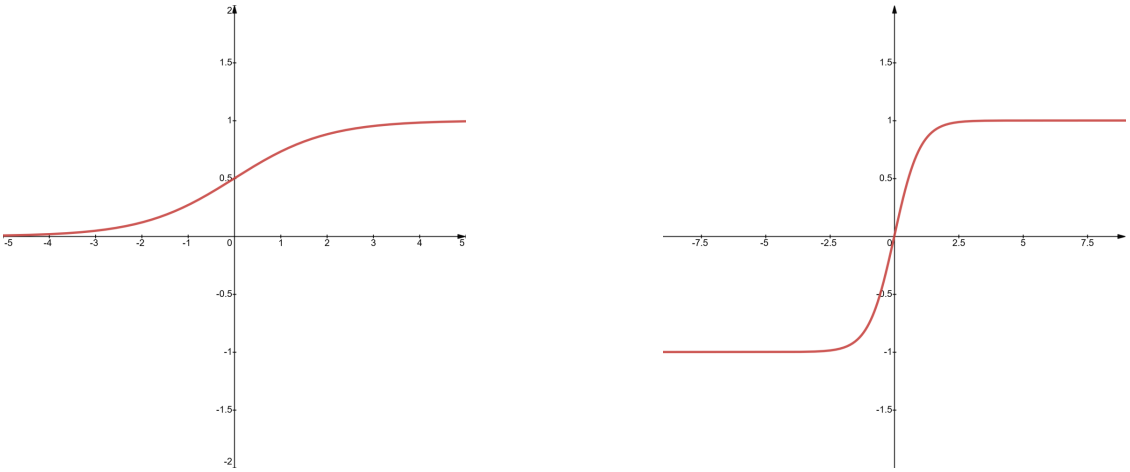


Figure 2.6: A plot of the sigmoid function (left picture) and the tanh function (right picture)

2.5 Topology

The ways in which neurons are organized and connected within a network is called the topology of a neural network. It is an important aspect since it directly impacts

the functioning and learning of a network. In most topologies, neurons are organized into logical groups called *layers*. Neurons are connected to other neurons through weighted links or connections, and it represents the only mechanism available for neurons to share data inside a network.
The weight associated with each connection denotes the strength of one node's influence on another. The layer that receives data from outside is known as *input layer*, whereas the layer that produces the outputs is the *output layer*. Every other layer is called *hidden layer*. The following explains different ways used to describe the shape and capability of a neural network:

- size: number of neurons in the network
- width: number of neurons in a specific layer
- depth: the number of layers in a neural network. Note that usually in the count we do not consider the input layer
- capacity: the type or structure of functions that can be learned by a network configuration
- architecture: the specific organization of the layers and neurons in the network.

Alternatively, the network can be expressed in terms of nodes and arcs using the terminology of the graphs, representing respectively the neurons and connections composing a neural network. From the many topology available in literature, which are typically application specific [21], we will discuss two of the most common and relevant ones for our work: feed-forward neural networks and recurrent neural networks.

2.5.1 Input layer

It is the first layer in a neural network. Its width is equivalent to the dimension of an input, i.e., an external data. An input is often represented in the form of a vector made of numerical values:

$$X_t = (x_1, x_2, ..., x_{N_{features}})_t \tag{2.8}$$

where $N_{features}$ denotes the dimension of an input vector and t identifies the vector inside a dataset T.
An element of an input vector is called *feature*. The external data, before it is fed to the network, normally go through a process called data cleaning, which refers to all kinds of tasks used for: detecting and repairing errors, data transformation, normalization and standardization.

2.5.2 Hidden layers

A hidden layer is any layer located between the input and output layers. It does not directly receive external data nor it produces outputs. Neurons applies different transformation on the input data using (2.2).

2.5.3 Output layer

It is the layer responsible for producing the output values. It can be a single value or a vector of values depending on the width of the layer, i.e., number of neurons present

$$\hat{Y}_t = (\hat{y}_1, \hat{y}_2, ..., \hat{y}_{N_{outputs}})_t \tag{2.9}$$

where $N_{outputs}$ denotes the number of output values generated and t identifies the output value generated following the input vector t.

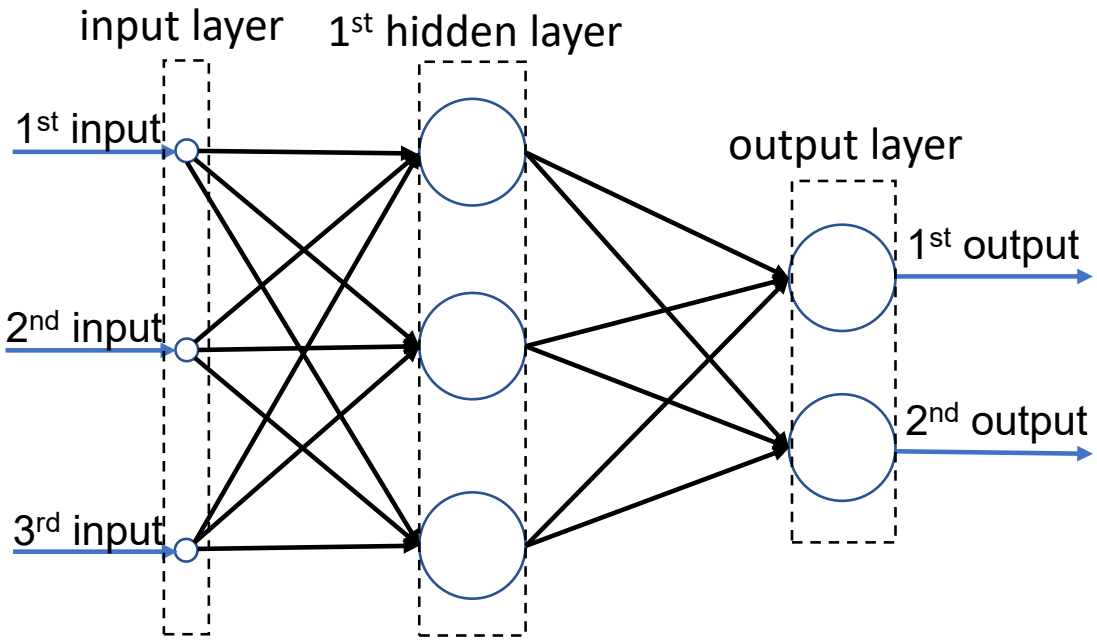


Figure 2.7: Topology of a fully connected feed-forward neural network

2.5.4 Feed-forward neural networks

Feed-forward neural network (FFNN) is one of the first and simplest type of artificial neural network conceived. The network layers are arranged in a sequence with neurons forming links only with neurons of the next layer, and in the case all

units are connected to every unit of the next layer the network is said to be *fully-connected*. In essence feed-forward neural networks do not contain any loops, so the data moves forward in one direction: the data is feed to the input layer, then it goes through the hidden layers until it reaches the output layer. This explains why these networks are called feed-forward networks. A feed-forward neural network can be implemented in its simplest form as a single layer with one neuron, which is capable of learning only linearly separable patterns (see Fig. 2.4)
As mentioned in Section 2.1, most real-life tasks can be expressed in terms of mathematical functions. That is also what neural networks are trying to accomplish. They can be seen as black boxes that approximate a mathematical function that, given any input, generates one or more output values. The better is the approximation, the more accurate are the predictions generated by the network. A fundamental theorem called "The universal approximation theorem" states that FFNNs with as few as one hidden layer can approximate any continuous function with any arbitrary degree of accuracy [22, 23]. The basic idea is to approximate the continuous function with a series of rectangles placed side by side, where a neuron conceptually represents a rectangle. Thus, adding more neurons results in a more accurate approximation of the function, but this is paid for by adding more computational complexity during the training phase [24].

2.5.5 Recurrent neural networks

Recurrent neural networks (RNN), in contrast to feed-forward networks, contain loops inside the network. That means that data can be fed back as input into the network before it is forward again for further processing. This allows the network to persist information between successive predictions. This effect emulates what is known as memory. In particular, special units called "memory units" stores important information from the recent that influence the current input and output. So the predictions are generally more precise compared to other neural networks architectures such as feed-forward neural networks. This makes them suitable for tasks that deals with sequential data or time series data such as speech recognition or natural language processing.

2.6 Training

Training is the process that aims to set the internal parameters of a neural network (i.e., weights and biases) to a set of values, such that the final model can generate accurate outputs in response to any given inputs. Generally, an output is more accurate the closer its value is to the desired value. At its core, training a neural network means to extrapolate the relationship, if any, between the inputs, i.e., the information fed to the ANN, with the desired values, known as *labels*. Colloquially

this process is also called *fit the model*. Typically, inputs and labels are grouped in a data structure called *dataset*.
Henceforth we will use the following conventions:

$$D = \{ (X_t, Y_{true_t}) \mid t \in (1, ..., s) \} \tag{2.10}$$

where D denotes a dataset, s is the number of elements contained in D (*size*), t identifies a tuple inside D, X_t and Y_t follow respectively the definitions (2.8) and (2.9). For common usage, D is divided into two subsets:

- TRA: a subset called *training set* that it is used to train an ANN
- TES: a subset called *test set* that it is used to evaluate an ANN once it has been training.

Generally, a good value for the train-test split ratio is 80-20 (e.g., 80 percent of inputs in D is allocated to TRA and 20 percent to TES), even though it may vary depending on the particular use case and size of the dataset.
During training, the learning algorithm iterates over the training set in multiple passes (training epochs), adjusting the weights and biases associated with the connections as it progresses with the goal of minimizing a *cost function*. This function, also known as error function or loss function, estimates the progress of the learning of an ANN in terms of the qualities (closeness) of its predictions with respect to the labels. What it does is measure the "distance" between the predicted values and the target values. As it progresses with training, the cost function will likely decrease over the epochs until it settles around a final value. When that happens, it is said that the neural network reached a convergence state, meaning that any additional training will not improve the model.
Choosing a loss function can be challenging as the function must comply with the properties of the problem that a neural network is trying to solve. In the following are described three common activation functions used for most types of problems:

- mean squared error (MSE). It computes the mean of the squared differences between the predictions and the labels

$$MSE = \frac{1}{f} \sum_{i=1}^{f} (y_{pred}^{(i)} - y_{true}^{(i)})^2 \tag{2.11}$$

- mean absolute error (MAE). It computes the mean of the absolute differences between the predicted values and the target values

$$MAE = \frac{1}{f} \sum_{i=1}^{f} |y_{pred}^{(i)} - y_{true}^{(i)}| \tag{2.12}$$

- cross-entropy (CSE). It is defined as:

$$CSE = -\sum_{i=1}^{n} y_{true}^{(i)} log(y_{pred}^{(i)}) \tag{2.13}$$

where $y_{true}^{(i)}$ denotes a label and $y_{pred}^{(i)}$ is the prediction generated by the ANN model associated with the inputs of $y_{true}^{(i)}$. It is important to note that the choice of loss function is directly related with the activation function used in the output layer of a neural network. For regression problems, i.e., when you want to predict a numerical value, such as the one presented in this thesis, the standard configuration is to use as loss function either MSE or MAE and as activation function a linear one in the output layer. Another widespread configuration is the one used for binary classification problems, i.e., when the output you are trying to predict is a binary value (0 or 1). The sigmoid is typically used as the activation function along with the (binary) cross-entropy as the objective function [25].
The loss functions defined above refer to the error associated with a single prediction, while the real cost function used to measure the overall loss on a dataset is defined as:

$$J(T, \theta) = \frac{1}{t} \sum_{i=1}^{t} loss(y_{pred}^{i}, y_{true}^{i}) \tag{2.14}$$

where $J(T, \theta)$ denotes the overall error computed over dataset T with particular set of weights θ, $loss$ denotes a loss function, i.e., (2.11), (2.12) or (2.13), and m is the size of dataset T. If the predictions of a model are completely incorrect, (2.14) will output a high value, while if they are close to the desired values, it will output a low value. As the weights and biases are tuned during training, the cost function will tell you if you are improving.
Before proceeding further, I would like to clarify the difference between *model parameters* and *model hyperparameters*. The first types are the variables we have so far called weights and biases, which are internal to the model whose values are derived from the training data. On the other hand, the hyperparameters are external variables that determine the structure of the model and the quality of the training process. They cannot be estimated from the data, and they must be optimized before the training phase, manually or automatically. They are important because they directly control the behavior and speed of the training algorithm and have a significant impact on the final performance of the model being trained. Common examples of hyperparameters are learning rate, activation functions, number of hidden layers, number of epochs, and batch size value. From now on, for convenience, when we use the term parameter, we will refer to the model parameters unless it is specified otherwise.
In practical terms, the training algorithm is composed of three phases:

1. the forward phase that deals with generating the predictions related to a batch size of inputs, with the current state of the model

2. the evaluation phase that deals with evaluating the objective function

3. the backward phase that deals with updating the weights and biases. Relative to FFNN networks, this phase is usually implemented by two algorithms called *backpropagation* and *gradient descent*. In general terms, the former deals with estimating the contributions, or rather the effect, that each weight has on the overall error. In contrast, the second algorithm decides how the various model parameters should be updated so that the overall loss decreases, using the results obtained from the backpropagation step.

Sometimes step 2 and 3 can be combined into a single phase. To summarize, the training algorithm can be represented like:

Algorithm 1 Training algorithm for $batch\_size = size(T)$

initialize θ to random values
declare and initialize an empty array v
while $epoch \leq MAX\_EPOCHS$ **do**
  for all $(X_t, Y_t) \in T$ **do**
    $\hat{Y}_t = forward\_phase(X_t, \theta)$
    store $(\hat{Y}_t, Y_t)$ in v
  end for
  $e = J(v, \theta)$
  $\theta = backward\_phase(e, \theta)$
  $epoch \leftarrow epoch + 1$
end while

Knowing a priori the best number of epochs can be difficult as it depends on the specific use case and the level of performance that the network is trying to achieve.

2.6.1 Forward phase (forward propagation)

It is the first phase of the training algorithm. It is responsible for generating a prediction $\hat{Y}_t$ (definition 2.9) given an input pattern X_t (definition 2.8) extracted from a dataset T. In the case of a fully connected feedforward neural network, X_t is fed to the input layer, which propagates the values to the first hidden layer. Then each hidden unit computes their output using (2.2) and forward their values to the units of the next layer. Then the process of computation and propagation is repeated up to the neurons of the output layer. For conciseness, the aforementioned steps are summarized in the function $forward\_phase(\cdot)$:

$$\hat{Y} = forward\_phase(X_t, \theta) \tag{2.15}$$

where θ denotes a specific set of weights and biases.

2.6.2 Backward phase

The backward phase is an essential step of the training algorithm. The goal is to reduce, after each epoch, the measure of distances between the produced output values and the desired output values: it does so by minimizing a cost function that is properly selected before the start of the training phase. As we anticipated in the previous section, a cost function quantifies how close the predictions are to the desired values. From calculus, we know that the gradient of a multivariable differentiable function represents the direction in which the function increase most quickly (the direction that points to a maximum), while its magnitude is the rate of increase in that direction. So the opposite of it shows the direction of the steepest descent. Since a loss function is also a multivariable function, the idea is to repeatedly adjust the network's weights and biases in the opposite direction of the gradient of the cost function computed with respect to the model parameters. The weight's updates are defined by an optimization algorithm called gradient descent (more in Section 2.6.3), while backpropagation is responsible for computing the gradients of the loss function. The "back" part of the name "backpropagation" stems from the fact that the computations of gradients with respect to every weight of the network are done in reverse order, starting from the neurons of the output layer and proceeding backward until the neurons of the input layer. The strength of this algorithm resides in its ability to compute the various gradients efficiently: the values are propagated backward and reused for the calculations related to the weights of the previous layers, as opposed to a more naive approach of computing independently every gradient. This efficiency makes it possible to train a neural network that may contain millions of weights.

2.6.3 Gradient descent

It is an optimization algorithm used by neural networks for finding a local or global minimum of a cost function. The algorithm updates the weights and biases at the end of each iteration according to this formula:

$$\theta_{i+1} = \theta_i - \alpha \frac{\partial J(\theta_i)}{\partial \theta_i} \tag{2.16}$$

where θ_i denotes collectively a specific set of weights at time i, and α is a coefficient called learning rate. For $i = 0$, θ is initialized with random values.
The minus sign in (2.16) is a consequence that the gradient of a function, as mentioned earlier, marks the direction of greatest increase, so to get the direction of greatest decrease, one must negate the gradient. The learning rate determines the size of each step taken (magnitude of an update), which figures out how fast or slow we will move towards the optimal weights (more in Section 2.6.4). We repeatedly apply (2.16) until the termination criteria are met, e.g., the value of the loss

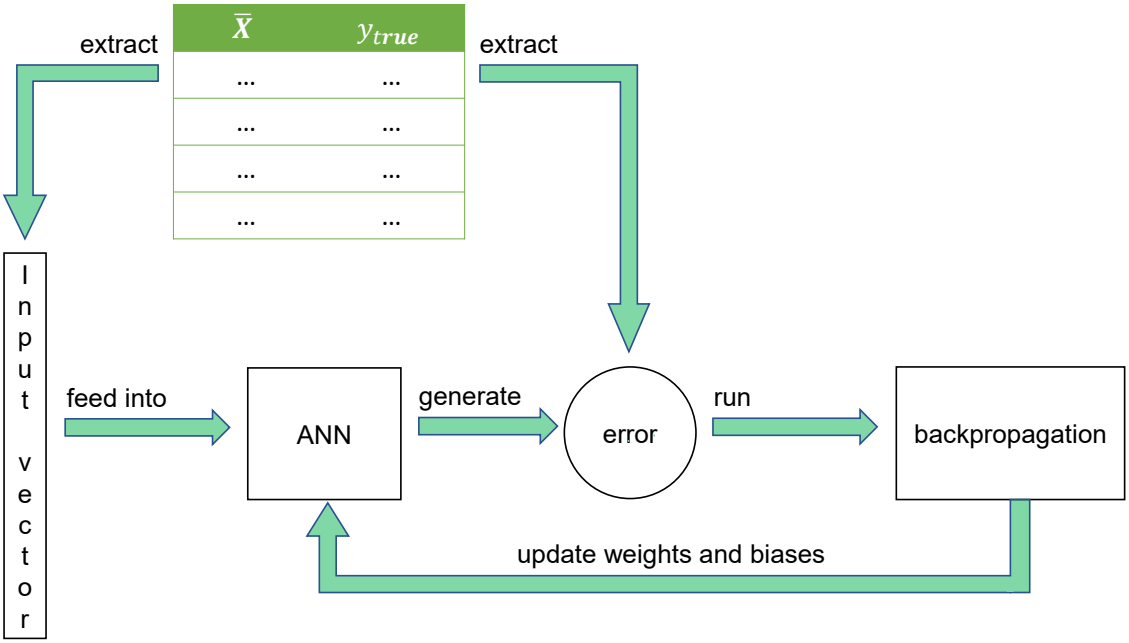


Figure 2.8: Training algorithm diagram

function is below a given threshold value or we have reached the agreed number of epochs.
An analogy that comes up frequently when describing the intuition behind the algorithm is the following: a person is trying to move down from a mountain, but it is surrounded by heavy fog. Hence, the person has not a clear path to walk on, but it has only a general overview of its proximity. An ingenious solution is to take repeatedly small steps towards the direction where the ground has the steepest descent around his current position. Using this method, he will either arrive at the bottom of the mountain or get stuck in a mountain hole. Minimizing the cost function is analogous to descending the mountain. However, like in the example, the neural network may get stuck in a suboptimal minimum (mountain hole) since it is working only with local information. Feeling the mountain's steepness is like computing the gradient, and taking a step is equivalent to updating the weights.
The next few paragraphs describe the mathematical steps involved in computing a gradient with respect to a generic weight. The convention used are:

1. superscript k denotes the k-th layer of a network. The input layer is considered to be the 0-th layer

2. subscripts consisting of a single value denote the index of a neuron inside its layer

3. subscripts associated with the weights consist of two numbers that denote

the identifiers of the neurons involved in the connection. We will treat hereafter only connections for FFNN networks, i.e., the two neurons involved are arranged on two adjacent layers. For example, w_{ij}^k denotes the link between neuron i of layer k-1 and neuron j of layer k.

As anticipated, the goal is to update every weight so that each prediction is closer after every update to their target values. The magnitude of a weight update is dictated by the gradient of the cost function with respect to that weight $\frac{\partial E(T,\theta)}{\partial w_{ij}^k}$, which conceptually what it does is to tell us what effect has a small change in a weight to the overall loss function.
From calculus, we know that the derivative of the sum of multiple functions is equivalent to the sum of their derivatives. Since the cost function, according to definition (2.14), is the average of single errors committed on a dataset T, its gradient with respect to weight w_{ij}^k becomes:

$$\frac{\partial E(T,\theta)}{\partial w_{ij}^k} = \frac{1}{S}\sum_{u=1}^{S}\frac{\partial E_u(y_{pred}^u, y_{true}^u)}{\partial w_{ij}^k} \tag{2.17}$$

Therefore, instead of solving (2.14) entirely, we can focus on the mathematical steps involved in calculating the derivative of the error associated with a single input. Then to get the overall gradient, we simply need to average the individual smaller gradients. Henceforth we assume that E is E_u unless otherwise specified.
Applying the chain rule on $\frac{\partial E}{\partial w_{ij}^k}$ we obtain:

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial out_j^k}\frac{\partial out_j^k}{\partial net_j^k}\frac{\partial net_j^k}{\partial w_{ij}^k} \tag{2.18}$$

The terms $\frac{\partial E}{\partial out_j^k}$ and $\frac{\partial out_j^k}{\partial net_j^k}$ are often simplified as

$$\delta_j^k = \frac{\partial E}{\partial out_j^k}\frac{\partial out_i^k}{\partial net_j^k} = \frac{\partial E}{\partial net_j^k} \tag{2.19}$$

while $\frac{\partial net_j^k}{\partial w_{ij}^k}$ can be rewritten as

$$\frac{\partial net_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{l=0}^{r_{k-1}} w_{lj}^k out_l^{k-1}\right) = out_i^{k-1} \tag{2.20}$$

where r_{k-1} denotes the number of neurons present in layer $k-1$ and δ_j is called the *error* term.
Therefore, (2.18) can be summarized as

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k out_i^{k-1} \tag{2.21}$$

The error δ_j^k assumes a different expression depending on whether we are computing the gradient with respect to a weight linked to a node of the output layer or not. In the former, δ_j^L assumes:

$$\delta_j^L = \frac{\partial E}{\partial net_j^L} = loss'(act_j^L(net_j^L), y_{true}^i) * act'^L_j(net_j^L) \tag{2.22}$$

where L denotes the output layer and act_j is the activation function of the j-th neuron. Putting everything together, the partial derivative of E with respect to a weight in the output layer w_{ij}^L is:

$$\frac{\partial E_u}{\partial w_{ij}^L} = loss'(act_j^L(net_j^L), y_{true}^u) * act'^L_j(net_j^L) * out_i^{k-1} \tag{2.23}$$

With regard to neurons in the hidden layer $1 \leq k < L$, using the chain rule, δ_j^k becomes

$$\delta_j^k = \frac{\partial E}{\partial net_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial net_l^{k+1}} \frac{\partial net_l^{k+1}}{\partial net_j^k} \tag{2.24}$$

where r^{k+1} denotes the number of nodes in the next layer. Note that, l starts from 1 because the bias out_0^k corresponding to w_{0j}^{k+1} is constant, i.e., its value does not depend on the outputs of the previous layers. The first term of the right side is, by definition, the error of the next layer, δ_l^{k+1}

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial net_l^{k+1}}{\partial net_j^k} \tag{2.25}$$

an alternative form of (2.1) is

$$net_j^k = \sum_{i=0}^{r_{k-1}} w_{ij}^k g(net_i^{k-1}) \tag{2.26}$$

and by substituting the above in the second term of (2.25), we get

$$\frac{\partial net_l^{k+1}}{\partial net_j^k} = w_{jl}^{k+1} g'(net_j^k) \tag{2.27}$$

The final expression for δ_j^k is

$$\delta_j^k = g'(net_j^k) \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \tag{2.28}$$

Putting everything together the partial derivative of E with respect to a weight w_{ij}^k in the hidden layers $1 \leq k < L$ is

$$\frac{\partial E}{\partial w_{ji}^k} = \delta_j^k out_j^{k-1} = out_j^{k-1} g'(net_j^k) \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \tag{2.29}$$

The error term associated with a weight in the k-layer is dependent on the error term of the weights at the k+1 layer (see (2.25)). Thus the standard procedure is to compute the error terms starting from the output layer and propagate them backward, updating them whenever necessary, layer after layer until the first layer is reached. This process is what makes backpropagation efficient at computing the gradients and suitable to be used together with gradient-based optimization algorithms such as gradient descent.
To compute the gradient of the cost function with respect to w_{ij}^k we simply average the individual derivatives computed over the entire dataset, i.e.,

$$\frac{\partial E(T,\theta)}{\partial w_{ij}^k} = \frac{1}{S}\sum_{u=1}^{S}\frac{\partial E_u}{\partial w_{ij}^k} \tag{2.30}$$

where S is the size of the dataset.
To recap, once the gradients have been calculated via backpropagation, the weights are updated via gradient descent using this formula:

$$w_{ij}^k = w_{ij}^k - \alpha\frac{\partial E(T,\theta)}{\partial w_{ij}^k} \tag{2.31}$$

where α is a scalar called the learning rate.
What we have described so far is the standard gradient descent, also known as vanilla gradient descent, in which the gradient of each update is computed over all examples contained in a training dataset. This process can be very time-consuming in large-scale applications [26, 27], where training datasets may contain millions of examples. A very common strategy to counteract this problem is to consider updating the weights after every x elements, where x is also known as *batch size*. It is a hyperparameter of the model that defines the number of patterns to consider for each update. Its value is usually a power of 2 and can range from $0 < batchsize < size(T)$, where $size(T)$ denotes the size of a dataset T. Two variants of the primary gradient descent algorithm are commonly found in the literature, differing only in the batch size value:

- stochastic gradient descent (SGD). It updates the model's parameters for each input pattern seen during training (batch size = 1). It tends to be much faster computation-wise but can cause the cost function values to fluctuate strongly over the course of a training instance because the gradient evaluated on a single input is a noisy approximation of the complete one
- mini-batch gradient descent is halfway between vanilla gradient descent and SGD. It performs an update after every mini-batch of n training samples seen, where n is in the range between 1 and the size of the training dataset (exclusive). This variant provides better efficiency compared to the vanilla version in terms of memory and speed, and it reduces the amount of noise found in SGD.

The main challenges associated with gradient descent algorithm are:

- choosing a suitable learning rate can be a daunting challenge. A learning rate that is too small leads to a slow convergence, while a learning rate that is too big causes the neural network to overshoot and never reach convergence
- the same learning rate is applied to all weights and biases. Sometimes we would like to apply different learning rates if our data set contains sparse data and features with very different frequencies
- finding a local minimum that generalizes well enough our problem. An objective function associated with real-life data sets have, in most cases, very complex surfaces with many different valleys (minimum points). Finding a global minimum can be sometimes infeasible.

To recap, gradient descent is an optimization algorithm for finding a minimum of a multivariable function. In our case, the function we want to optimize, i.e., the objective function, is dependent on the model parameters. By repeatedly applying (2.31) on the weights and biases, we hope to eventually arrive at a set of optimal values for the parameters so that the overall error associated with our model is the absolute smallest.

2.6.4 Learning rate

The learning rate, also called step size, is a positive scalar, often between 0 and 1. Its purpose is to scale the magnitude of a weight update, found inside the gradient descent algorithm. Intuitively, it determines how big the modification of NN parameters is in the direction of a local minimum. It is a vital hyperparameter that impacts whether a neural network can reach convergence, and if so, at what speed. This parameter can be either tuned in manually by the practitioner or adjusted automatically by an optimizer during training phase. Generally speaking, we should set the learning rate to a value that is neither too high nor too low for several reasons:

- having a small learning rate can cause the network to converge too slowly and get stuck in a sub-optimal minimum (left picture of Fig. 2.9)
- having a big learning rate can cause the network to overshoot, leading to an increase of the overall loss (right picture of Fig. 2.9)

Although it is not possible to know for certainty the optimal learning rate a priori, there are some well-known techniques available to figure out a reasonable learning rate. They are built on the recurrent idea of changing the learning rate dynamically as an ANN progresses with its training instead of keeping it constant for all the

train process. Examples of these techniques are: "learning rate schedules" and "adaptive learning rate". The former reduces the learning rate according to a predefined schedule, e.g., half the learning rate after every epoch. This technique has shown, in some application contexts, better performances compared to using a fixed learning rate. However, it requires defining a schedule in advance, which can be problematic since it is hard to know which are the best settings to adopt. Furthermore, it lacks flexibility and adaptability since we are most likely required to define a new schedule for a different type of problem. The adaptive learning rate algorithms update the learning rate at run time based on heuristic methods, such as adapting its value according to the values of past gradients. Their main advantage over other techniques is that they require less work on tuning the hyperparameters.

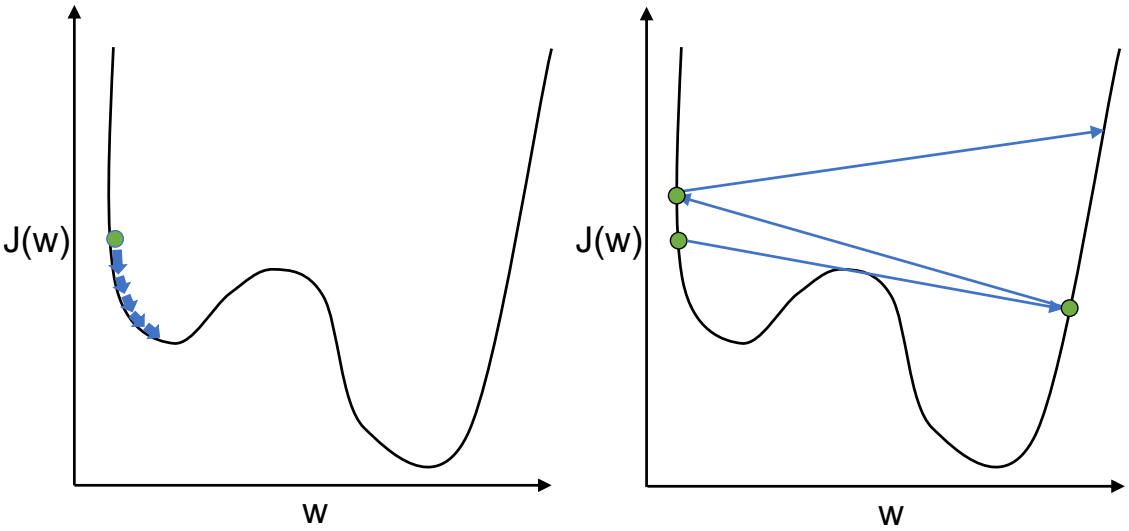


Figure 2.9: Effect of the learning rate on the cost function: a small learning rate can cause a slow convergence (left picture), while a large learning rate can cause the neural network to overshoot (right picture)

2.6.5 Momentum

Momentum is another technique extensively used to help a neural network finding an optimal set of model parameters. It is a concept that has been taken from physics, which expresses an object's capability in movement to continue its trajectory even when an external opposing force is applied to it. The same core idea has been translated for machine learning. A term called momentum has been added to (2.31) that takes into account the general direction of the previous updates and

minimize the effect of opposite changes. So (2.31) becomes:

$$\Delta w_{ij}^{k(t)} = \alpha \frac{\partial E(T,\theta)}{\partial w_{ij}^{k(t)}} + m\Delta w_{ij}^{k(t-1)} \tag{2.32}$$

$$w_{ij}^{k(t)} = w_{ij}^{k(t)} - \Delta w_{ij}^{k(t)} \tag{2.33}$$

where m, called momentum factor, denotes a coefficient, t indicates an update associated to weight, and $\Delta w_{ij}^{k(t-1)}$ is the update vector at time $t-1$. It is common practice to use momentum values close to 0.9, but it can take any value in the range between 0 and 1.
With momentum, the neural network has better chances of not getting stuck in a local minimum, avoiding oscillating around it. In fact, the momentum propels the updates of the weights of a neural network to follow the general direction of the last updates, reducing the effect of opposing changes. In practical terms, this is implemented by saving in the momentum some parts of the past gradients. As we can see from (2.33), momentum increases when the gradient continues to point in the same direction while dampening the effect of opposing changes. As a result, the neural network gains faster convergence and decreases oscillations.

2.6.6 Vanishing and exploding gradients

According to (2.31), each of the weights of a neural network receives an update that is proportional to the gradient of the loss function with respect to the current weight. In some cases, the gradient can be so small that it prevents the weight from changing value. In the worst case, this can completely stop the training of a neural network. One cause of this problem is the activation functions. For example, let us look at the hyperbolic tangent function. Its gradient has values in the range (0,1), and as we know, backpropagation computes gradients using the chain rule. This has the effect of multiplying n of these small numbers to compute the gradients of the first layers in an n-layer network, which means that the gradient decreases exponentially as we approach the neurons in the first layers. The problem presented above is called *vanishing gradients*.
A similar problem called *exploding gradient* is when the activation functions have derivatives that can take on extremely large values, resulting in large gradients. This, in turn, causes large updates in the weights, making the neural network unstable. In the extreme case, the values could be so large that they cause overflows; that is, the machine cannot correctly represent the value of the gradients.

Several techniques can be used to mitigate the problems mentioned above:

- reducing the number of layers that make up a neural network
- use a technique called gradient clipping to prevent exploding gradients. The idea is to scale the gradients so that their norm does not exceed a given value.

- use activation functions that suffer less from the vanishing gradient problem, such as ReLu [28].

2.6.7 Underfitting and Overfitting

Let us briefly introduce two of the most common issues that negatively affect a model: underfitting and overfitting. Underfitting refers to a model that can neither learn from the training data nor generalize to unseen data. It is easy to detect since underfitted models have poor performance metrics, e.g., inaccurate predictions. In the context of neural networks, generalization refers to the model's capability to produce good predictions, i.e., reasonably close to the desired output, to sets of inputs that it has never seen. This means that the current configuration of the neural network failed to capture any relevant pattern from the training data. On the other hand, a model that suffers from overfitting matches nearly perfectly to the training data while it does inadequately on data that it has never seen before. The main reason why a model goes into overfitting is because of the noise collected from the training dataset. Noises are those features that appear in the training dataset that are, for example, mislabeled or rarely show up in other similar datasets. This causes the neural network to acquire distorted or incorrect information, generating less accurate predictions when dealing with real-world data [29]. There are multiple strategies available that can be adopted to reduce the effect of overfitting:

- early stopping: it is a strategy that consists of stopping the training when it reaches a point where the performance metric evaluated over the testing data set stops improving. Hence, it simply freezes the model before it gets a chance to overfit (Fig. 2.10)

- expanding the training data set: a model performance can be significantly improved using a more extensive data set

- dropout is based on the idea of temporarily dropping some nodes during training to reduce the phenomenon called co-adaptation. In a way, we are reducing the capacity of our neural network, forcing it to create "stronger" connections on those links where usually would have little influence on the final prediction. Furthermore, dropout makes it easier to train the neural network, reducing the computation complexity of the model significantly. It is an effective strategy to adopt for large or complex networks.

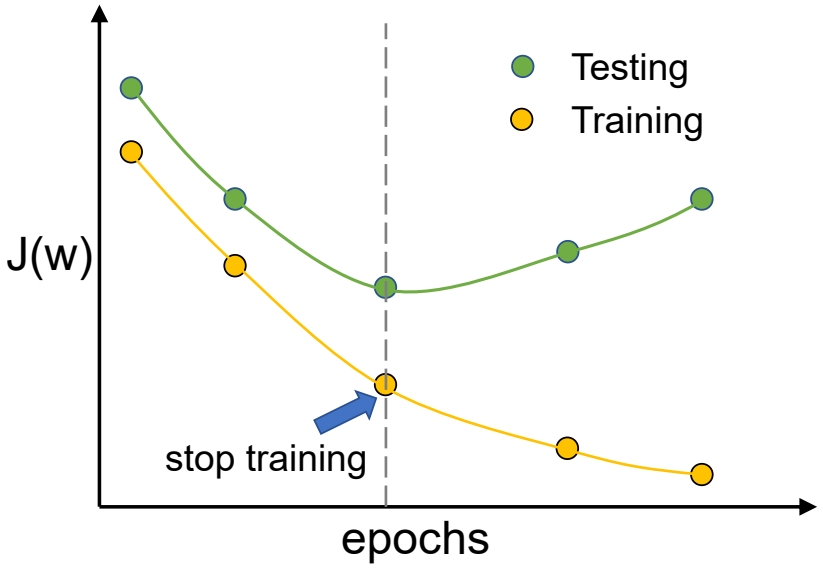


Figure 2.10: Example of the strategy called "early stopping"

Chapter 3

Channel Prediction

3.1 Introduction

In literature, there are countless methods available for estimating channel properties of a communication link, collectively known as channel state information (CSI). These techniques generally fall into one of these two categories: channel prediction schemes based on traditional statistical methodologies or methods designed around machine learning algorithms, with particular attention on neural networks-based prediction [30, 31].
This work will focus on the latter with the aim of building a predictor that can estimate the future quality of a wireless channel between two devices in real-time. Specifically, the target is to predict with what probability a packet sent over the channel arrives at its destination. The idea is to train an ANN model using the knowledge of current and past transmissions outcomes to determine the probability of success of future ones. Below is reported an example that will go through the main steps of the algorithm. First of all, let us introduce two variables: i and p_i. The former denotes a counter that is incremented after each transmission attempt, while the latter indicates its outcome (success or failure). The counter has an initial value of 0, so if i is equal to 10, the transmitter has sent ten packets on the wireless channel. In order to predict what will happen to the 11th packet, the neural predictor will use a mathematical function, learned during training, that accepts as input the results of the last transmissions (e.g., the 10th, 9th, and 8th) to estimate the sought-after measure. The size of the data considered each time is delimited by the left and right edge of a sliding window with no holes (see Fig. 3.1), which determines respectively the oldest and newest packet examined. The window's width is one of the model's parameters under study, which changes from channel to channel.
Once the final model is ready, i.e., after it has been trained and tested, we are not limited to working only with the success rate, but we can derive from it other metrics, such as the loss ratio.

3.2 Methodology

This section will precisely describe the quantities of our interest and outline the main features of a new type of predictor used as a reference model in the testing phase. Below we introduce some of the notations that will be used in the next few paragraphs:

- let p_i denote the outcome of the *i-th* transmission over the wireless channel connecting a *transmitter* and a *receiver*. In probabilistic terms, it denotes an *event*. It has value 1 in case of success, 0 otherwise. The index can be viewed also as the time of transmission. The two interpretations are equivalent in that both denote a particular event within an ordered sequence, where the concept of before and after applies. Both meanings are used interchangeably in this text
- let $P_{N_{events}}$ be $\{\, p_i \mid i \in (\, 1, 2, \ldots, N_{events}\,)\,\}$, where N_{events} is the size of the set containing all events
- let P_t be $\{\, p_i \mid i \in (\, t, t-1, \ldots, t-N_{past}+1\,)\,\}$, where N_{past} is the width of a window delimiting a subsequence in $P_{N_{events}}$
- let D be a set of tuples $\{\, (X_t, y_{true_t}) \mid t \in (1, ..., s)\,\}$, where X_t represents the *t-th* input vector, y_{true_t} is the associated label, and s is the size of the set D. In other words, D is our *dataset*
- let N_{next} be the size of the target window, which outlines the future transmissions we are interested in in the form of a sliding window.

As previously anticipated, the objective is to create an ANN-based model, mainly using FFNNs, such that it can predict with reasonable accuracy the success rate in the near future given the recent past. Note that unlike the example given in the introductory section, the subject of interest is the near future and not the outcome of the next frame. We made this choice because it is incredibly challenging and complicated to create a predictor that can reliably estimate the outcome of a single transmission due to the inherent issues affecting a wireless channel, e.g., noise. The near future, henceforth called *target window*, is delimited by a window of width N_{next}, similarly the recent past is bounded by a window of width N_{past} (see Fig. 3.1). A generic target window at time t is represented as the arithmetic mean over the N_{next} consecutive events from time $t+1$. Mathematically this is expressed as:

$$y_{true_t} = \frac{1}{N_{next}} \sum_{i=1}^{N_{next}} p_{i+t} \tag{3.1}$$

The equation above represents what we called *labels* in Chapter 2. They are used to assess the goodness of an ANN's predictions by comparing them with the target

values. The comparisons are made in two instances: during training while the algorithm is still "learning" and during the test phase when validating the final model.

As for the inputs of a neural network (i.e., X_t), they are vectors whose elements are called *feature*. In turn, a feature is a simple arithmetic mean of a variable-size block of transmission outcomes. A different window characterizes a different feature of an input vector, with all having a width multiple of N_{step}^{ANN}. The largest one is described by the parameter N_{past}^{ANN} (see Fig. 3.1). So, the *k-th* feature of an input vector X_t is defined as follow:

$$x_t^k = \frac{1}{N_{past_k}^{ANN}} \sum_{i=1}^{N_{past_k}^{ANN}} x_{t-i+1} \tag{3.2}$$

where k identifies the *k-th* feature of an input vector (counting from 1), $N_{past_k}^{ANN}$ $(= k * N_{step}^{ANN})$ is the width of the window, and t identifies the time/example.

For instance, to produce an input vector $X_t = \{x_1, x_2, ..., x_{N_{features}}\}$ we would need to apply (3.2) on a window of length N_{step} to get x_1, on a window of length $2 \cdot N_{step}$ to get x_2, and so on until $x_{N_{features}}$, which has a window of length $N_{features} \cdot N_{step}$ (N_{past}^{ANN}).

Now that we have the starting blocks in place, we need to be able to answer two basic questions before we can proceed any further: how can we assess if a model/predictor is good or not? And, what does it mean "good"? Generally speaking, the closer the actual predictions are to the labels (henceforth "actual" values/predictions refers to the prediction made by the model), the better a model is considered to be. The concept of closeness is quantified in terms of distance, or rather error, between the labels and actual values. For regression problems such as this one, we typically would use as metrics:

$$MSE = \frac{1}{s} \sum_{i=1}^{s} e_i^2 \tag{3.3}$$

$$MAE = \frac{1}{s} \sum_{i=1}^{s} |e_i| \tag{3.4}$$

where s is the number of examples or equivalently is the number of *labels*, and e_i is the difference between y_{true_t} and the actual value generated by the ANN-based predictor given X_t as input. MSE and MAE are acronyms respectively for "mean squared error" and "mean absolute error".

For conciseness, we introduce a function $f^{ANN}(\cdot)$, that summarize the operation performed by an ANN model, i.e., given an input vector X_t produce a prediction $\hat{y}_t^{ANN}$. Then we can reformulate e_i as:

$$e_i = y_{true_t} - f^{ANN}(X_t) \tag{3.5}$$

In order to fairly evaluate an ANN model, it is common practice to divide the dataset D in two disjoint subsets: the *training* dataset D_{train} and the *test* dataset

D_{test}. This is done because we want to ensure that the model is evaluated only on *unseen data* during the testing phase. Otherwise, we do not have a way to correctly interpret the metrics used to assess the predictor (e.g., MSE or MAE), on cases, for example, whether the model actually extracted some useful generalization pattern or simply memorized the entire training dataset (*overfitting*). With the help of MSE or MAE, we can numerically characterize only one aspect of a neural network: its accuracy. However, these metrics, by themselves, are generally not enough to determine if a neural network is overall good until it is compared to a reference point, which in turn sets the level of performance that we are trying to achieve or surpass. The baseline is usually set up a priori by the practitioner, and it varies from context to context, but, for example, it can be computed or extracted from similar past experiments. In other cases, the ANN is directly validated against other models that are functionally equivalent. The points of comparison can be multiple: from matching results using MSE or MAE to define custom metrics that better express the properties of the problem in question. In our case study, we followed the second approach by defining a heuristic-based predictor and then using a metric called win-ratio to compare the two.

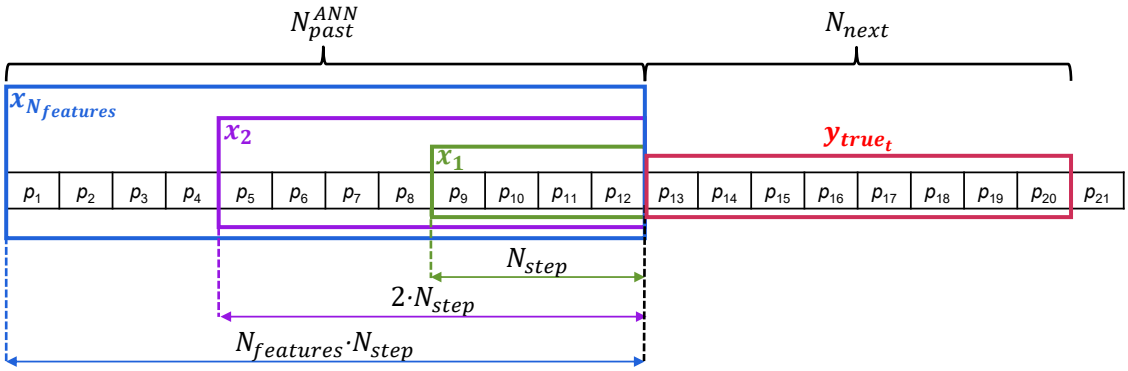


Figure 3.1: Input features of the ANN model when time t is 12 (example with $N_{\text{features}} = 3$, $N_{\text{step}} = 4$, and $N_{\text{next}} = 8$).

3.3 Heuristic predictor

The heuristic channel predictor was designed on the assumption that the dynamics of near future transmissions are similar, in terms of outcomes, to those of the recent past. Then, we can reliably estimate the success rate for a generic time interval t as an arithmetic mean of the most recent N_{past}^{AVG} transmission outcomes prior to t, analogously to how the features of an input vector are computed. However, instead of having multiple overlapping blocks, the heuristic model uses only a single block

(see Fig. 3.2). In mathematical terms:

$$\hat{y}_t^{AVG} = \frac{1}{N_{past}^{AVG}} \sum_{i=1}^{N_{past}^{AVG}} p_{t-i+1} \tag{3.6}$$

where N_{past}^{AVG} denotes the length of a window delimiting the set of past transmissions. Similar to what we did for the ANN model, we introduce a function $f^{AVG}(\cdot)$ that models the operation performed by the heuristic model, i.e., given a subset of N_{past} input samples, collectively denoted with P_t^{AVG}, provide an estimation of the target y_{true_t}. Hence, (3.6) can be rewritten as:

$$\hat{y}_t^{AVG} = f^{AVG}(P_t^{AVG}) \tag{3.7}$$

N_{past}^{AVG} is the only parameter of this type of model. The choice of its value is important and needs careful tuning since it will heavily influence the outcome of the final comparison between the two types of predictors. Ideally, we should set this parameter to a value that optimizes the metric/s present during the validation phase, so that we get the best possible heuristic predictor. In the case MSE is used, it means finding a value of N_{past}^{AVG} that minimizes the overall MSE computed over the training dataset. Note that the dataset employed for the above operation is the training set and not the test set, even though the latter is the one being used during the benchmark. The reason behind this choice resides in the fact that in real applications, we have at our disposal data with labels only during training. Let us now discuss how we can compute the optimal value from a practical point of view. The simplest of the methods is to use a brute-force-based approach, that is, to try all possible admissible values of N_{past}^{AVG} until the optimal solution is found. N_{past}^{AVG} is a parameter that depends on two other factors: the training dataset and N_{next}, i.e., the width of the target window. Whenever the values of either parameter are changed, we must recalculate N_{past}^{AVG} accordingly. Once we have selected the training dataset D_{train} and the value of N_{next}, the idea is to evaluate the MSE on D_{train} for all the values of N_{past}^{AVG} within the sequence $(1,2,\ldots,size(D_{train}))$, generating the predictions using (3.6). Then, we have to associate to N_{past}^{AVG} the window corresponding to the smallest MSE found in the previous step. More details have been presented in pseudocode form in Algorithm 2.

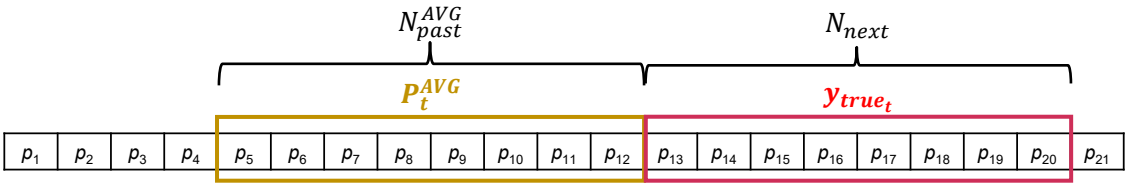


Figure 3.2: Input feature of the heuristic model when time t is 12 (example with $N_{\text{past}}^{\text{AVG}} = 8$ and $N_{\text{next}} = 8$).

Algorithm 2 $bestNPastAVG(D_{train}, N_{next}, computeMetric)$

$bestWidth \leftarrow 0$
$bestMetric \leftarrow null$
for $N_{past} \leftarrow 1$ **to** $N_{past} \leq D_{train}.size$ **do**
  initialize $targets$ and $predictions$ to an empty array
  for $t \leftarrow N_{past}$ **to** $t < D_{train}.size - N_{next}$ **do**
    $\hat{y}_t^{ANN} = \frac{1}{N_{past}} \sum_{i=t-N_{past}+1}^{t} D_{train}[i]$
    $y_{true_t} = \frac{1}{N_{next}} \sum_{i=t+1}^{t+N_{next}} D_{train}[i]$
    store $\hat{y}_t^{ANN}$ in $predictions$
    store y_t in $targets$
    $t \leftarrow t + 1$
  end for
  $currentMetric \leftarrow computeMetric(targets, predictions)$
  if $bestMetric \equiv null$ **or** $isBetter(currentMetric, bestMetric)$ **then**
    $bestMetric \leftarrow currentMetric$
    $bestWidth \leftarrow N_{past}$
  end if
  $N_{past} \leftarrow N_{past} + 1$
end for
return bestWidth

Algorithm 2 requires three arguments: a training dataset D_{train}, the width of the target window N_{next}, and a function $computeMetric(\cdot)$ that evaluates the $predictions$ against the $targets$ based on a user-selected metric, e.g., see (3.3) and (3.4). The function $isBetter(\cdot)$ is a placeholder for a procedure that semantically all it does is return *true* in case its first argument is "better" than its second argument according to the selected metric, *false* otherwise.
From the point of view of computational complexity, Algorithm 2, as any algorithm based on brute force, is inefficient and time-consuming. For large datasets, it means not converging to a solution in an acceptable time. Therefore in these instances, we need alternative approaches that are more efficient and faster, which opt to find an approximate solution to the problem at hand instead of an optimal one. For example, in our case, instead of examining all window lengths, we can test only a subset of them. In particular, we can only try the lengths that are a multiple of a base x, effectively reducing the number of executed iterations by x times. This implies increasing the speed of the algorithm by x fold. As a further improvement, once we have found a solution, denoted as $approximateBest$, we can reapply Algorithm 2 within the interval $(approximateBest - x, \ldots, approximateBest, \ldots, approximateBest + x)$ to fine-tune our current solution.
In summary, the purpose of the heuristic approach is to set a valid and meaningful baseline to be used during our tests. To this end, the heuristic model is initially

evaluated on a training database D_{train} to find a specific value of N_{past}^{AVG}, that minimizes the MSE. As for the predictions, the algorithm generates the estimate for a generic time t by averaging the N_{past}^{AVG} samples before t, as we did similarly for the targets.

Chapter 4

Database Acquisition

4.1 Introduction

This chapter describes the setup and techniques used to acquire the databases used for training and testing the ANNs and heuristic model. The general scheme adopted is the following: a commercial desktop PC sends frames at a constant and continuous rate, on two separate Wi-Fi channels through two disjoint Wi-Fi adapters, to an access point placed at a fixed distance. We will refer to this particular configuration as *basic setup*. As explained in Chapter 3, the pieces of information we are interested in are the outcomes of every transmission made on a Wi-Fi channel between two devices (i.e., PC and access point) starting from an arbitrary point in time. To achieve that, we have built an ad hoc software application that saves the identifiers of all frames transmitted over the channels and their relative outcomes in a database system. It is built on top of an architecture called SDMAC [32, 33], which defines a set of functions collected in a library that allows the calling applications to directly access and control some of the low-level operations performed by Wi-Fi adapters, such as automatic frame retransmission. It is necessary since most of these operations are generally not accessible for applications running in user space, such as this one. Using these specific MAC-level functions, we can precisely control how and when a frame is transmitted over a wireless medium. Conceptually the main application performs three macro functions, executed in the following order:

1. every T_s seconds, the application sends a request to the Wi-Fi adapter to transmit a frame to the access point via a specific channel. This operation is accomplished by calling the function $SDMAC\_DATA\_req(\cdot)$

2. the application hangs and waits indefinitely until it receives a notification about the transmission outcome ($success$ or $failure$). This operation is performed by calling the function $SDMAC\_DATA\_con(\cdot)$

3. the application writes the statistics of interest regarding the request just made in a database system.

$SDMAC\_DATA\_req(\cdot)$ and $SDMAC\_DATA\_con(\cdot)$ are functions found inside the SDMAC library [33]. Then the application repeats the three steps above till it gets the complete database, e.g., stop after a predefined number of transmissions. We will discuss below, in general, the basic ideas and implementation choices made by the SDMAC authors during its design and the specific options that were adopted while writing the main application. As already mentioned, to send a packet, the application calls the function $SDMAC\_DATA\_req(\cdot)$, which in turn, all it does is primarily to call an endpoint (*sendto*) exposed by the POSIX raw sockets API. Passing through the driver, the packet is then temporarily saved inside a queue before being sent to its destination by the adapter. The transmission queue, also called the ring buffer, is a kind of loading-unloading zone present in all modern Wi-Fi adapters containing the packets to be forwarded, stored in a particular order. In this first phase, several parameters and mechanisms of the Wi-Fi standard deserve special attention during the configuration stage: backoff, transmission speed, and the retransmission process. The latter is a mechanism that deals with retransmitting lost or damaged packets due to transmission failures. One of the most common causes is the expiration of a timer before receiving an acknowledgment for the packet. Under normal circumstances, the transmitter expects to receive a confirmation from the recipient following a transmission. If this does not happen within a time interval, the packet is considered lost from the transmitter's point of view, triggering the retransmission mechanism. Concerning our work, we disabled this mechanism for the following reasons:

- ensure that each packet is sent exactly every T_s seconds. With retransmissions enabled, retries may be forwarded at inconsistent rates

- guarantee that only one packet is present in the ring buffer at any given time. This aspect is important because among the various statistics collected, some track the temporal properties of each transmission. These values could be distorted if there are other packets in the queue. For example, the time at which the request is sent could be very different from the actual transmission time if the Wi-Fi adaptor is busy disposing of lost packets. Theoretically, we could run into this kind of problem also for small values of T_s. However, in practice, this does not happen because T_s is usually in the order of seconds, while packet transmission time is in the realm of microseconds. Nevertheless, the latter will tend to increase if the dynamics of retransmissions come into play, increasing the delay to even a few seconds in cases of high workload.

By disabling it and setting the parameter T_s to an appropriate value, we guarantee that requests are sufficiently equispaced so that there cannot be more than

one packet at the same time in the ring buffer. In order to decrease latencies and further improve the accuracy of time measurements, the random backoff was disabled. It is a mechanism present in the Wi-Fi standard that serves to minimize the probability of collision. Collision refers to the event in which one or more devices want to transmit a packet at the same time. In these cases, the device which has detected the channel as busy waits for a random time, called backoff time, before attempting a new transmission. Deactivating the random backoff is equivalent to transmitting the frame immediately after the channel becomes idle. The last relevant parameter to which we made a change is the transmission rate. In modern Wi-Fi adapters, the transmission rate is adjusted on the fly by adaptive mechanisms, which decide the transmission rate for each frame. The drivers in our Wi-Fi adapters use a rate adaptation algorithm called Minstrel. The way it has been conceived and designed, Minstrel is suitable when the application context makes use of retransmission procedures [34, 35]. Since the latter has been disabled for the aforementioned reasons, it only makes sense to disable this feature. Consequently, the transmission rate must be managed on the application side (with SDMAC). The authors of SDMAC, through experimental studies, have shown that having a fixed bit rate helps to improve the determinism of a system, and therefore in our case, to obtain more reliable and precise time measurements. For precisely these reasons, the transmission speed was set to 54Mbps, the same used in [33]. The next phase is concerned with detecting whether a transmission has been successful. Upon receiving an acknowledgment (success) or upon expiration of a timer (failure), the device generates an interrupt meant to inform the operating system that a response has arrived. Without going into implementation details that are beyond the scope of this thesis, the operating system will handle the interrupt as soon as possible by executing a predefined routine. The latter, in turn, will execute a set of instructions within the driver, including those used to detect the transmission's outcome and those to collect statistics associated with it, such as indicators of channel quality. All this information is then propagated in SDMAC from the driver to the application via a character device. Remember that in the meantime, the program is in an indefinite "waiting" state and remains blocked until it receives new data from the character device. Once the information has reached the application, it then proceeds to save it in a database.

4.2 Experimental setup

This section will present the settings of the testbed used during the database acquisition. The basic setup introduced in the previous section was duplicated in the final experiment. So in total, two desktop PCs were used to transmit to two different access points on four separate channels, two for each PC. To summarize, the application acquired the database by:

- using two PC desktops (transmitter A and transmitter B) with installed a Linux Ubuntu operating system (v.18.04 with kernel v.4.4.0) and equipped with a dual-band TP-Link TL-WDN4800 Wi-Fi adapter managed by the ath9k device driver (v. 4.4.2-1)
- having the sampling time T_s fixed at 0.5 seconds on all channels
- having the transmission speed set to 54Mbps
- having the retransmission process and Minstrel deactivated.

Transmitter A sends two packets every T_s seconds to access point A, one on channel 1 and the other on channel 9. Similarly, transmitter B transmits to access point B on channels 5 and 13.
For each transmission made, the application saves inside a database a record with the following information:

- channel name (ch1, ch5, ch9, or ch13)
- progressive number of a packet. It is global inside a file.
- timestamp in ms according to Unix time
- Time Stamp Counter (TSC) at the time of transmission. The TSC represents the number of CPU cycles from its reset at the time of packet dispatch.
- TSC at the time of ACK reception. Analogous to above but calculated at the time of ACK reception or at the expiration of the ACK timeout.
- outcome of the transmission. 1 in case of success, 0 otherwise
- received signal strength indicator (RSSI) of the ACK. It measures the strength of the received ACK. If the timer expires before the reception of the acknowledgment (known as ACK timeout), its value is undefined, thus unreliable.
- link quality according to iwconfig
- signal level according to iwconfig
- bit rate, which is fixed to 54 Mbps in the case of the database acquired in this thesis.

For various reasons (failures, interference, and disconnections), it was not possible to sample continuously on the same channel for the entire duration of the experiment, thus breaking the transmissions on each channel into many trunks. Then, the outcomes of the transmissions from each trunk were saved in a separate text file

having as name: `<channel>_<trunk>`, where <channel> and <trunk> are placeholders for the channel name and the trunk identifier.
Therefore the complete database is partitioned first by channel and then by trunks (see Fig. 4.1). Although only part of the information collected will be used during

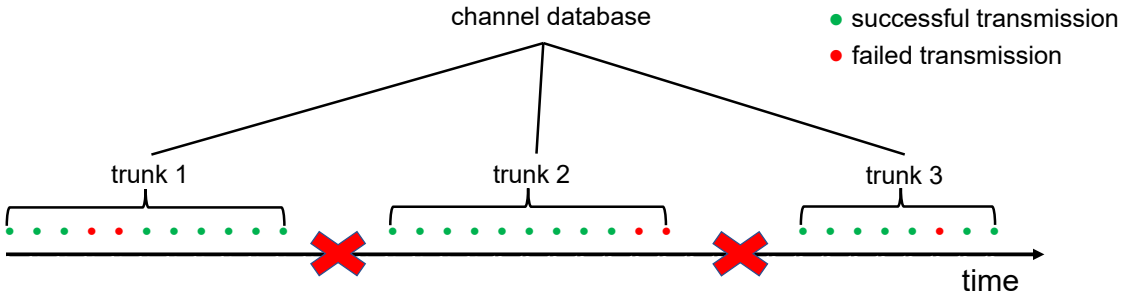


Figure 4.1: Transmission's outcomes of a channel over time

the training and testing of the predictors, the dataset represents a valuable source of information that can be used to conduct other experiments based on this work. The characteristics of the collected databases are summarized in Table 4.1.

channel	F_0 (MHz)	# files	# records/file	# records	# days	code name
1	2412	55	155365.46	8389735	48.55	db_ch1
5	2432	55	153239.07	8274910	47.89	db_ch5
9	2452	23	308187.30	7088308	41.02	db_ch9
13	2472	23	308025.30	7084582	41.00	db_ch13

Table 4.1: Some metrics about the collected database.

Henceforth to refer to the database acquired on a particular channel, we will use the corresponding abbreviation present in the "code name" column.
In summary, experimental data were acquired in an extensive acquisition campaign based on real devices, which lasted more than 40 days. We repurposed an existing application, based on the SDMAC architecture, to transmit periodically with a fixed period a frame on four distinct Wi-Fi channels operating in the 2.4 GHz band. We then collected and saved the various statistics related to each transmission, such as its outcome, into a database.

Chapter 5

Software

5.1 Introduction

This section describes the structure and the main components that constitute the software application. It is mainly about the design choices, the algorithms employed, and the various obstacles encountered during the code writing phase. In addition, we will expose the reasons why certain solutions were preferred over others for the most critical parts of the application. From a general point of view, the program consists of two macro components:

- an auxilliary software application that deals with processing the collected data and converting them to an ad hoc format that will then serve as a dataset to a neural network
- a main software application that takes care of configuring, training, and testing the neural networks

The two components will be explained in Section 5.2 and 5.3, respectively.

5.2 Auxilliary software application

The auxilliary software has the task of transforming the collected raw data, as described in Chapter 4, Section 4.2, into a dataset with an ad hoc format appropriate to be processed by a neural network during the training phase. A dataset can be seen as a table consisting of two columns and many rows: the first column, abbreviated as $\bar{X}$, denotes the inputs of a neural network (input vectors), while the second column, abbreviated as y_{true}, represents the labels associated with the inputs. $\bar{X}$ is a vector of real numbers whose elements are arithmetic averages computed over sequences of transmission outcomes made before a generic time t, likewise for y_{true} but considering transmissions from t. Conceptually, to compute

the *i-th* input vector, i.e., the first column of the i-th row, the application proceeds as follow:

1. creates an imaginary window that covers the sequence containing all transmissions from t to $t - N_{past} + 1$
2. divides it into many overlapping blocks of increasing lengths with their right side aligned. The smallest block has size N_{step}, the second smallest block has size $2 * N_{step}$, the third smallest block has size $3 * N_{step}$, and so on until the largest block, which has size N_{past}. Clearly, for this to work, N_{past} must be a multiple of N_{step}
3. computes the arithmetic average on each of those blocks to get the elements of the input vector.

Similarly, the label associated with the input vector is obtained by averaging the results of transmissions from $t + 1$ to $t + N_{next}$.
N_{next}, N_{past}, and N_{step} are the same parameters firstly introduced in Chapter 3. By convention, we will use x_1, x_2, ..., $x_{N_{features}}$ to denote the feature (element) computed on the N_{step}, $2 * N_{step}$, ..., N_{past} block, respectively, and refer to this specific format as *overlapping format*.

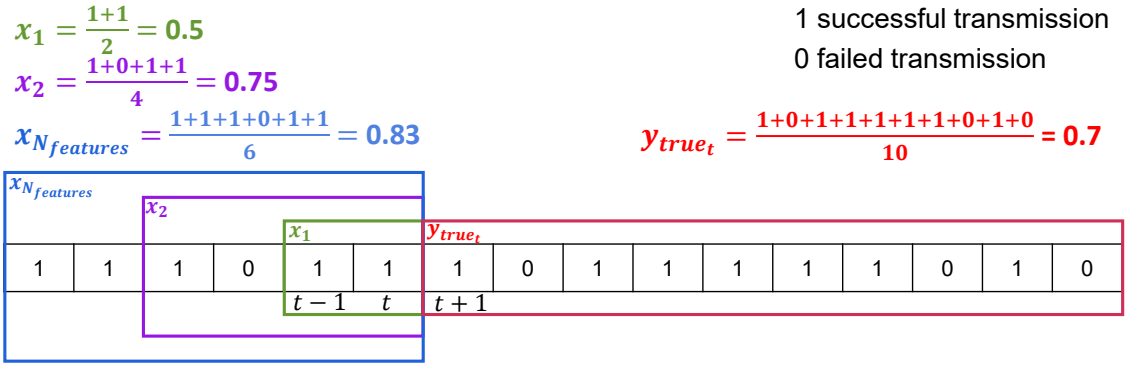


Figure 5.1: Computation of an input vector and the corresponding label (example with $N_{\text{step}} = 2$, $N_{\text{past}}^{\text{AVG}} = 6$, and $N_{\text{next}} = 10$).

From a practical point of view, the first thing to do is set the values of N_{past}, N_{step}, and N_{next}. All three take integer values. The last one, N_{next}, represents the width of the target window and expresses how far into the future we want to estimate the channel quality at each prediction. The first and second define how to manage past samples. Once these values are set, we can proceed with the primary phase of the algorithm. We will conceptually describe the steps involved by illustrating them with an example:

1. assuming we have chosen to work on *db_ch*1 (see Table 4.1), the initial step is to read in memory the first of the 54 files.

2. Step two is to select for each record of the dataset the field related to the result of the transmission and save it inside a vector, which we will denote as *ack_series*. The latter together with N_{step}, N_{past}, and N_{next} are the minimum information required to run the algorithm that maps the data into records with *overlapping format* as format type.

3. In step three, we proceed to extract from *ack_series* the features of all the input vectors, following this specific order: first, we compute all x_1 features, then x_2 features, and so on up to the $x_{N_{features}}$ features. The feature extraction is done by introducing a sliding window that has the same size as the block associated with the feature we are currently working on. For example, if we are extracting x_1, the window has size N_{step}. Also, the left edge of this window must be aligned with *ack_series* at position: $N_{past} - size(window)$, where $size(\cdot)$ denotes the size of its input. This is because it is the largest window, i.e. N_{past}, that characterizes the first valid input vector. Then we shift the moving window one position at a time to the right and each time repeat these two operations: sum the elements contained in it and save this value in a support vector. Having reached the end of *ack_series*, we divide each sum by the length of the window to obtain the feature values (see Fig. 5.2). To obtain the rest of the features, step three is repeated for all the other lengths. This strategy was preferred over calculating the averages directly because we can exploit some properties of our problem to calculate the sums more efficiently, thus gaining computational time—more on later.

4. In the final step, we save the input vectors obtained in a file.

The procedure just described is then repeated for the other files part of the raw database associated with channel 1. In the end, the final dataset for channel 1 will consist of 54 files, each equivalent to one (mini) dataset.

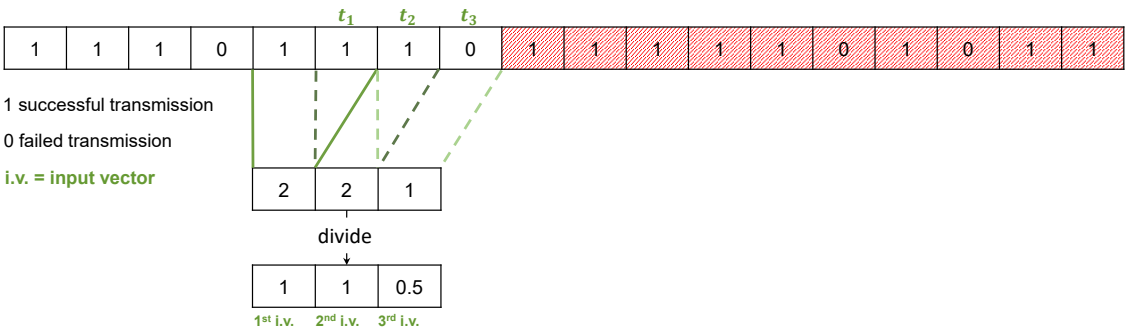


Figure 5.2: Computation of feature x_1 for all the input vectors (example with $N_{\text{step}} = 2$, $N_{\text{past}}^{\text{AVG}} = 6$, and $N_{\text{next}} = 10$).

The files have been named using this convention: `<channel>_<trunk>_<N`$_{\texttt{step}}$`,`

`N_past>_<N_next>_<format>_<version>`, where <version> represents a specific image of the raw database, while the other fields are self-explanatory.
The number of elements, i.e. input vectors, contained in a file can be simply calculated using this formula:

$$M = len(ack\_series) - N_{next} - N_{past} + 1 \tag{5.1}$$

where $len(\cdot)$ is a function that returns the size of its argument.

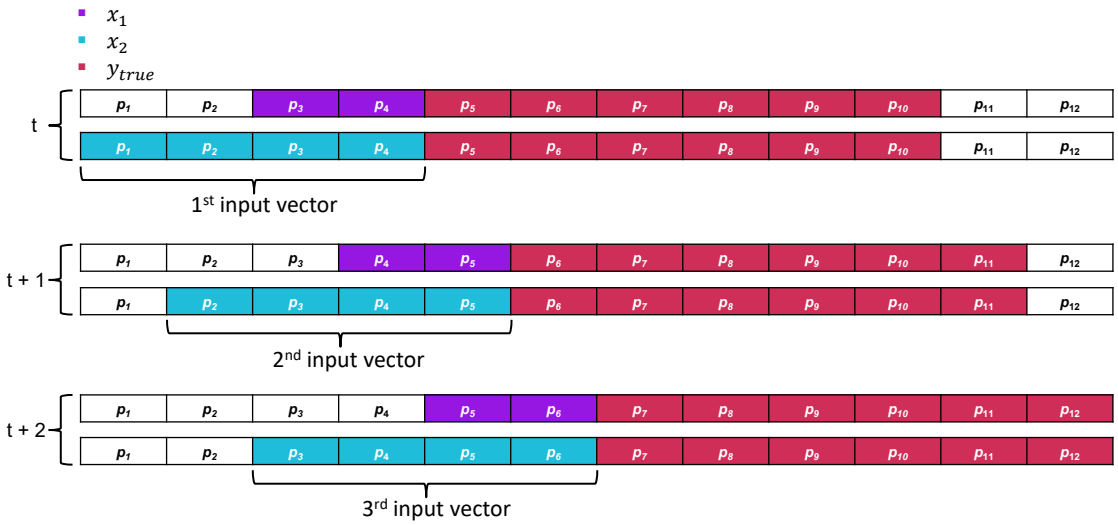


Figure 5.3: Visualization of all input vectors on *ack_series* (example with $N_{\text{step}} = 2$, $N_{\text{past}} = 4$, and $N_{\text{next}} = 6$).

All the reasoning did so far is valid for all other channels as well.
For block sizes that contain more than one element (i.e., $N_{step} > 1$), doing the sums by explicitly going to sum all the individual elements is computationally expensive, especially if *ack_series* is very large. An optimization we can implement arises from observing that part of a sum related to a window at a generic position has already been computed in the previous iteration. Specifically, the sum of the window at position t can be derived by adding and subtracting the *incoming element* and the *outgoing element*, respectively, to the sum calculated at position $t-1$ (see Fig. 5.4). With this optimization, the computational complexity of the algorithm goes from $\mathrm{O}(M \cdot N_{past})$ to O(M), where we used the big O notation to indicate the worst case of the algorithm's execution time, minus a constant. The algorithm in the simplest case has complexity $\mathrm{O}(M \cdot N_{past})$ because it has to perform in total M sums and each sum is computed on N_{past} elements, while in its advanced implementation it performs roughly $2 \cdot M$ sums. The latter arises from the fact that the moving window shifts M times and all the elements touched by the window, except the very first and very last, are used in two different calculations: a sum and a subtraction

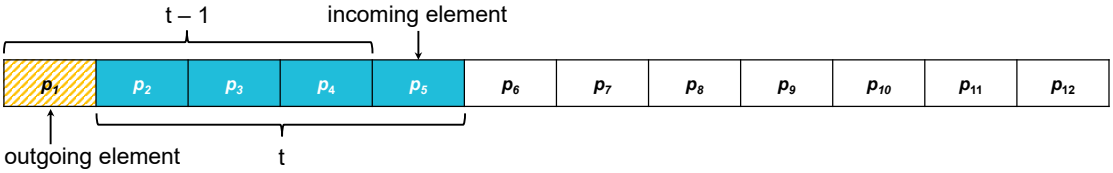


Figure 5.4: Visualization of the incoming and outgoing element when shifting the window to the right by one

(try to imagine having a window of size 1).
To recap, the goal of the auxiliary application is to transform the raw database into a tabular data structure, commonly known as a dataset, consisting of input vectors and labels. The input vectors are a collection of real numbers whose elements are arithmetic averages calculated on the transmissions outcomes made on the channel, as are also the labels. The final dataset consists of many separate files, each representing the transformation of a raw database file into tabular data structure.

5.3 Main software application

This section focuses on the structure and code of the main software application. The application was developed entirely in python for its simplicity, flexibility of experimentation, and availability of an extensive collection of open-source libraries for creating neural networks. In the following paragraphs, the explanations, where necessary, will be supplemented by code snippets to improve clarity and readability. However, the specific technological choices do not diminish the generality of our work, as the ideas and algorithms introduced are independent of the development platform and programming language in use. This section is further divided into three subsections that describe the configuration, the training, and the testing of neural networks, respectively.

5.3.1 Configuration

By configuration, we mean all that part that concerns the definition of the structure and behavior of neural networks, therefore: the parameters that define the topology (number of layers and neurons, type of connections), the hyperparameters (learning rate, batch size, momentum), the optimizers and initializers, and the datasets. In practical terms, the configurations are set up via two files that use a standardized format named YAML for setting the various parameters. This centralized strategy allows any user to use the application without having any a priori knowledge of its internal structure and code. The first of the two files ($\texttt{conf}_{\texttt{NN}}$) allows you to set the parameters related to the topology, optimizers, and initializers, specifically:

- the type of model. As of today, the application supports only sequential models (FFNN)
- the internal structure of each layer: type, number of neurons, activation functions, and weights initializer. Our experiments will only use dense layers, i.e., every neuron is connected to all neurons in the next layer. However, there are other forms of layers available in literature that have not been presented in this work but are essential for other types of neural networks, such as RNN or convolutional neural networks. Weight initializers define how the random weights associated with the connections are chosen
- the optimizer to use during training, e.g., SGD or Adam.

The second configuration file ($\texttt{conf}_{\texttt{training}}$) allows to control the settings related to the training, such as:

- the number of iterations on all the training dataset of a run (epochs)
- the batch size
- the learning rate. Two values are allowed: a real number that remains fixed for the entire duration of the training or a function, called scheduler, created by the user, which decides the values at the beginning of each epoch
- the datasets. These files are not declared explicitly, i.e., through their path, but are retrieved by the application through some other variables included in this configuration file. This is made possible because, as explained in Section 4.2, the filenames all follow a common naming convention that allows the application to generate their paths on the fly:

  $$\texttt{<channel>\_<trunk>\_<}\texttt{N}_{\texttt{step}}\texttt{, N}_{\texttt{past}}\texttt{>\_<}\texttt{N}_{\texttt{next}}\texttt{>\_<format>\_<version>} \quad (5.2)$$

  Each field $<\ldots>$ is present as a configuration parameter. The `<trunk>` parameter accepts a vector of real numbers that characterizes all the mini-datasets to be used in a single run of a training process.
  Another important aspect is that the user can define more than one pair of $\texttt{<N}_{\texttt{step}}\texttt{, N}_{\texttt{past}}\texttt{>}$, so that we can sequentially train many neural networks with different training datasets using the same configuration. This feature is essential in order to automate the training process and obtain a large number of different models for the analysis that will be conducted in Chapter 6
- the path to a folder containing all datasets used.

To summarize, the configuration phase allows the practitioner to define the neural networks and manage the training process by setting up two files, making the whole configuration process flexible and straightforward.

5.3.2 Training

Once the configuration parameters are set, the application proceeds to create and train a neural network based on these values. For this purpose, we decided to use a library called *Keras*, one of the most popular ones out there to interface with the world of neural networks. It was chosen for the ease of use and expressiveness of its API, which allows the end-user to easily and comfortably create neural networks as a composition of simple elements such as, for example, neurons, layers, and activation functions. As a whole, the algorithm is abstractly made up of a sequence of operations that are repeated until a criterion is satisfied, each one designed with a specific goal:

1. read and parse the two configuration files ($\texttt{conf}_{\texttt{NN}}$ and $\texttt{conf}_{\texttt{training}}$)
2. load in memory the training mini-dataset
3. extract the input features and labels from the training mini-dataset and eventually concatenate the rest of the features if more than one dataset has been declared, similarly, for the labels.
4. create a neural network based on the values of the configuration files
5. perform the training of the neural network
6. load in memory the test dataset. Eventually perform the same operation done for the training dataset if more than one test dataset has been specified
7. load the heuristic model
8. compare and evaluate the performance of the neural network versus the heuristic model
9. save the results of the comparison in a database.

The application repeats steps 2 to 9 if there is more than one pair of $\texttt{<N}_{\texttt{step}}\texttt{, N}_{\texttt{past}}\texttt{>}$. Below, we will discuss point by point how the various stages were implemented and the strategies adopted. As the initial step, the application reads, interprets and converts the information contained in the two configuration files into a data structure suitable to the programming language in use. The name and the position of the two configuration files are assumed to be fixed. Conversion is performed by a library called *yaml*, which allows to transform YAML type files into a collection of Python objects, known as dictionary. The result of this conversion is denoted by the variable `configs`. From the settings in $\texttt{conf}_{\texttt{training}}$, the application obtains at run-time the absolute paths of the datasets to be used during the training and testing phases. The paths are generated by concatenating the various parameters according to (5.2). A valid alternative is to indicate the file paths directly, but this

strategy has been discarded because it does not scale as the problem grows.
As the second step, the application reads the training mini-datasets in memory through a library called *pandas*, which saves the information in a table-like object, known as *DataFrame*. To this abstraction are associated the most common operations that we would expect to be able to perform on a table, such as reading and deleting specific rows, dividing the table by columns and rows, joining several tables and so on.
As the third step, the application merges the mini-datasets in the order in which they were read and divides the complete dataset into two parts: the inputs and the labels. The division is a simple process thanks to pandas because it is only a matter of selecting one of the two columns since, as explained in Section 5.2, the mini-datasets by construction are also tables made of two columns: one reserved for inputs and the other for labels.
The following two steps are the central parts of the algorithm. The application creates and trains the neural network based on the values read from $\texttt{conf}_{\texttt{NN}}$ and $\texttt{conf}_{\texttt{training}}$, respectively. At the time of writing this thesis, the application only supports sequential models (FFNN), i.e., loop-free neural networks composed of layers arranged one after the other, with a single input and output layer. To this end, the application first calls the function *keras.Sequential*(), which creates an empty network represented through the variable *model*. The various layers that make it up will be added to this model as they are created using *model.add*(·). To create dense layers, the *keras.layers.Dense*(·) function can be called, which allows to define among the various characteristics: the number of neurons, the activation function, the kernel initializer, the bias initializer. This is the minimum structure required in order to proceed to compile the model, which consists of a specific internal check in Keras to verify the correctness of the model just defined. The optimizer and the objective function (e.g., MSE, MAE) are declared in this phase. Clearly, these parameters vary depending on the type of problem we are trying to solve. For regression problems, such as this one, it is usual to use Adam as the optimizer and MSE/MAE as the objective function.

```
import tensorflow as keras

def create_model(configs):
    model = keras.Sequential()

    for _ in configs.tot_layers:
        layer = keras.layers.Dense(
            ..., # units
            activation=..., # activation function
            kernel_initializer=..., # weight initializer
            bias_initializer=..., # bias initializer)
```

```
        model.add(layer)

    model.compile(optimizer=..., loss=...)
```

Listing 5.1: Example of a function that creates a neural network using the Keras interface

Once we have obtained the model, we can proceed with the training of the neural network. It is done by calling the function $model.fit(\cdot)$ passing as arguments the inputs and the corresponding labels, the number of epochs for which we want to train the model, and the batch size value. In addition, we can specify whether to shuffle the inputs. The shuffle consists of randomly ordering the data at the beginning of each epoch before they are provided to the neural network for training. This operation is done to minimize the risk of creating batches that are not representative of the overall dataset. This problem becomes apparent in case a neural network is being trained using SGD. In fact, if the data are not randomized at each iteration, every data point is analyzed in the same sequence across all epochs, causing each gradient to be distorted and dependent on the update done by the data point before it. By shuffling, we ensure that each example updates the model independently without being influenced by the same inputs. The same concerns apply when using mini-batches. In this cases, ideally each batch should contain data points that are representative of the complete dataset, so that the model can then extrapolate useful patterns. Without the shuffle, the chances of the model fitting too well to a particular sequence of data increases, which may not be indicative of the overall trend of the complete dataset. For the reasons mentioned above, the shuffle is active by default in all of our configurations.

```
from tensorflow import keras

# example of a scheduler that halves the learning rate
# for the first 5 epochs, then keeps it constant
def lr_scheduler(epoch, lr):
    if epoch < 5:
        return lr / 2

    return lr

# for brevity, the code to obtain the inputs and the labels
# is omitted

model = create_model(configs)
model.fit(
    <inputs>,
```

```
<labels>,
epochs=<...>,
batch_size=<...>,
callbacks=[keras \
            .callbacks \
            .LearningRateScheduler(lr_scheduler)])
```

Listing 5.2: Pseudocode for training a neural network in Keras using a custom learning rate scheduler

5.3.3 Testing

Once the training is complete, we can proceed with the testing phase, which compares the above model's performance against a reference predictor based on a set of metrics and one or more test datasets. The goal of this comparison is to evaluate the quality of a neural network and to find a configuration that turns out to be better than the reference model. Generally, metrics estimate the quality of a neural network, and consequently of its predictions, by measuring the "distance" of the predictions from the target values, even if they do it in different degrees and forms. Thus, we can say that most of the cases, a model is considered better than another if its forecasts are, on average, closer to the target value. However, the definition may vary depending on the metrics and criteria applied—more on Chapter 6.
As the next step, the application loads the test datasets and the reference model into memory. The latter is the predictor designed following the procedure described in Section 3.3. Recall that for each group of test datasets, there is an equivalent heuristic model, built on the basis of the optimal value of N_{past}^{AVG} extracted from the training dataset. A group is considered different from another group if one of the following statements hold: the collections of the files used are not identical; the collections are the same, but the arrangement of files is different. In addition, the test datasets, like the training datasets, are also designed to use a parameter of N_{past}, denoted as N_{past}^{ANN}. The latter is chosen by the practitioner during the configuration phase and will almost certainly be different from the value of N_{past}^{AVG}. In order to properly compare the models, we need to realign the heuristic model with the test datasets so that the predictions for the same target windows are at the same locations. So in case N_{past}^{AVG} and N_{past}^{ANN} do not match, the application removes the first x inputs from either the test dataset or the heuristic model, whichever has the smallest N_{past} value. The number x is calculated as: $max(N_{past}^{AVG}, N_{past}^{ANN}) - min(N_{past}^{AVG}, N_{past}^{ANN})$, where $max(\cdot)$ and $min(\cdot)$ are functions that return the maximum and minimum of their arguments, respectively. Once this operation is completed, the test dataset and the heuristic model will have in the exact locations, respectively, the inputs and predictions to the same target windows.

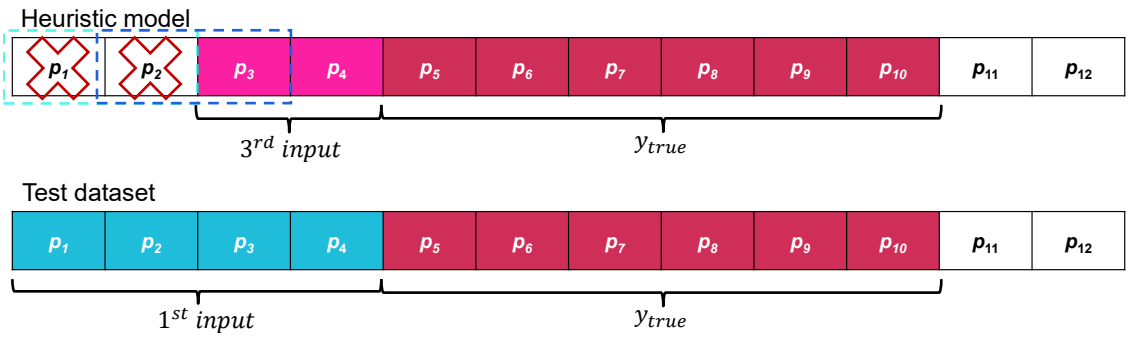


Figure 5.5: Realignment of the predictions

The next step is to extract the inputs and the labels from the test datasets, as it was done for the training datasets, and to use the trained ANN model to generate the predictions on the test inputs. This last operation is carried out by calling the *model.predict*$(\cdot)$ method.

```
# test_inputs are the inputs extracted
# from the test datasets
ANN_predictions = model.predict(<test_inputs>)
```

Listing 5.3: Example forecast generation

Having reached this point, the only data left of interest are the heuristic model, the ANN predictions on the test datasets, and the labels.
In this last step, the application compares the predictions generated by the ANN model with those of the heuristic model based on a set of objective indicators that are representative of the problem defined in Chapter 3. A total of five metrics were chosen: MAE, MSE, win-ratio, filtered win-ratio, and empirical distribution function of the errors (or equivalently its complement). This set of indicators are also known as key performance indicators (KPIs) in the literature. In general, the first two indicators are metrics that are chosen for their simplicity and provide a good preliminary indication of the overall quality of the models. To obtain their values, the application relies on two implementations made available by a popular library optimized for vector calculations called *sklearn*.

```
import sklearn

# code to obtain predictions and targets
# is omitted for simplicity
mae = sklearn.metrics.mean_absolute_error(<predictions>,
                                          <targets>)
mse = sklearn.metrics.mean_squared_error(<predictions>,
                                         <targets>)
```

Listing 5.4: Example on how to compute MAE and MSE using sklearn

Unlike the MSE and the MAE which follow the standard definition, *win-ratio* is a custom KPI that in its simplest version was defined as the percentage of wins by the neural network over the heuristic model.

$$win\_ratio = \frac{\#wins_{ANN}}{\#predictions} \tag{5.3}$$

For the same target window, a win is the event in which the prediction generated by the ANN is closer to the target than that of the heuristic model, or similarly, the error produced by the former in absolute terms is smaller than that generated by the target model. In practical terms, the application calculates the error generated by the ANN model and that of the heuristic model for each target window. Then it compares the values. If the error of the former, in absolute terms, is smaller than that of the latter, the algorithm increments the counter that keeps track of the number of ANN wins. In the end, this counter is divided by the number of target windows present within the test datasets. In pseudocode, this is translated to:

Algorithm 3 $computeWinRatio(predictions_{ANN}, predictions_{AVG}, labels)$

```
totalWinsANN ← 0
for i ← 1 to i ≤ labels.size do
    error_ANN = abs(predictions_ANN[i] − labels[i])
    error_AVG = abs(predictions_AVG[i] − labels[i])
    if error_ANN ≤ error_AVG then
        totalWinsANN ← totalWinsANN + 1
    end if
end for
return totalWinsANN / labels.size
```

The $abs(\cdot)$ function returns the absolute value of the given number.
The filtered win-ratio is a modified version that calculates the win-ratio according to (5.3), but only considering the predictions in which the error committed by either model is equal to or greater than a given minimum. For the purposes of our analysis, the application considers as minima the values belonging to a finite sequence, bounded both above and below. The plot of this function is handy and practical to carry out a graphical analysis of the trend of the performances of the ANN model against the heuristic one. The details of the procedure are explained in Algorithm 4.
Regarding the last metric, the empirical cumulative distribution function (eCDF) represents the distribution function associated with the errors committed in the predictions. It is defined as:

$$F(e) = \frac{\#\, errors \leq e}{n} = \frac{1}{n} \sum_{i=1}^{n} 1_{X_i \leq e} \tag{5.4}$$

Algorithm 4
computeWinRatioFiltered($predictions_{ANN}, predictions_{AVG}, labels, minima$)

initialize $errors_{ANN}$ and $errors_{AVG}$ to empty arrays of size $labels.size$
for $i \leftarrow 1$ **to** $i \leq labels.size$ **do**
  $errors_{ANN}[i] = abs(predictions_{ANN}[i] - labels[i])$
  $errors_{AVG}[i] = abs(predictions_{AVG}[i] - labels[i])$
end for
initialize $winRatios$ to an empty map
for all $minimum \in minima$ **do**
  $totalWinsFilteredANN \leftarrow 0$
  $totalPredictionsFiltered \leftarrow 0$
  for $i \leftarrow 1$ **to** $i \leq labels.size$ **do**
    if $errors_{ANN}[i] \geq mimimum$ **or** $errors_{AVG}[i] \geq minimum$ **then**
      if $error_{ANN} \leq error_{AVG}$ **then**
        $totalWinsFilteredANN \leftarrow totalWinsFilteredANN + 1$
      end if
      $totalPredictionsFiltered \leftarrow totalPredictionsFiltered + 1$
    end if
  end for
  if $totalPredictionsFiltered \equiv 0$ **then**
    $winRatios[minimum] \leftarrow 0$
  else
    $winRatios[minimum] \leftarrow \frac{totalWinsFilteredANN}{totalPredictionsFiltered}$
  end if
end for
return winRatios

where 1_{X_i} is the indicator of the error i and n is the total number of predictions. It shows for each value the fraction of the errors that are less or equal to the specified value.
Assuming that we have already computed the errors associated with a model's predictions, subsequently referred to as E_n, the data we need to plot its eCDF are two: a vector containing all distinct errors, i.e., without duplicate values, sorted in ascending order, and a vector whose components are the values obtained by applying (5.4) on E_n. The former and latter represent the x and y components of our graph, respectively. The application, in practical terms, performs the following operations:

1. performs an ascending sorting of E_n

2. removes the duplicates from E_n. These are the values of the x-axis. At the same time, it keeps track of the number of copies (frequencies) and associates

it with the corresponding element

3. perform the cumulative sum of the frequencies vector, i.e., each element is the sum of the previous elements, and divide all elements by the total number of predictions. These are the values of the y-axis.

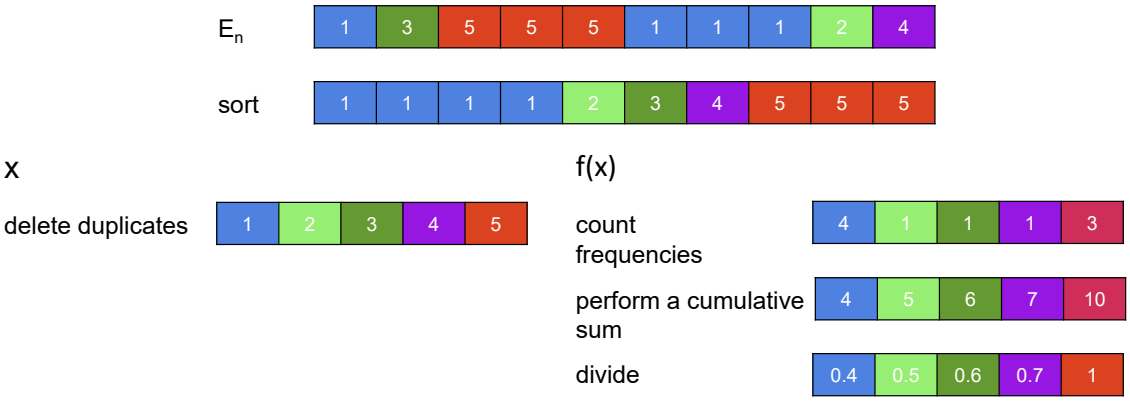


Figure 5.6: Visualization of the steps involved in the computation of an eCDF

To plot the graph, the application used the *matplotlib.pyplot.step*(·) function of the *matplotlib* library, the de facto standard library for plotting graphs in python.
The last step, which is optional but in many cases necessary, is to save the details of each experiment, e.g., configurations, metrics, and data, to a database for later use. The application writes the above information into a Google sheets spreadsheet via a library called *gspread*. The latter allows the client application to authenticate itself through a personal gmail account and perform read and write operations on a spreadsheet by contacting the appropriate google sheets API. Among the information that we saved, we find:

- the parameters of the neural network such as its topology, its hyperparameters
- optimizers and initializers used
- training and test datasets used
- heuristic model used for the comparison
- metrics obtained

5.4 Big data algorithm

Until now, we have overlooked a critical aspect in our analysis: the machine running our application. This machine, also known by the term *host*, can be an ordinary

desktop PC, a laptop, a remote machine, or even top-of-the-range computers like mainframes. Although there is an large selection of machines of different kinds and power nowadays, the devices are still limited by their hardware, which determines the amount of memory and computing power we can exploit. These limitations then affect the maximum performance that programs can achieve in terms of speed, power, and memory. In addition to having to execute the business logic correctly, computer software must manage these limitations to ensure that the program runs correctly and uninterruptedly. Otherwise, there is a severe risk of compromising the program's quality and incurring serious risks such as, for example, crashes and exceptions. Our application is not robust in this sense because it loads the data in memory without first considering the available space of the machine. In fact, until now, we have always assumed that the hardware characteristics of the host are sufficient to accommodate all operations performed by the application. However, in order for our work to have general validity, we must also consider more restricted working environments.

Generally, there are two popular strategies used to counteract this type of issue: either enhance the hardware specifications of the machine (scaling-up) or improve the efficiency of the algorithms that make up our programs. The first strategy is an acceptable solution if the problem does not grow over time and there are ways to expand the machine, which is not always the case. The approach generally used, including our case, is to improve existing algorithms or adopt functionally equivalent strategies that are more efficient and less expensive.

This section will discuss an alternative version of the algorithm seen so far, which is theoretically capable of working with an arbitrary number of datasets of various sizes.

Up to this point, training an ANN network has been delegated to the $model.fit(\cdot)$ method by passing all inputs and labels directly to it. Alternatively, the method also accepts a so-called generator, a simple python class that produces data on the fly as Keras requests it. For a class to be considered a generator by the library, it must have a well-defined structure; the easiest and most straightforward way to create one is to have a normal class extend $keras.utils.Sequence$ and implement the two methods: $\_\_getitem\_\_(\cdot)$ and $\_\_len\_\_(\cdot)$. Each time it is called, the former must return a new batch complete with inputs and labels, while the latter must provide the number of steps that make up an epoch, i.e., the number of batch iterations before an epoch is considered complete. Then, it will be the care of Keras to appropriately call $\_\_getitem\_\_(\cdot)$ to get the data during training.

How is a batch generated? Assuming we are working with more than one mini-dataset, ideally, we would like to generate batches by randomly extracting inputs and labels considering the union of all the datasets. However, since we assume that the machine cannot hold all the data in memory simultaneously, this in many cases involves reading from disk and discarding the same file multiple times within an epoch, creating an overhead that in many applications is unacceptable. To see why

this might be a problem, let us imagine we are in a situation like this: we want to train an ANN using two mini-datasets as input and that the host's memory is large enough to contain only one at a time. Moreover, we want to make sure that the batches are representative of the overall data, so we randomly select the elements from the entire dataset, i.e., union of the two. In the worst case, we may find ourselves in the situation where the batches must be extracted alternately from one file and then to the other. This implies reading one file, then discard it, then read the other file and discard it, and repeat the process until the end of the training. Every time the operating system loads a new file into memory, the application is idle and the train of the ANN is suspended. All these moments of waiting prolong the training process, slowing it down by about 5-20 times compared to the norm. This increase of 0-20 times is due to the different workload and the number of selected datasets, which then affects the number of I/O operations performed. It is well known that I/O operations involving the disk are expensive operations that negatively impact the running time of software. That is why we want to limit the disk accesses and execution of algorithms as much as possible with data already in memory. The advanced version of the algorithm was designed to minimize the I/O operations from disk, limiting us to read at most each mini-dataset once within an epoch. Conceptually, the algorithm is divided into 7 phases:

1. associates to each mini-dataset selected during the configuration phase an unique identifier
2. performs a random sorting of the identifiers
3. loads in memory the next C_{max} mini-datasets following the order determined by the identifiers. C_{max} is an integer that denotes the maximum number of files that the machine can load into memory at a time.
4. associates to each example (input vector plus label) loaded in memory a progressive identifier
5. shuffles the example identifiers
6. generates a new batch by selecting the examples according to the ordering of the identifiers. If there are not sufficient examples to extract, repeat the process from point 3
7. at the end of each epoch, repeats the procedure from step 2.

The basic idea is to use two sets of identifiers to simulate the global shuffle, then load into memory one set of files at a time, and extrapolate all batches from the data in memory before proceeding to load the next set of files. In this way, the application reads each file within each epoch exactly once. The disadvantage of this approach is that batches are not generated randomly as in the ideal case. However,

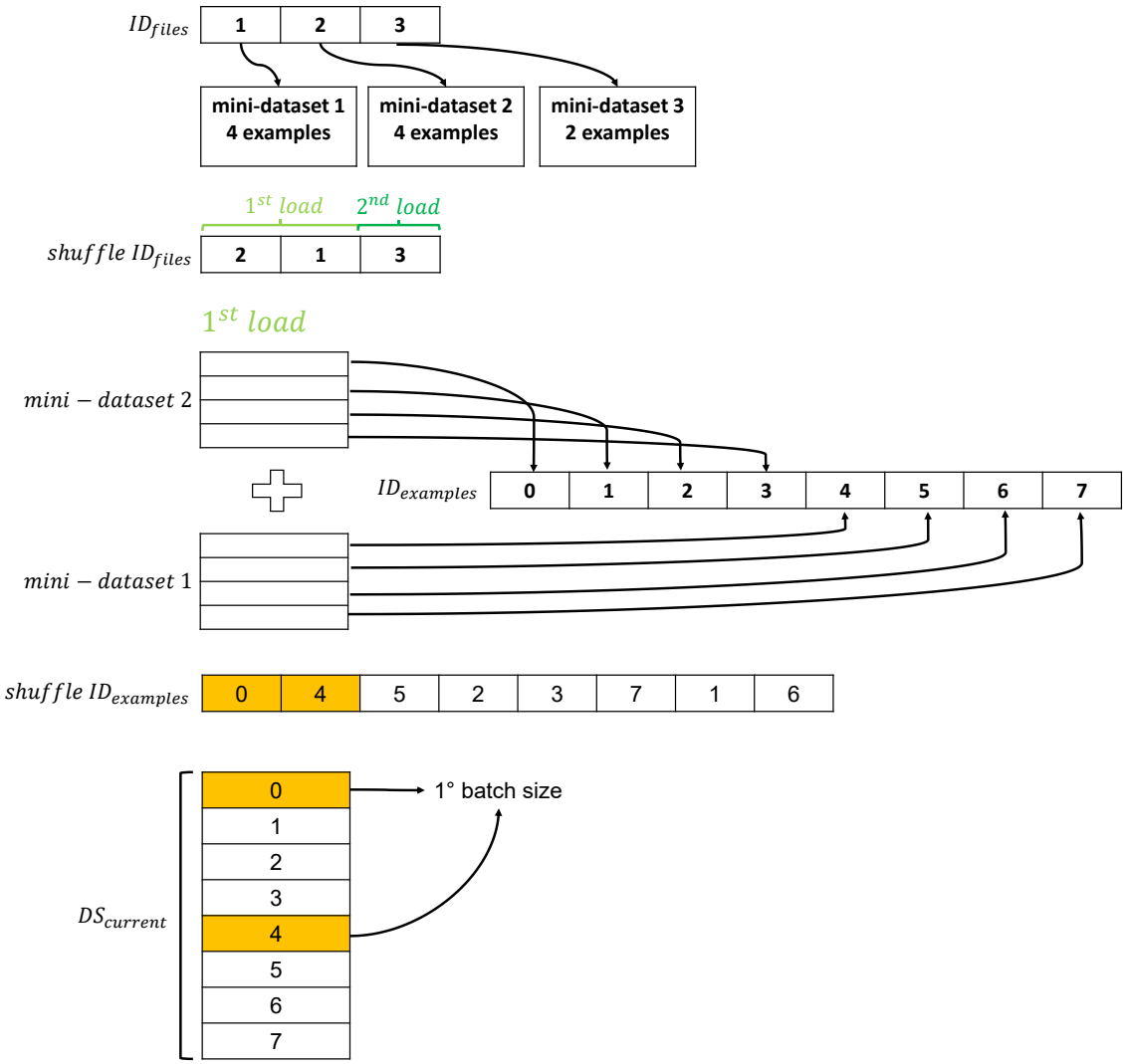


Figure 5.7: A full iteration of the big data algorithm with $batch\ size = 2$ and $C_{max} = 2$

we can decrease the magnitude of this problem by ensuring that each file contains a limited number of patterns. If each data pattern was saved in a separate file, executing step 2 would have the same effect as shuffling the complete dataset. On the other hand, limiting the file size causes an increase in the total number of files, and therefore also the number of I/O operations to be performed increases. So we have to find a tradeoff between size and quantity of files for the algorithm to work satisfactorily.

From a practical point of view, we need to modify both the auxiliary and primary applications. So far, each raw file has been converted, with a 1:1 ratio, into the corresponding mini-dataset using the auxiliary application. However, with the introduction of the big data algorithm, we will further divide each mini-dataset into

several equal parts. To this extent we introduced a new parameter called T_{max}, which denotes the maximum number of examples (inputs plus labels) that a file can contain. It is advisable to set it to a power of 2 because it will be helpful in the batch extraction phase to have it to a value that is a multiple of the batch size, assuming that the latter is also a power of 2. The total number of files we can obtain from a mini-dataset with T_{real} examples can be computed as:

$$\left\lceil \frac{T_{real}}{T_{max}} \right\rceil \tag{5.5}$$

where $\lceil \cdot \rceil$ is the ceiling function.

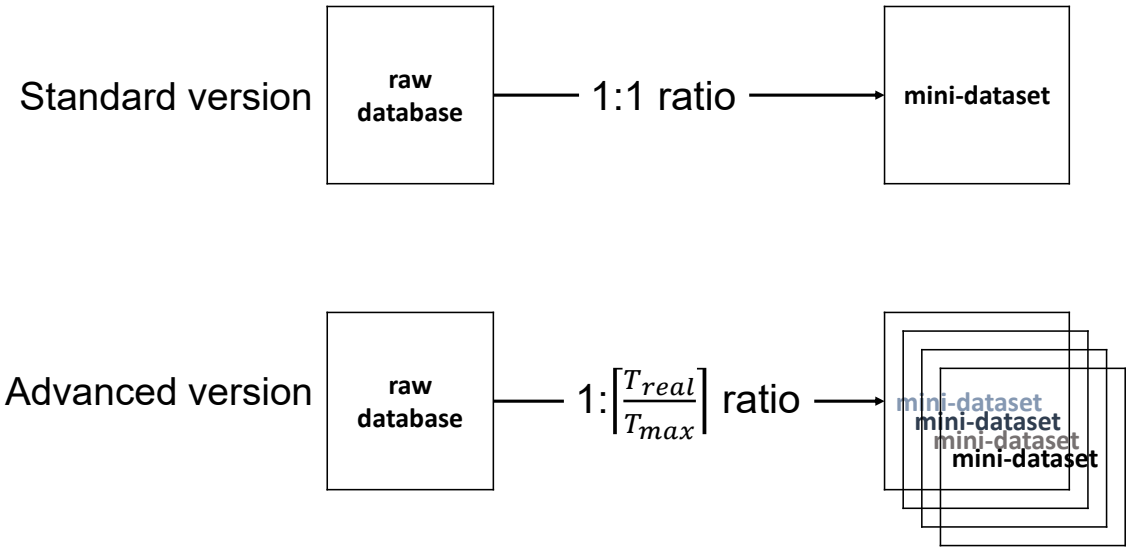


Figure 5.8: Comparison of the standard and advanced version of the auxiliary software in the transformation of a raw database file

All parts have exactly T_{max} examples except the last one, which has:

$$gp2le(T_{max} \bmod T_{real}) \tag{5.6}$$

inputs and labels, where $gp2le(\cdot)$ is a function that finds the greatest power of 2 less than or equal to its given argument. Using (5.6), we may end up discarding some data but, as we will see later, it will be to our advantage to have the sizes of all the files as a multiple of the batch size.
As for the primary application, the changes made concern steps 2-3 of the algorithm introduced in Section 5.3.2. Instead of loading all data into memory, the application instantiates a generator, a python class object that extends the *keras.utils.Sequence* class, which must implement the following two methods:

$\_\_getitem\_\_(\cdot)$ and $\_\_len\_\_(\cdot)$. Then this object is passed as an argument at the time $model.fit(\cdot)$ is called.

The generator life cycle is conceptually divided into three phases:

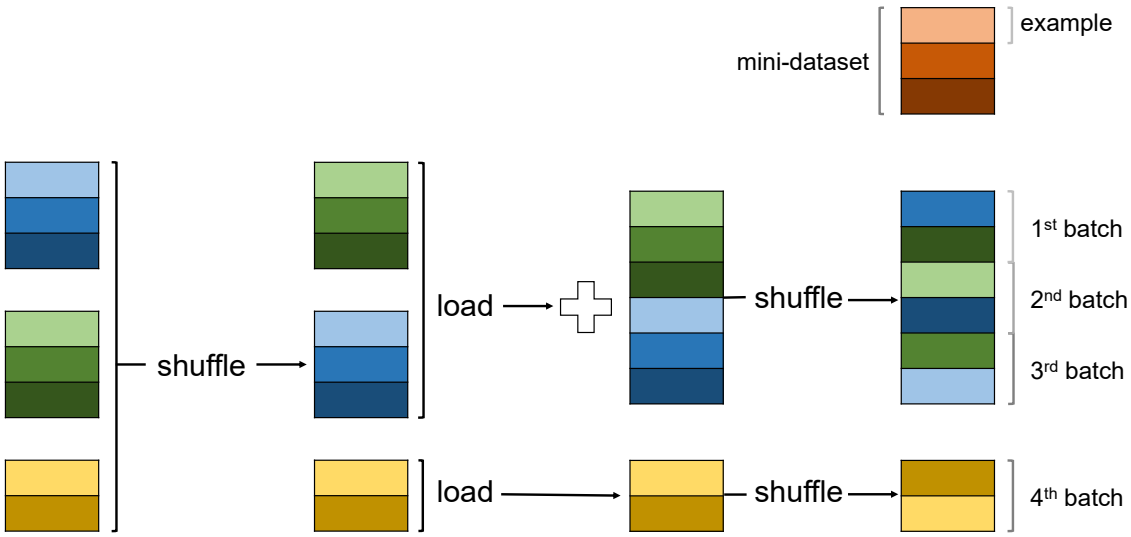
1. the initialization phase
2. the data loading phase
3. the batch generation phase.

Phase 2 and 3 are executed alternatively in an loop until the end of the training process. The goal of the first phase is to internally save and initialize the various arguments passed to the class constructor. Among the most important ones are: the batch size, the total number of epochs, the absolute paths to the mini-datasets, and C_{max}, a parameter denoting the maximum number of datasets to load into memory at a time. Each path is associated with a unique identifier, e.g., an integer, which remains constant for the entire life cycle of the generator. Henceforth this collection of identifiers will be referred to as ID_{files}. The order of the identifiers in ID_{files} represents the order in which files are loaded within a single epoch. For this reason, before proceeding to the following phases, we perform a shuffle on ID_{files}.
Phase 2 and 3 are implemented by $\_\_getitem\_\_(\cdot)$. In terms of pseudocode, the method can be summarized as follows:

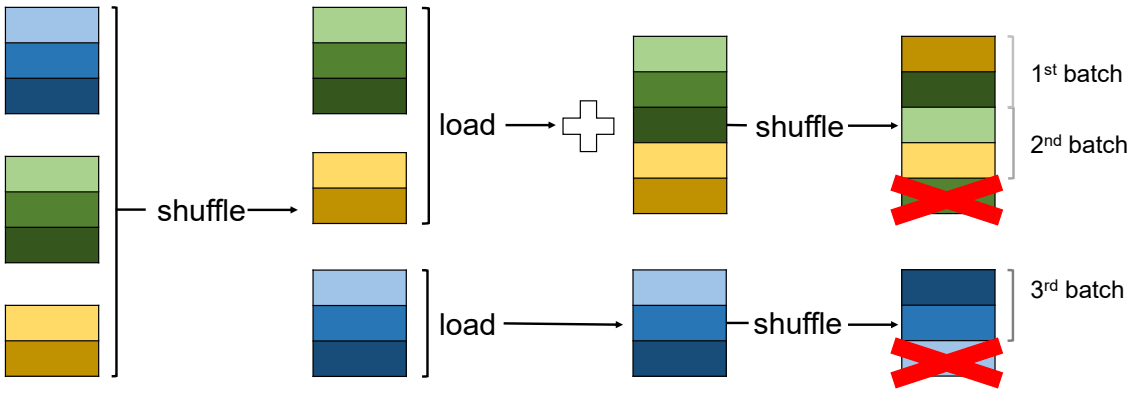
1. loads and concatenates the next C_{max} mini-datasets if there is no data in memory or no more batches can be extracted from the current data. Associate a progressive identifier, e.g., integer, with each example (input vector plus label) loaded into memory. We denote the concatenated dataset and the set of identifiers, respectively, with $DS_{current}$ and $ID_{examples}$. Then, we perform a shuffle on $ID_{examples}$
2. select the next *batch size* examples from $DS_{current}$. It is advisable to save $DS_{examples}$ as a vector/matrix or a similar structure such that the identifiers can be used as indices to get the elements of a batch.

The generator implements an extra method in addition to those already mentioned, conventionally called $on\_epoch\_end(\cdot)$, which Keras invokes at the end of each epoch. In this segment, two operations are performed in preparation for the next epoch: the shuffle on ID_{files} and the memory release of the current data.
The last useful method is $\_\_len\_\_(\cdot)$, which has the task of informing Keras about how many steps an epoch is made of at the beginning of the training. The value is calculated in the initialization phase by dividing the sum of the sizes of each mini-dataset by the batch size value. For the algorithm to work correctly, the value returned by $\_\_len\_\_(\cdot)$ must match the actual number of batches that are created within each epoch. In some cases, these two values may differ depending

on the order in which the mini-datasets are loaded. To see why let us look at an example: let us assume that we have three mini-datasets available that contain respectively 3, 3, and 2 examples, a batch size of 2, and that we can load into memory at most two files at a time ($C_{max} = 2$). According to what has been said so far for $\_\_len\_\_(\cdot)$, the method will return 4 as a result of $\frac{3+3+2}{2}$, which represents the number of batches that Keras expects in each epoch. The number of steps is actually 4 only if the files are loaded in this sequence: first the two files with the 3 examples, then the one with 2 (see Fig. 5.9a). In other combinations (3, 2, then 3) or (2, 3, then 3), we would only have 3 complete batches in total because of the way the algorithm is designed (see Fig. 5.9b).



(a) *batch size* $= 2$ and $C_{max} = 2$



(b) *batch size* $= 2$ and $C_{max} = 2$

Figure 5.9: Batch extraction without discarded examples (top picture) and with discarded examples (bottom picture).

That is why it was suggested, at the beginning of the section, to set C_{max} to a power of 2 because we want to ensure that the size of each file is a multiple of the batch size. So, no matter how the mini-datasets are loaded, we would have the same number of batches in each epoch. Obviously, we assume that the batch size is also a power of 2, which is almost always the case.
In summary, in this section, we presented the big data algorithm, which is an improved version of the algorithm introduced in Section 5.3.2, that considers the memory limitations imposed by the machine hardware by adopting solutions that extract the training data on the fly instead of preloading them all in memory.

Chapter 6

Results

6.1 Introduction

This chapter illustrates and analyzes the results obtained comparing the performance of the ANN models with the heuristic models on the basis of multiple test datasets. The latter were built using the procedure described in Section 5.2, using the data collected during packet transmissions over multiple Wi-Fi channels. For each test dataset, we created a corresponding heuristic model that acts as one of the reference model for our benchmarks. Then, we ran these tests on a set of ANN models, each characterized by a different configuration of one or more hyperparameters, e.g., value of N_{past}^{ANN}. Before we begin with our analysis, we introduce below the indicators that are used to assess the quality of ANN and heuristic models:

- MSE: $\frac{1}{N}\sum_{i=1}^{N} e_i$, where e_i denotes a prediction error and N is the size of a test dataset.
- MAE: $\frac{1}{N}\sum_{i=1}^{N} e_i^2$. Same as above.
- win-ratio. It shows the percentage of times a model's predictions edged out the other model's predictions in terms of accuracy (see Algorithm 3).
- win-ratio filtered. It is an alternative version of the win-ratio, in which it takes into account only prediction errors greater than a threshold value (see Algorithm 4).
- eCDF. It computes the cumulative density function of the prediction errors of a model (see definition 5.4). In other words, given a value x, it returns the ratio of prediction errors that are less than or equal to x. We model this indicator with the function $ecdf(\cdot)$.
- complementary eCDF (eCCDF). As the name suggests, it is complementary of the eCDF, i.e., it returns the ratio of prediction errors that are greater than

or equal to a given value x. It is defined as: $cecdf(x) = 1 - ecdf(x)$, where we used $cecdf(\cdot)$ function to model this indicator.

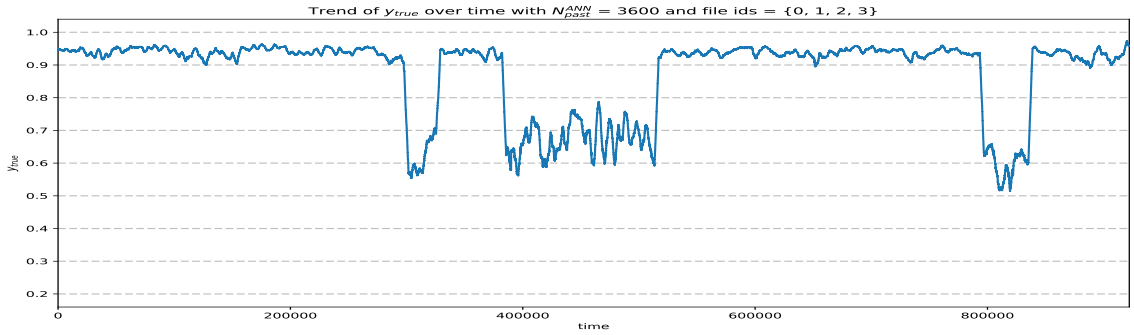
6.2 Test lists

Regarding the data, as shown in Table 4.1, the raw database is divided into two levels: the first level is composed of 4 parts, one for each Wi-Fi channel we transmitted on, while the second level denotes the number of files that make up that specific channel. The datasets were derived by transforming each file into a corresponding and distinct (mini) dataset using the procedure described in Section 5.2 via the auxiliary application. Relative to the test datasets, they are merely collections, even of one element, of mini-datasets. They were designed with the goal of creating tests that were unique, varied, and most importantly, representative of the various quality states in which the 4 channels were found during the sampling periods. The ultimate aim is to quantify, on the one hand, the ability of our models to generalize and, on the other hand, to demonstrate the (in)effectiveness of the methodology described in Section 3.2 in the practical domain. For each channel, we created test datasets by selecting a subset of the mini-datasets associated with that particular channel. We will list below the identifiers of the mini-datasets that were used in each test dataset:

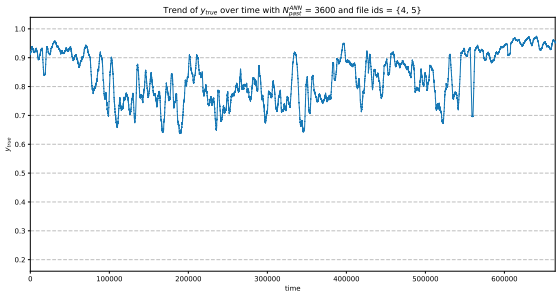
Channel	Testing file IDs	# y_{true}^{3600}	# Days	Code name
1	0, 1, 2, 3	922053	5.42	list 1
1	4, 5	664283	3.88	list 2
1	6, 7, 8, 9, 10	852793	5.04	list 3
5	0, 1, 2, 3	911552	5.36	list 4
9	0, 1, 2, 4	446435	2.67	list 5
9	9	1313119	7.62	list 6
13	0, 1, 2, 4, 5	1152996	6.78	list 7

Table 6.1: Test list collection

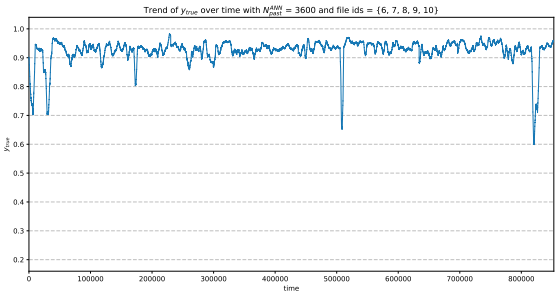
y_{true}^{3600} denotes the number of targets available inside a list with $N_{next} = 3600$, # Days represents the sampling duration, while the IDs specify the files that belong to a list (we used the same convention applied in Table 4.1). For each channel, we constructed the experiments such that we cover the target windows of 5, 10, and 30 minutes, which is equivalent to setting N_{next} to 600, 1200, and 3600, respectively. In general, to derive the number of targets within a list for any value of N_{next}, we can use this formula: $\#\ y_{true}^{next} = (3600 - next) \cdot (\#\ files) + (\#\ y_{true}^{3600})$, where $\#\ files$ denotes the number of files inside a list.
The plots of the test lists covering channel 1 are shown in Fig. 6.1. The three test lists each represent a very different qualitative state of channel 1 found during



(a) Test list 1



(b) Test list 2



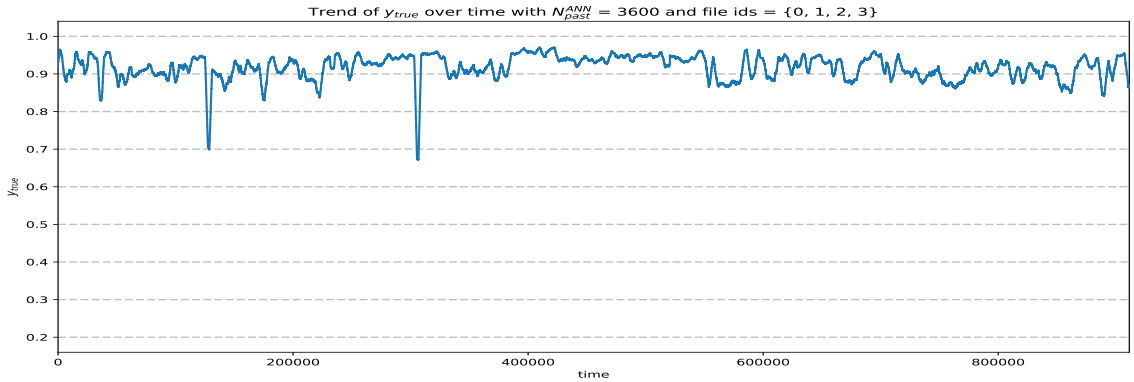
(c) Test list 3

Figure 6.1: y_{true} trends of the test lists associated with channel 1.

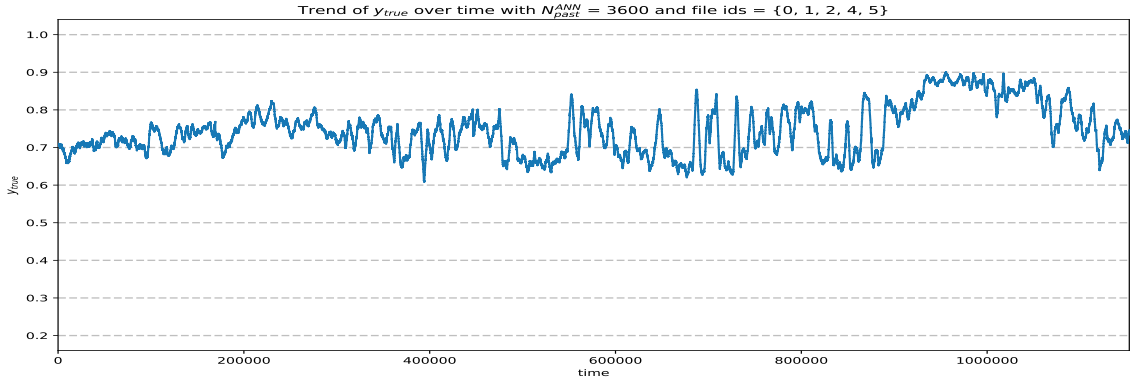
the sampling period: the target in lists 1 and 3 has a generally stable trend, with some square wave variations in the former and steep drops in the latter. While in list 2, the target displays an irregular shape. Similarly, the remaining lists have been chosen for their particular trend that varies from regular to sudden dips to irregular ones (see Fig. 6.2 and 6.3). A common feature of all the test lists is that as the value of N_{next} is decreased, the trends of the targets tend to be less outlined and more irregular. However, the overall appearance of a specific list remains unchanged as N_{next} varies (see Fig. 6.4).
Section 6.3 illustrates the results obtained using the standard algorithm with a reduced database, while Section 6.4 shows the results obtained using the big data algorithm with the complete datasets.

6.3 Preliminar analysis

As mentioned in the previous section, each experiment is characterized by a unique combination of test list and N_{next} value. Each of them is associated with a unique and distinct reference model, which we have called *heuristic model*, which serves as a baseline for our tests. The object under test is the different ANN models trained in different conditions in terms of dataset and hyperparameters to find configurations that outperform the heuristic model according to the selected metrics.



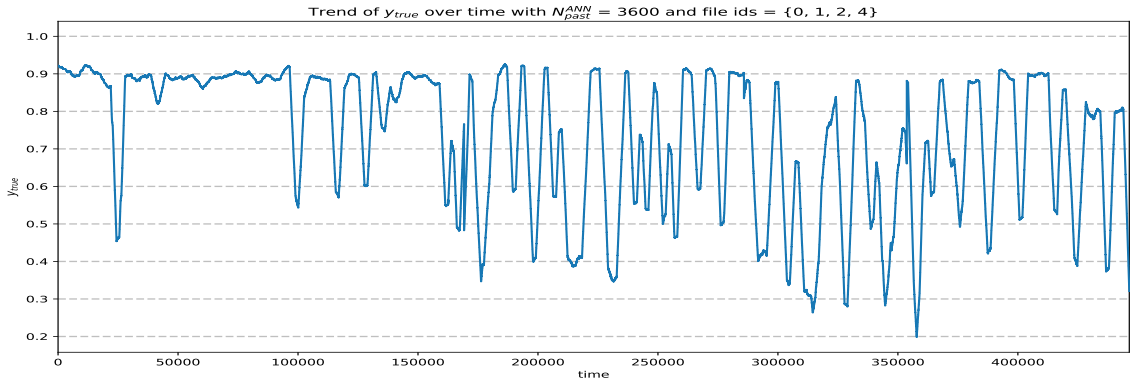
(a) Test list 4



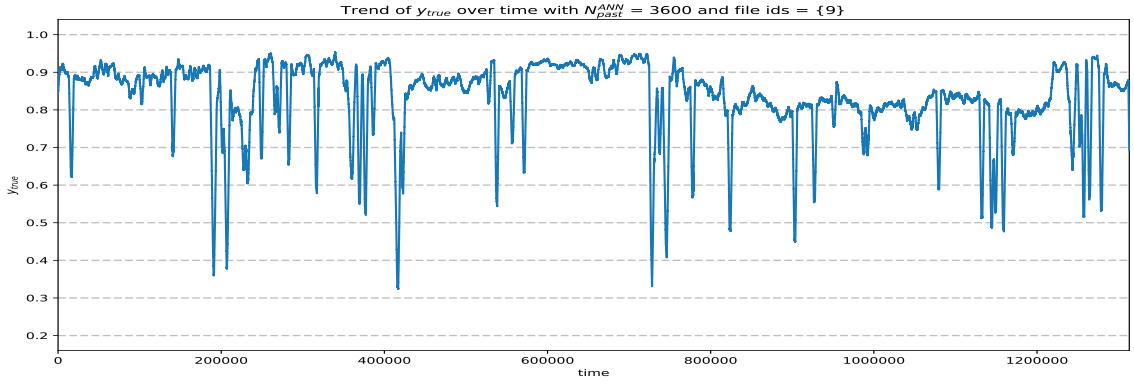
(b) Test list 7

Figure 6.2: y_{true} trends of the test lists associated with channel 5 (a) and 13 (b).

By construction, heuristic predictors are models that depend solely on the variable N_{past}^{AVG}. This parameter changes as the test or training datasets vary. To obtain a meaningful comparison, N_{past}^{AVG} must be set to the best possible value, i.e., the one that generates the best heuristic model for a particular test list. Best means finding a value of N_{past}^{AVG} that optimizes the performance indicators summarized in the previous section. For example, in the case of MSE, we want to find the value of N_{past}^{AVG} that realizes the smallest value of MSE on the test list. In reality, however, N_{past}^{AVG} is computed by considering the training dataset corresponding to that test list. We do this because we want to ensure a fair and meaningful comparison with the ANN models, given that the latter are trained solely using training data. In addition, both types of models, to have any practical usefulness, must be able to generate quality predictions on data for which the targets are unknown, which means finding the set of optimum parameters using only training data. For the computation of the best N_{past}^{AVG} relative to a test list, we can choose to use either the MSE or the MAE as performance metrics since the behavior of the other performance indicators depends, albeit indirectly, on the performance of the MSE or MAE. For example, assuming to keep fixed the MSE value for the ANN, a decrease or an increase of



(a) Test list 5

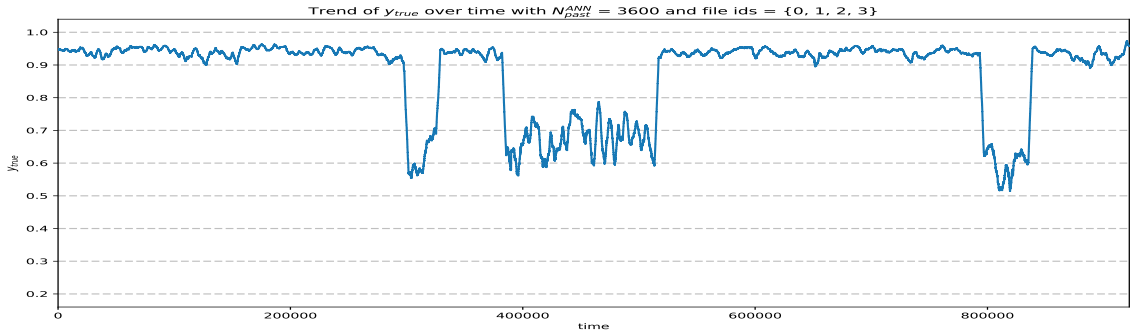


(b) Test list 6

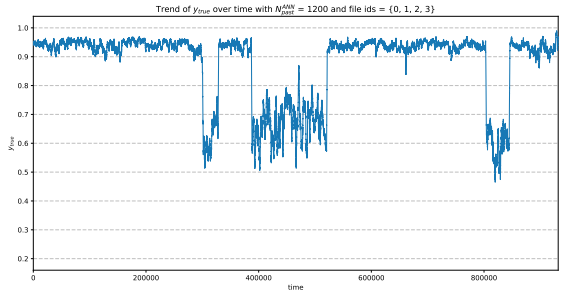
Figure 6.3: y_{true} trends of the test lists associated with channel 9

the MSE for the heuristic model calculated on a test list involves, respectively, to an improvement or worsening of the win-ratio in its favor. The same phenomenon happens similarly for the other three metrics. The choice between MSE or MAE is irrelevant for the final results of the experiments because analyzing the values obtained on all the test lists, calculated on the training dataset, the trend of the two metrics is nearly identical. We preferred anyway to use the MSE because it coincides with the objective function associated with all the configurations of the ANN networks. Moreover, when examining the experimental results obtained by applying Algorithm 2 on the training files, we noticed that the optimal value of N_{past}^{AVG} for the test lists referring to the same channel were almost identical, marking at most a difference in absolute terms of less than one percent. Therefore, to simplify our analysis and eliminate some of the variables involved, we decided to use a single value of N_{past}^{AVG} for all lists referring to the same channel, choosing the median value. We reported the optimal values of N_{past}^{AVG} for the different channels and N_{next} values in Table 6.2.

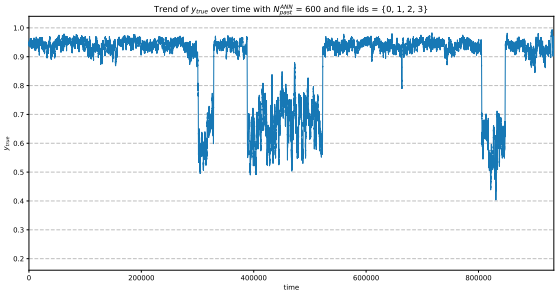
Specific to ANNs, the training dataset associated with an experiment is obtained using the raw files not contained inside the chosen test list. For example, if we are



(a) $N_{past} = 3600$



(b) $N_{past} = 1200$



(c) $N_{past} = 600$

Figure 6.4: y_{true} trends of the test list 1 with different values of N_{past}

channel/N_{next}	ch1	ch5	ch9	ch13
600	370	440	120	690
1200	440	480	130	750
3600	2450	13075	14400	3750

Table 6.2: Display of the optimal values of N_{past}^{AVG} as the channel and N_{next} vary

using the *list 1* as the test dataset, then the training dataset is extracted from all raw files belonging to that channel except the files with id 0, 1, 2, and 3 (see Table 6.3). All the experiments performed in this section use a neural network with the same structural configuration and partially share some of the other hyperparameters related to the training process. This model is the starting point for our analysis and is denoted henceforth as ANN_{base}.
Each experiment is divided into two recurring steps:

- training a model with the ANN_{base} configuration using a different training dataset each time that differs for the value of N_{past}^{ANN}
- compare the trained model with the heuristic model specific to the test list and analyze the results.

List	Training file IDs	# y_{true}^{3600}	# Days
1	$\forall i \in \{0,\ldots,19\} \setminus \{0,1,2,3\}$	2722371	16.07
2	$\forall i \in \{0,\ldots,19\} \setminus \{4,5\}$	2980141	17.60
3	$\forall i \in \{0,\ldots,19\} \setminus \{6,7,8,9,10\}$	2791631	16.45
4	$\forall i \in \{0,\ldots,19\} \setminus \{0,1,2,3\}$	2695045	15.91
5	$\forall i \in \{0,\ldots,10\} \setminus \{0,1,2,4\}$	3022366	17.61
6	$\forall i \in \{0,\ldots,10\} \setminus \{9\}$	2155682	12.66
7	$\forall i \in \{0,\ldots,10\} \setminus \{0,1,2,4,5\}$	2316061	13.51

Table 6.3: Lists of files used in training databases

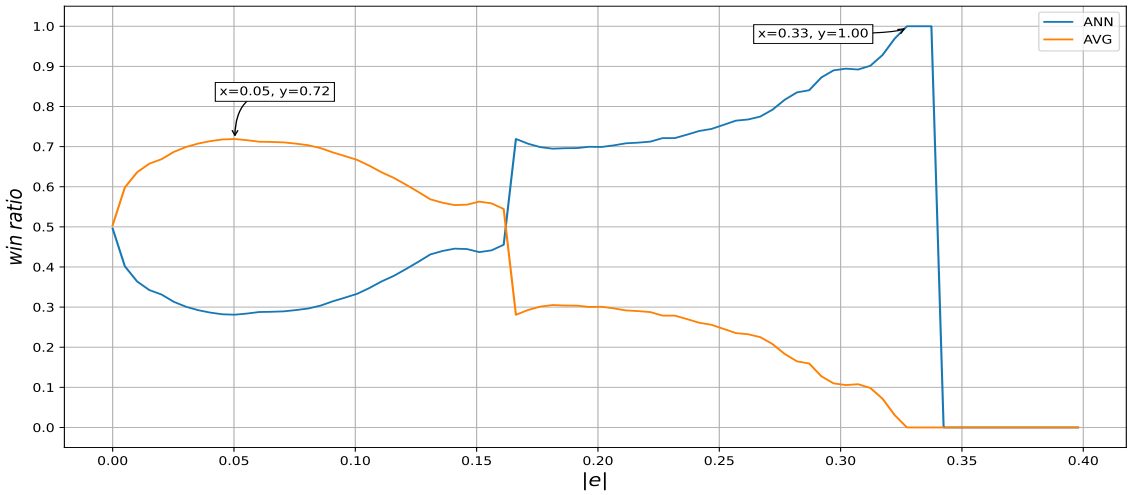
The characteristics of the ANN_{base} model are as follows: a FFNN consisting of 1 input layer of N_{input} neurons, 2 intermediate layers of 32 neurons each, and 1 output layer of 1 neuron. Each layer is densely connected with the next layer. Each neuron has ReLu as its activation function except the output neuron, which has a linear activation function. The weights are initialized with the *glorot normal initializer*, also called *Xavier normal initializer*. Each model is trained in 10 epochs with a batch size of 64, using Adam as an optimizer. The learning rate is set to an initial value of 0.01, which is halved at the end of each epoch by a scheduler (see Section 5.3.2). We used MSE as the objective function for all our configurations, as is customary for regression problems of this kind.
We ran our python programs in an environment known as *Google Colaboratory*, also known as *Colab*, which is a free Jupyter notebook environment that allows a user to run python code on the cloud. At the time of our experiments, we had at our disposal a machine with 2 Intel(R) Xeon(R) CPU processors with a clock speed of 2.20 GHz, 107.72 GB of disk space, and 12.69 GB of RAM.
Regarding training, for each combination of test list and N_{next} (600, 1200, or 3600), we trained 12 ANN_{base} models, each characterized by a different training dataset. The diversity consists in the values of N_{past}^{ANN}, starting from 1200 up to 14400 in increments of 1200, i.e., $\{i \cdot 1200 | i \in (1,\ldots,12)\}$, with a value of N_{step} fixed at 120. We present below the most significant part of the preliminary results obtained on the test lists with $N_{past} = 3600$.
For each test list, we present the measurements of two different models: the heuristic model with the optimal value of N_{past}^{AVG} and the ANN model that displayed the best MSE, i.e., with the smallest value. As depicted in the Table 6.4, the ANN models beat the corresponding heuristic models in all respects except for the ANN model associated with list 1. These last two models are very similar qualitatively, as can be seen from the win-ratio of 49.58%, which means, for example, that on average, 49 out of 100 times the predictions of the ANN model are more accurate than those of the heuristic model. Considering only absolute errors greater than 0.17, the win-ratio exceeds the 70% threshold to reach 100% wins for errors greater

N_{next}	Test list	Method	N_{past}	MSE $[\cdot 10^{-3}]$	MAE $[\cdot 10^{-2}]$	w [%]
3600	1	AVG	2450	2.18	2.32	–
		ANN	14400	2.27	2.53	49.58
	2	AVG	2450	4.05	4.57	–
		ANN	14400	3.14	4.14	55.60
	3	AVG	2450	1.76	2.18	–
		ANN	4800	1.27	1.95	56.03
	4	AVG	13075	1.14	2.21	–
		ANN	14400	0.82	1.83	56.17
	5	AVG	14400	27.22	12.86	–
		ANN	8400	24.77	11.02	57.51
	6	AVG	14400	9.25	5.89	–
		ANN	14400	6.55	4.86	54.39
	7	AVG	3750	1.92	3.09	–
		ANN	6000	1.48	2.81	55.37

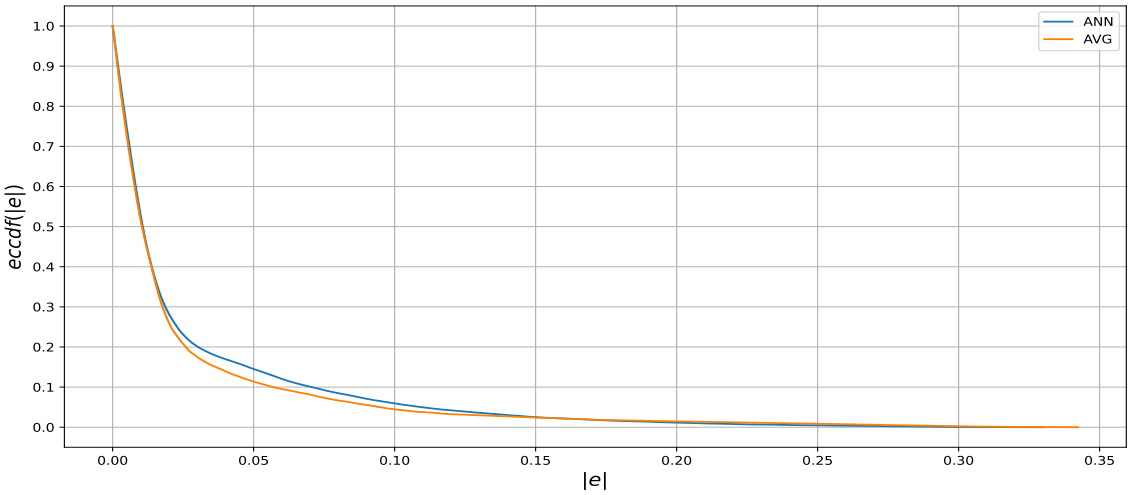
Table 6.4: Prelimary results of the AVG and ANN predictors on the test lists with $N_{past} = 3600$ using several KPIs

.

than 0.33 (see Fig. 6.5a). Recall that in the filtered win-ratio analysis, we only consider targets for which at least one of the two models committed an error greater than a given value x. Although the ANN model for list 1 is quantitatively inferior, the errors committed still fall within the range of values for which we consider the forecasts to be good, as supported by the MSE value of $2.53 \cdot 10^{-2}$. This means that in the optics of predicting the mean frame delivery probability of the next 30 minutes, the ANN model will commit on average an error of 2.53%. The best win-ratio was obtained on list 5 with 57.51%. Interestingly, it is also the list where the models realized the highest MSE and MAE values, looking at all experiments. Intuitively, the cause can be traced to the poor selection of the files that make up this list. Analyzing the results of list 6, which is associated with the same channel as list 5, we notice that the value of MAE is 2.26 times lower (from 11.02 to 4.86) with three times the size of the dataset, meaning that the source of the inaccurate predictions is due in part to the size of the test list. Secondly, as depicted in Fig. 6.3a, the target trend over time is very irregular, with frequent drops and sudden rises that touch highs of 0.9 and lows of 0.2. So it is understandable how both models struggle to generate accurate forecasts. Despite this, the methodology of using neural networks has proven to be a better solution than the heuristic technique.
We present, as the last case, the models in list 2 that achieved the median win-ratio. The eCCDF curve relative to the ANN model is always below that of the heuristic model for both small and large errors (see Fig. 6.6b), and this results in a win-ratio that tends to increase as larger errors are considered. These trends found in list 2 are also present for the predictors of the other lists that have not been analyzed.



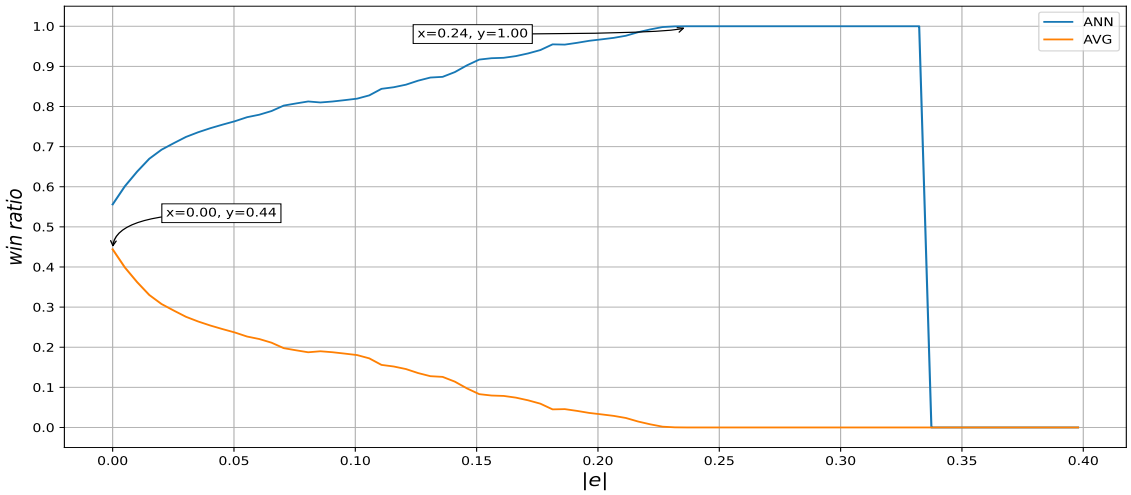
(a)



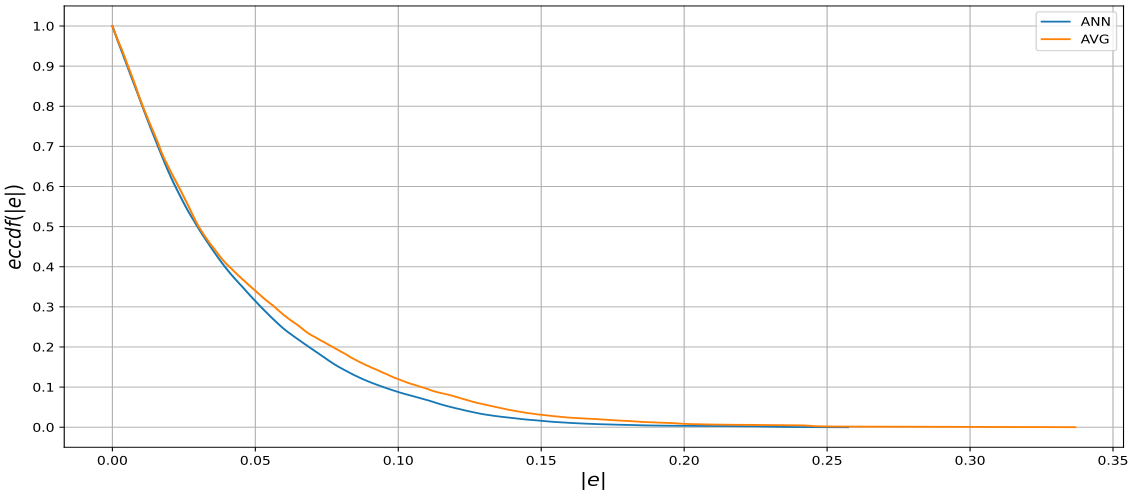
(b)

Figure 6.5: (a) Filtered win-ratio and (b) eCCDF of the absolute errors committed by ANN and AVG on list 1

By reducing N_{next} to 1200 (10 minutes), we see a sharp improvement in all KPIs analyzed across all models, particularly those referring to channel 9. As was previously established, the dynamics of this channel are very turbulent, making it difficult to create an accurate predictor for target windows as long as 30 minutes. By reducing this window to 10 and 5 minutes, we found significant improvements in the prediction accuracy for both methods. The values are reported in Table 6.5. We added to this table an additional row for each test list, which reports the measurements obtained by a model that was trained with the same best value of N_{past} obtained in $N_{next} = 3600$.
For $N_{next} = 600$ (5 minutes), the results are very similar to those anticipated in



(a)



(b)

Figure 6.6: (a) Filtered win-ratio and (b) eCCDF of the absolute errors committed by ANN and AVG on list 2

our analysis with $N_{next} = 1200$. The only noteworthy aspect is the improvements in MSE and MAE obtained on channel 9: we find a decrease of 37.37% and 26.03% on the targets of list 5, and 26.45% and 15.76% on list 6, respectively. As for the other lists, the measures obtained when varying N_{next} are stable, and no great improvements are appreciated. These metrics are difficult to improve without adding features of other kinds or different types of ANNs, in particular the values of MSE and MAE, which turn out to be already relatively small.
To recap, we have seen so far that ANN models are, in almost all cases, better predictors than the corresponding heuristic models, especially for analyzing future dynamics of medium duration (5 and 10 minutes) over unstable channels. However,

N_{next}	Test list	Method	N_{past}	*MSE* $[\cdot 10^{-3}]$	*MAE* $[\cdot 10^{-2}]$	w [%]
1200	1	AVG	440	1.36	2.20	–
		ANN	14400	1.41	2.22	54.77
		ANN	10800	1.34	2.17	54.88
	2	AVG	440	4.07	4.48	–
		ANN	14400	3.22	4.03	60.96
		ANN	3600	3.20	4.02	61.30
	3	AVG	440	1.11	1.96	–
		ANN	4800	0.93	1.85	55.94
		ANN	8400	0.94	1.84	56.57
	4	AVG	480	0.94	1.88	–
		ANN	14400	0.72	1.73	54.13
		ANN	7200	0.72	1.69	56.65
	5	AVG	130	19.96	7.66	–
		ANN	8400	16.26	7.59	52.34
		ANN	2400	16.67	7.22	57.80
	6	AVG	130	6.21	4.35	–
		ANN	14400	4.72	4.17	49.16
		ANN	4800	3.97	3.49	60.78
	7	AVG	750	1.84	3.13	–
		ANN	6000	1.40	2.77	58.25
		ANN	7200	1.39	2.76	58.66

Table 6.5: Prelimary results of the AVG and ANN predictors on the test lists with $N_{past} = 1200$ using several KPIs

we have also seen that the differences in the measurements are not substantial in some areas, especially in those whose values are already close to the best achievable result with the current methodology adopted. On the other hand, if we zoom our analysis on specific areas, we see that the performance of ANNs models is significantly better than their heuristic counterpart.

6.4 Results

This section presents and analyzes the results achieved by repeating the experiments reported in the previous section, using as a training algorithm the big data algorithm described in Section 5.4 applied to the complete database. In addition to the basic experiments, we have expanded our selection by adding for each list two new configurations of neural networks: on the one hand, we have increased the number of neurons present in the intermediate layers, from 32 to 64, on the other

N_{next}	Test list	Method	N_{past}	MSE $[\cdot 10^{-3}]$	MAE $[\cdot 10^{-2}]$	w [%]
600	1	AVG	370	1.19	2.17	–
		ANN	14400	1.15	2.13	54.53
		ANN	8400	1.12	2.10	55.19
	2	AVG	370	2.93	3.75	–
		ANN	14400	2.55	3.55	56.87
		ANN	3600	2.52	3.53	57.35
	3	AVG	370	0.89	1.88	–
		ANN	4800	0.77	1.79	55.17
		ANN	8400	0.78	1.79	55.42
	4	AVG	440	0.76	1.83	–
		ANN	14400	0.59	1.71	52.21
		ANN	8400	0.63	1.71	54.53
	5	AVG	120	11.86	5.66	–
		ANN	8400	9.77	5.77	50.41
		ANN	4800	10.44	5.34	59.39
	6	AVG	120	4.04	3.75	–
		ANN	14400	3.43	3.69	50.13
		ANN	2400	2.92	2.94	65.36
	7	AVG	690	1.59	2.94	–
		ANN	6000	1.32	2.74	55.43
		ANN	12000	1.31	2.73	55.69

Table 6.6: Prelimary results of the AVG and ANN predictors on the test lists with $N_{past} = 600$ using several KPIs.

hand, we have increased the value of the batch size. Therefore to each test list are associated four different neural networks configurations:

- configuration C_1 uses ANN_{base} together with the training algorithm presented in Section 5.3.2 applied to a reduced database, i.e., the results shown in the previous section
- configuration C_2 uses ANN_{base} together with the big data algorithm applied to the complete database
- configuration C_3 modifies ANN_{base}, increasing the batch size from 64 to 256
- configuration C_4 modifies ANN_{base}, increasing the number of neurons of each intermediate layer from 32 to 64.

Both C_3 and C_4 use as default training algorithms the big data algorithm employed to the whole database. Each of these two configurations is differentiated from C_2

List	Training file IDs	# y_{true}^{3600}	# Days	⇑ [%]
1	$\forall i \in \{0,\dots,54\} \setminus \{0,1,2,3\}$	7276936	43.13	268
2	$\forall i \in \{0,\dots,54\} \setminus \{4,5\}$	7534706	44.66	254
3	$\forall i \in \{0,\dots,54\} \setminus \{6,7,8,9,10\}$	7346196	43.51	264
4	$\forall i \in \{0,\dots,54\} \setminus \{0,1,2,3\}$	7172611	42.53	267
5	$\forall i \in \{0,\dots,22\} \setminus \{0,1,2,4\}$	6531560	38.17	217
6	$\forall i \in \{0,\dots,22\} \setminus \{9\}$	5664876	33.22	262
7	$\forall i \in \{0,\dots,22\} \setminus \{0,1,2,4,5\}$	5821235	34.04	252

Table 6.7: List of files used to train neural networks. ⇑ denotes the percentage of increase in dataset size with respect to the experiments presented in the previous section.

by a single hyperparameter so that we can then evaluate the impact that each of these parameters has on the final performance of the models. This means keeping the value of N_{past}^{ANN} fixed for all configurations relative to the same list, picking the value associated with the model found in C_1 with the smallest MSE, i.e., those shown in Table 6.4.
Regarding training, the datasets used are at least twice as large as the corresponding reduced datasets. The details of the enlarged set of training files are shown in Table 6.7.
Even though the models were trained with a larger database, there were no noticeable improvements as shown in Table 6.8. On the contrary, in some experiments, we reported, although marginal, some worsening of the KPIs. Specifically, in lists 1, 3, 5, and 6, passing from configuration C_1 to C_2, we have noticed a decrease on average of the win-ratio of 1.28%, touching a maximum of 2.17% reported in list 5. As for the MAE, the values remained roughly the same, reporting deterioration and improvement, in absolute terms, to the maximum of $0.07 \cdot 10^{-2}$ and $0.17 \cdot 10^{-2}$ reported in lists 6 and 1, respectively. Analyzing the results in greater depth, we discovered an interesting situation involving ANN models C_1 and C_2 in list 1. Switching from C_1 to C_2, although the average win-ratio dropped by 0.14%, from 49.58 to 49.45, the MAE improved by 6.72%, from $2.53 \cdot 10^{-2}$ to $2.36 \cdot 10^{-2}$. This implies that although the number of wins did not change, the predictions of the model C_2 have become more accurate. This fact is also confirmed by its eC-CDF, depicted in Fig. 6.7.
Although the overall win-ratio has decreased, the improvement in the MAE has benefited the filtered win-ratio in some areas where it did not previously perform as well. This gain becomes immediately evident by comparing the filtered win-ratio plots associated with C_1 and C_2, depicted in Fig. 6.8, especially in the initial areas. In the latter, the model reaches 70% of victories starting from 0.08, while in the former, the model exceeds this value only after it has passed 0.16.

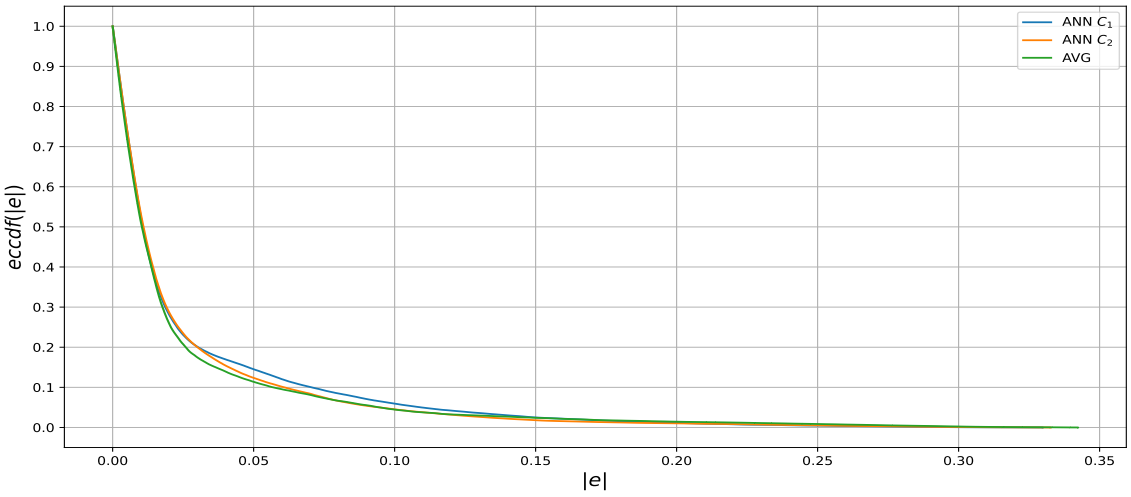


Figure 6.7: eCCDF of the absolute errors committed by ANN C_1, C_2 and AVG on list 1

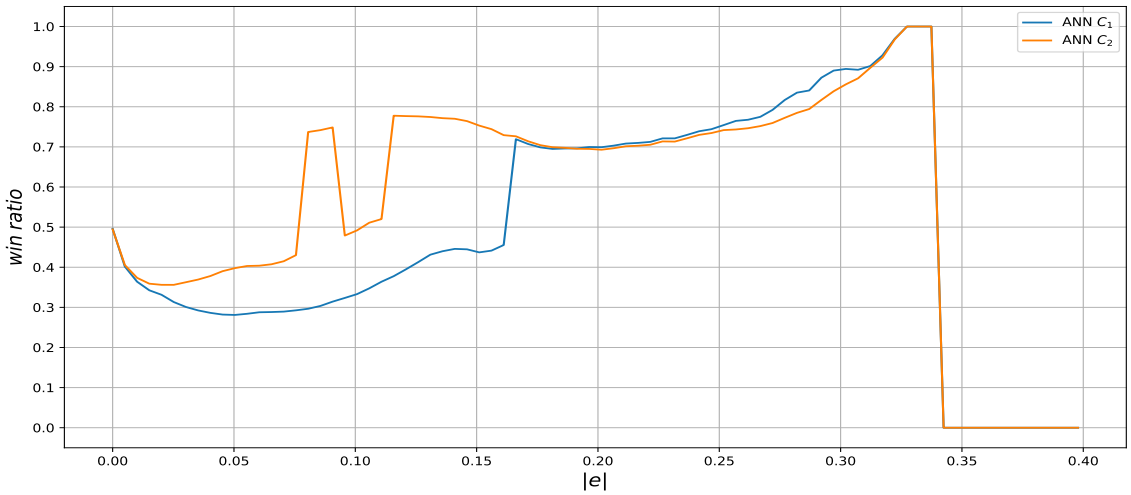
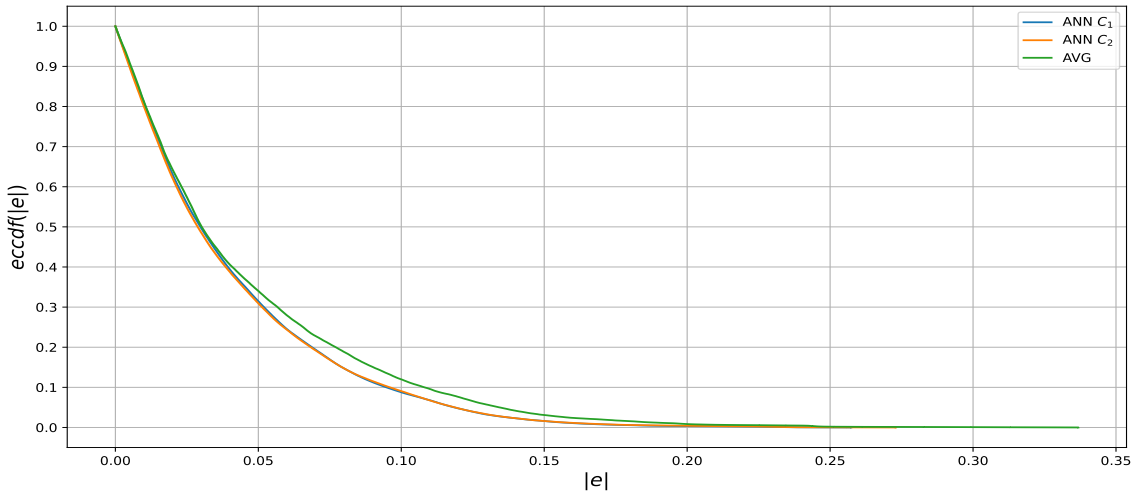
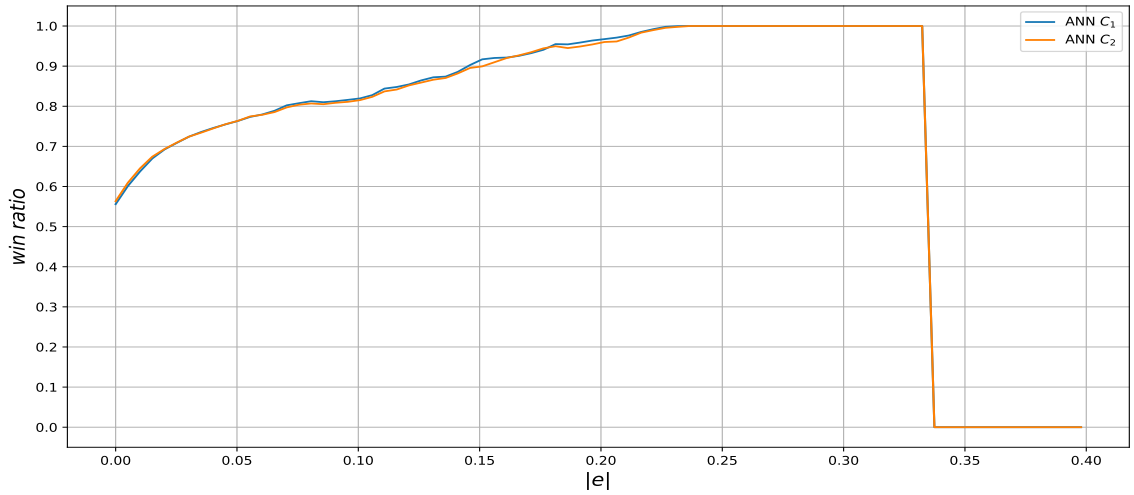


Figure 6.8: Win-ratio filtered by the absolute errors committed by ANN C_1 and C_2 on list 1

As for the experiments in which the win-ratio metric improved by switching from C_1 to C_2, i.e., lists 2, 4, and 7, the ANN model pairs remained nearly equal in terms of MSE and MAE, with maximum variations of 2.44% and 0.72%, respectively. It is natural that with such minimal variations, even the graphs of the eCCDFs and filtered win-ratios have remained quite similar, as shown, for example, in Fig. 6.9. The effects of increasing the batch size from 64 to 256, i.e., moving from C_2 to C_3, as expected did not benefit any KPIs of any test list, except list 6, but still in only marginal terms as seen in the other examples. C_3 is a valid alternative to C_2, despite the lower performances, in cases where we would like to prioritize the



(a)



(b)

Figure 6.9: (a) Filtered win-ratio and (b) eCCDF of the absolute errors committed by ANN C_1 and C_2 on list 2

training speed, which in our specific case, increased, on average, by 3.86 times. As for increasing the number of neurons in the intermediate layers, i.e., moving C_2 to C_4, the variations are imperceptible and unpredictable in both directions. For the above reasons and the fact that increasing the number of neurons involves an additional computational cost, which in turn lengthens the training time, the C_4 configuration is not recommended for our problem. The results of experiments with N_{past} values of 600 and 1200 are presented in Appendix A.
To recap, we have seen that training neural networks with an enlarged database, in our specific case, did not significantly improve the KPIs, especially regarding the overall win-ratio. Nevertheless, analyzing the eCCDFs for the experiments in which

a gain on the MAE was realized, as modest as 6.72%, we see that the magnitude of errors generated by the ANN predictors is visibly smaller than in the corresponding C_1 configuration. This gain is even more evident and pronounced when analyzing the win-ratio in some circumscribed areas of the errors. Regarding the effects of hyperparameters on the performance of a model, we saw that increasing the batch size or the number of neurons did not bring significant improvements to the analyzed metrics, except reducing or increasing the training time.

N_{next}	Test list	Config.	Method	N_{past}	MSE $[\cdot 10^{-3}]$	MAE $[\cdot 10^{-2}]$	w [%]
3600	1	–	AVG	2450	2.18	2.32	–
		C_1	ANN	14400	2.27	2.53	49.58
		C_2	ANN	14400	1.95	2.36	49.45
		C_3	ANN	14400	2.01	2.40	49.52
		C_4	ANN	14400	1.94	2.36	49.34
	2	–	AVG	2450	4.05	4.57	–
		C_1	ANN	14400	3.14	4.14	55.60
		C_2	ANN	14400	3.14	4.11	56.33
		C_3	ANN	14400	3.19	4.13	56.10
		C_4	ANN	14400	3.13	4.10	56.46
	3	–	AVG	2450	1.76	2.18	–
		C_1	ANN	4800	1.27	1.95	56.03
		C_2	ANN	4800	1.29	1.97	54.39
		C_3	ANN	4800	1.32	2.01	53.21
		C_4	ANN	4800	1.30	1.98	54.19
	4	–	AVG	13075	1.14	2.21	–
		C_1	ANN	14400	0.82	1.83	56.17
		C_2	ANN	14400	0.84	1.83	56.39
		C_3	ANN	14400	0.86	1.89	54.02
		C_4	ANN	14400	0.85	1.88	54.49
	5	–	AVG	14400	27.22	12.86	–
		C_1	ANN	8400	24.77	11.02	57.51
		C_2	ANN	8400	23.44	11.00	55.34
		C_3	ANN	8400	23.94	11.08	54.05
		C_4	ANN	8400	23.63	10.98	55.47
	6	–	AVG	14400	9.25	5.89	–
		C_1	ANN	14400	6.55	4.86	54.39
		C_2	ANN	14400	6.38	4.93	53.19
		C_3	ANN	14400	6.29	4.83	54.69
		C_4	ANN	14400	6.38	4.97	52.87
	7	–	AVG	3750	1.92	3.09	–
		C_1	ANN	6000	1.48	2.81	55.37
		C_2	ANN	6000	1.47	2.81	55.42
		C_3	ANN	6000	1.47	2.81	55,32
		C_4	ANN	6000	1.47	2.81	55,43

Table 6.8: Results of the AVG and different ANN predictors on the test lists with $N_{past} = 3600$ using several KPIs

.

Chapter 7

Conclusions

In this thesis, we exploited machine learning, specifically feed-forward neural networks, to predict the quality of a wireless channel over multiple time windows in terms of frame delivery ratio. This ability can be exploited in many different ways by both application processes and low-level hardware devices to improve reliability and efficiency of Wi-Fi channels. For example, we can leverage the predictions of the ANN model to adjust the transmission channel on the fly if the quality of the communication is expected to deteriorate in the immediate future. The methodology presented in this thesis does not have a practical use if used alone; instead, it is an integrative and supporting tool for those mechanisms that are designed to leverage this kind of information, such as, for example, adaptive algorithms installed in Wi-Fi adapters. The benchmark results achieved are reassuring, with ANN models outperforming conventional methods based on averages in almost all KPIs analyzed. However, in some experimental conditions, the results presented, in absolute terms, are lower than expected, denoting shortcomings and limitations of the adopted methodology. As confirmed in Section 6.4, despite how much one might try to change the configurations, the performances obtained in the different settings are very similar, proving to be difficult to improve with the current method. While working on this thesis, we published an article in a scientific journal with some of the results obtained from our experiments [17]. For future work, it is recommended to increase the number of distinct features, e.g., such as received signal strength indicator (RSSI) or transmission latency, and exploit a different type of neural networks that also considers the time factor, i.e., recurrent neural networks, to substantially improve the quality of the techniques seen here.

Appendix A

Tables

Channel	IDs	# y_{true}^{3600}	# y_{true}^{1200}	# y_{true}^{600}	# Days	Code name
1	0, 1, 2, 3	922053	931653	934053	5.42	list 1
1	4, 5	664283	669083	670283	3.88	list 2
1	6, 7, 8, 9, 10	852793	864793	867793	5.04	list 3
5	0, 1, 2, 3	911552	921152	923552	5.36	list 4
9	0, 1, 2, 4	446435	456035	458435	2.67	list 5
9	9	1313119	1315519	1316119	7.62	list 6
13	0, 1, 2, 4, 5	1152996	1164996	1167996	6.78	list 7

Table A.1: List of files used for the test databases

List	Training file IDs	# y_{true}^{3600}	# y_{true}^{1200}	# y_{true}^{600}	# Days
1	$\forall i \in \{0,\ldots,19\} \setminus \{0,1,2,3\}$	2722371	2758371	2767371	16.07
2	$\forall i \in \{0,\ldots,19\} \setminus \{4,5\}$	2980141	3020941	3031141	17.60
3	$\forall i \in \{0,\ldots,19\} \setminus \{6,7,8,9,10\}$	2791631	2825231	2833631	16.45
4	$\forall i \in \{0,\ldots,19\} \setminus \{0,1,2,3\}$	2695045	2731045	2740045	15.91
5	$\forall i \in \{0,\ldots,10\} \setminus \{0,1,2,4\}$	3022366	3036766	3040366	17.61
6	$\forall i \in \{0,\ldots,10\} \setminus \{9\}$	2155682	2177282	2182682	12.66
7	$\forall i \in \{0,\ldots,10\} \setminus \{0,1,2,4,5\}$	2316061	2328061	2331061	13.51

Table A.2: List of files used for the reduced training databases

List	Training file IDs	# y_{true}^{3600}	# y_{true}^{1200}	# y_{true}^{600}	# Days	$\Uparrow$ [%]
1	$\forall i \in \{0,\ldots,54\} \setminus \{0,1,2,3\}$	7276936	7394536	7423936	43.13	268
2	$\forall i \in \{0,\ldots,54\} \setminus \{4,5\}$	7534706	7657106	7687706	44.66	254
3	$\forall i \in \{0,\ldots,54\} \setminus \{6,7,8,9,10\}$	7346196	7461396	7490196	43.51	264
4	$\forall i \in \{0,\ldots,54\} \setminus \{0,1,2,3\}$	7172611	7290211	7319611	42.53	267
5	$\forall i \in \{0,\ldots,22\} \setminus \{0,1,2,4\}$	6531560	6574760	6585560	38.17	217
6	$\forall i \in \{0,\ldots,22\} \setminus \{9\}$	5664876	5715276	5727876	33.22	262
7	$\forall i \in \{0,\ldots,22\} \setminus \{0,1,2,4,5\}$	5821235	5862035	5872235	34.04	252

Table A.3: List of files used for the training databases. $\Uparrow$ denotes the percentage of increase in dataset size with respect to the reduced databases.

N_{next}	Test list	Config.	Method	N_{past}	MSE $[\cdot 10^{-3}]$	MAE $[\cdot 10^{-2}]$	w [%]
1200	1	–	AVG	440	1.36	2.20	–
		C_1	ANN	10800	1.34	2.17	54.88
		C_2	ANN	10800	1.23	2.06	57.40
		C_3	ANN	10800	1.18	1.98	59.69
		C_4	ANN	10800	1.22	2.07	56.75
	2	–	AVG	440	4.07	4.48	–
		C_1	ANN	3600	3.20	4.03	60.96
		C_2	ANN	3600	3.11	3.98	60.95
		C_3	ANN	3600	3.56	4.19	54.33
		C_4	ANN	3600	3.09	3.97	60.82
	3	–	AVG	440	1.11	1.96	–
		C_1	ANN	8400	0.94	1.84	56.57
		C_2	ANN	8400	0.93	1.82	57.43
		C_3	ANN	8400	0.92	1.82	57.31
		C_4	ANN	8400	0.93	1.82	57.37
	4	–	AVG	480	0.94	1.88	–
		C_1	ANN	7200	0.72	1.69	56.65
		C_2	ANN	7200	0.73	1.75	53,79
		C_3	ANN	7200	0.77	1.83	51.41
		C_4	ANN	7200	0.74	1.76	53,55
	5	–	AVG	130	19.96	7.66	–
		C_1	ANN	2400	16.67	7.22	57.80
		C_2	ANN	2400	15.52	7.14	54.90
		C_3	ANN	2400	15.47	7.03	56.18
		C_4	ANN	2400	15.61	7.10	56.61
	6	–	AVG	130	6.21	4.35	–
		C_1	ANN	4800	3.97	3.49	60.78
		C_2	ANN	4800	3.80	3.39	60.86
		C_3	ANN	4800	3.89	3.44	60.04
		C_4	ANN	4800	3.94	3.53	59.08
	7	–	AVG	750	1.84	3.13	–
		C_1	ANN	7200	1.39	2.76	58.66
		C_2	ANN	7200	1.41	2.77	58,83
		C_3	ANN	7200	1.42	2.77	59.10
		C_4	ANN	7200	1.41	2.77	59.14

Table A.4: Results of the AVG and different ANN predictors on the test lists with $N_{past} = 1200$ using several KPIs

.

N_{next}	Test list	Config.	Method	N_{past}	MSE $[\cdot 10^{-3}]$	MAE $[\cdot 10^{-2}]$	w [%]
600	1	–	AVG	370	1.19	2.17	–
		C_1	ANN	8400	1.12	2.10	55.19
		C_2	ANN	8400	1.06	2.04	56.48
		C_3	ANN	8400	1.06	2.04	56.32
		C_4	ANN	8400	1.06	2.04	56.54
	2	–	AVG	370	2.93	3.75	–
		C_1	ANN	3600	2.52	3.53	57.35
		C_2	ANN	3600	2.44	3.49	57.17
		C_3	ANN	3600	2.63	3.56	54.04
		C_4	ANN	3600	2.44	3.49	57.04
	3	–	AVG	370	0.89	1.88	–
		C_1	ANN	8400	0.78	1.79	55.42
		C_2	ANN	8400	0.78	1.77	55.99
		C_3	ANN	8400	0.77	1.77	56.00
		C_4	ANN	8400	0.78	1.77	55.83
	4	–	AVG	440	0.76	1.83	–
		C_1	ANN	8400	0.63	1.71	54.53
		C_2	ANN	8400	0.65	1.78	51.43
		C_3	ANN	8400	0.65	1.73	54.03
		C_4	ANN	8400	0.66	1.79	51.11
	5	–	AVG	120	11.86	5.66	–
		C_1	ANN	4800	10.44	5.34	59.39
		C_2	ANN	4800	9.31	4.98	59.34
		C_3	ANN	4800	9.23	5.01	59.15
		C_4	ANN	4800	9.40	5.04	58.35
	6	–	AVG	120	4.04	3.75	–
		C_1	ANN	2400	2.92	2.94	65.36
		C_2	ANN	2400	2.72	2.89	63.21
		C_3	ANN	2400	2.71	2.85	63.94
		C_4	ANN	2400	2.80	2.94	61.69
	7	–	AVG	690	1.59	2.94	–
		C_1	ANN	12000	1.31	2.73	55.69
		C_2	ANN	12000	1.33	2.74	55.99
		C_3	ANN	12000	1.33	2.74	55.99
		C_4	ANN	12000	1.33	2.74	56.01

Table A.5: Results of the AVG and different ANN predictors on the test lists with $N_{past} = 600$ using several KPIs

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," 2015.

[2] S. M. Grigorescu, B. Trasnea, T. T. Cocias, and G. Macesanu, "A Survey of Deep Learning Techniques for Autonomous Driving," *CoRR*, vol. abs/1910.07738, 2019.

[3] G. Campagna and R. Ramesh, "Deep Almond : A Deep Learning-based Virtual Assistant [Language-to-code synthesis of Trigger-Action programs using Seq 2 Seq Neural Networks]," 2017.

[4] S. Vaidya, P. Ambad, and S. Bhosle, "Industry 4.0 – A Glimpse," *Procedia Manufacturing*, vol. 20, pp. 233–238, 2018. 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA.

[5] T. P. Carvalho, F. A. A. M. N. Soares, R. Vita, R. da P. Francisco, J. P. Basto, and S. G. S. Alcalá, "A systematic literature review of machine learning methods applied to predictive maintenance," *Computers & Industrial Engineering*, vol. 137, p. 106024, 2019.

[6] M. Paolanti, L. Romeo, A. Felicetti, A. Mancini, E. Frontoni, and J. Loncarski, "Machine Learning approach for Predictive Maintenance in Industry 4.0," in *2018 14th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, pp. 1–6, 2018.

[7] A. Essien and C. Giannetti, "A Deep Learning Model for Smart Manufacturing Using Convolutional LSTM Neural Network Autoencoders," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 6069–6078, 2020.

[8] M. M. Rathore, S. A. Shah, D. Shukla, E. Bentafat, and S. Bakiras, "The role of ai, machine learning, and big data in digital twinning: A systematic literature review, challenges, and opportunities," *IEEE Access*, vol. 9, pp. 32030–32052, 2021.

[9] S. Scanzio, L. Wisniewski, and P. Gaj, "Heterogeneous and dependable networks in industry – A survey," *Computers in Industry*, vol. 125, p. 103388, 2021.

[10] N. Salles, N. Krommenacker, and V. Lecuire, "Performance Study of IEEE 802.15.4 for Industrial Maintenance Applications," in *IEEE International Conference on Industrial Technology, ICIT2008*, (Chengdu, China), p. CDROM, Apr. 2008. 6 pages.

[11] A. Aijaz, "High-performance industrial wireless: Achieving reliable and deterministic connectivity over ieee 802.11 wlans," *IEEE Open Journal of the Industrial Electronics Society*, vol. PP, pp. 1–1, 03 2020.

[12] Z. Li, M. A. Uusitalo, H. Shariatmadari, and B. Singh, "5G URLLC: Design Challenges and System Concepts," in *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pp. 1–6, Aug 2018.

[13] J. de Carvalho Silva, J. J. P. C. Rodrigues, A. M. Alberti, P. Solic, and A. L. L. Aquino, "LoRaWAN — A low power WAN protocol for Internet of Things: A review and opportunities," in *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*, pp. 1–6, 2017.

[14] E. Mackensen, M. Lai, and T. M. Wendt, "Bluetooth Low Energy (BLE) based wireless sensors," in *SENSORS, 2012 IEEE*, pp. 1–4, 2012.

[15] S. Scanzio, M. G. Vakili, G. Cena, C. G. Demartini, B. Montrucchio, A. Valenzano, and C. Zunino, "Wireless Sensor Networks and TSCH: A Compromise Between Reliability, Power Consumption, and Latency," *IEEE Access*, vol. 8, pp. 167042–167058, 2020.

[16] G. Cena, S. Scanzio, and A. Valenzano, "Improving Effectiveness of Seamless Redundancy in Real Industrial Wi-Fi Networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2095–2107, 2018.

[17] S. Scanzio, F. Xia, G. Cena, and A. Valenzano, "Predicting Wi-Fi link quality through artificial neural networks," *Internet Technology Letters*, vol. n/a, no. n/a, p. e326, 2021.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Commun. ACM*, vol. 60, p. 84–90, May 2017.

[19] L. Serrano, *What is machine learning?*, pp. 1–16. Manning, 2021.

[20] S. Hayou, A. Doucet, and J. Rousseau, "On the Impact of the Activation Function on Deep Neural Networks Training," 2019.

[21] S. Scanzio, P. Laface, L. Fissore, R. Gemello, and F. Mana, "On the use of a multilingual neural network front-end," pp. 2711–2714, 09 2008.

[22] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[23] D.-X. Zhou, "Universality of Deep Convolutional Neural Networks," 2018.

[24] Z. Wang, A. Albarghouthi, G. Prakriya, and S. Jha, "Interval Universal Approximation for Neural Networks," 2021.

[25] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation Functions: Comparison of trends in Practice and Research for Deep Learning," 2018.

[26] S. Scanzio, S. Cumani, R. Gemello, F. Mana, and P. Laface, "Parallel Implementation of Artificial Neural Network Training for Speech Recognition," *Pattern Recogn. Lett.*, vol. 31, p. 1302–1309, Aug. 2010.

[27] M. Mongelli and S. Scanzio, "A neural approach to synchronization in wireless networks with heterogeneous sources of noise," *Ad Hoc Networks*, vol. 49, pp. 1–16, 2016.

[28] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[29] X. Ying, "An Overview of Overfitting and its Solutions," *Journal of Physics: Conference Series*, vol. 1168, p. 022022, feb 2019.

[30] A. K. Gizzini, M. Chafii, A. Nimr, and G. Fettweis, "Deep Learning Based Channel Estimation Schemes for IEEE 802.11p Standard," *IEEE Access*, vol. 8, pp. 113751–113765, 2020.

[31] W. Jiang and H. D. Schotten, "Neural Network-Based Fading Channel Prediction: A Comprehensive Overview," *IEEE Access*, vol. 7, pp. 118112–118124, 2019.

[32] G. Cena, S. Scanzio, and A. Valenzano, "A software-defined MAC architecture for Wi-Fi operating in user space on conventional PCs," in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, pp. 1–10, 2017.

[33] G. Cena, S. Scanzio, and A. Valenzano, "SDMAC: A Software-Defined MAC for Wi-Fi to Ease Implementation of Soft Real-Time Applications," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3143–3154, 2019.

[34] F. Tramarin, S. Vitturi, and M. Luvisotto, "A Dynamic Rate Selection Algorithm for IEEE 802.11 Industrial Wireless LAN," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 2, pp. 846–855, 2017.

[35] D. Xia, J. Hart, and Q. Fu, "Evaluation of the Minstrel rate adaptation algorithm in IEEE 802.11g WLANs," in *2013 IEEE International Conference on Communications (ICC)*, pp. 2223–2228, 2013.