UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS VILELA SANCHES DE MAMANN

APRENDIZADO OFFLINE E ONLINE DE REDES NEURAIS NO CONTEXTO DE CASAS INTELIGENTES E DE COMPUTAÇÃO EM NÉVOA

CURITIBA

2023

LUCAS VILELA SANCHES DE MAMANN

APRENDIZADO OFFLINE E ONLINE DE REDES NEURAIS NO CONTEXTO DE CASAS INTELIGENTES E DE COMPUTAÇÃO EM NÉVOA

Offline and online neural network learning in the context of smart homes and fog computing

Dissertação apresentada como requisito para obtenção do título de Mestre em Ciências do Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Drª. Myriam Regattieri De Biase da Silva Delgado

Coorientador: Prof. Dr. Daniel Fernando Pigatto

CURITIBA

2023

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Curitiba

LUCAS VILELA SANCHES DE MAMANN

APRENDIZADO OFFLINE E ONLINE DE REDES NEURAIS NO CONTEXTO DE CASAS INTELIGENTES E DE COMPUTAÇÃO EM NÉVOA

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Ciências da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Engenharia De Computação.

Data de aprovação: 07 de Julho de 2023

Dra. Myriam Regattieri De Biase Da Silva Delgado, Doutorado - Universidade Tecnológica Federal do Paraná

Dra. Ana Cristina Barreiras Kochem Vendramin, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Joao Vitor De Carvalho Fontes, Doutorado - Universidade Federal de São Carlos (Ufscar)

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 07/07/2023.

AGRADECIMENTOS

Meus maiores agradecimentos vão aos meus orientadores Profª. Drª. Myriam e Prof. Dr. Daniel. Sem eles esta jornada certamente não teria nem mesmo se iniciado. Agradeço por terem aceitado me orientar durante este projeto, por terem sido pacientes, por terem me apoiado e incentivado e por terem contribuído de diversas formas durante todo o caminho. Não me esquecerei do tempo e esforço que ambos empregaram ao longo destes anos.

Agradeço também ao Maurício Copatti, por ter participado de discussões e experimentos, agregando conteúdo ao trabalho, especialmente na área de *Fog*.

RESUMO

À medida que aplicações de sistemas inteligentes baseados em Redes Neurais Artificiais (RNA), e em particular os modelos baseados em aprendizado profundo, se tornam altamente populares, surgem algumas desvantagens da implementação tradicional baseada em computação na nuvem. Questões como alto custo monetário para armazenar e executar aplicativos, baixa privacidade em dados e modelos, e alta latência que afeta estes modelos executados na nuvem podem dificultar seu uso, levando a experiências insatisfatórias por parte de seus usuários. A computação em névoa aparece então como uma possibilidade interessante. Este trabalho explora, no contexto de casas inteligentes, uma topologia de névoa como alternativa aos métodos de aprendizado *online* e modelos baseados em RNA executados *offline*. O trabalho propõe o uso de diferentes componentes rasos para formar modelos mais profundos e a utilização de um modelo recorrente profundo tradicional para tratar os dados de forma temporal. Nos experimentos envolvendo aprendizado *offline*, comparam-se seus desempenhos na resolução de oito problemas de classificação distintos. Os problemas dizem respeito às atividades realizadas por um morador, em cada um dos oito cômodos de uma casa inteligente usada como estudo de caso. Os resultados mostram que a hibridização de um modelo de *autoencoder* com classificadores baseados em redes neurais de múltiplas camadas é capaz de detectar atividades raras e fornecer bons resultados para quase todos os cômodos, especialmente quando abrangendo *pipelines* de dimensões adequadas. No entanto, vale mencionar que o modelo tradicional de múltiplas camadas é bastante competitivo. No contexto *online*, embora o desempenho do melhor modelo diminua, como esperado, algumas observações relevantes resultam dos experimentos, principalmente o fato de que a computação em névoa fornece resultados não muito distantes dos sistemas em nuvem, mas demandando menos recursos. A proposta *online* baseada em névoa surge, portanto, como uma alternativa para operação em ambientes com restrição de recursos computacionais ou tempo de processamento, como ocorre em dados *streaming*.

Palavras-chave: redes neurais artificiais; aprendizado online/offline; computação em névoa; problemas de classificação; atividades de usuário em casas inteligentes.

ABSTRACT

As smart applications based on Artificial Neural Networks (ANNs) become highly popular, particularly the models comprising deep learning, some drawbacks of traditional cloud-based deployment emerge. Issues like high monetary cost, for storing and running applications, low privacy on data and models, and high latency experienced by cloud-based neural networks might make their use difficult, leading to poor user experiences. Fog computing appears therefore as an interesting alternative. This work explores, in the context of smart homes, a fog topology as an alternative to online learning and ANN-based models running offline. The work proposes using different shallow components to form deeper models; it also adopts a traditional deep recursive approach to deal with temporal aspects of data. Experiments involving offline learning compare their performance on eight different classification problems which consist of activities performed by a user in each one out of eight rooms in the smart home addressed as the case study. Results show that the hybridization of an auto-encoder with classifiers based on multi-layer perceptrons can detect rare activities and provide good results for almost all rooms, particularly when encompassing suitable neural structure sizes in the pipelines. However, it is worth mentioning that the traditional multilayer model is quite competitive. In the online context, although the performance of the best approach decreases, as expected, some relevant insights result from experiments, especially that fog computing provides results not too far from cloud systems, yet demanding fewer resources. The proposal based on fog computing and online learning appears therefore as an alternative when dealing with streaming data on restricted environments in terms of computation resources or time.

Keywords: artificial neural networks; online/offline learning; fog computing; classification problems; smart home user activities.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND INITIALISMS

ANN	Artificial Neural Network
BPTT	Back-Propagation Through Time
CEC	Constant Error Carousel
IoT	Internet of Things
LSTM	Long Short-Term Memory
MLP	Multi-Layer Perceptron
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RTRL	Real-Time Recurrent Learning

SUMMARY

1 INTRODUCTION

The Internet of Things (IoT) is an environment of connected intelligent objects that interact using Internet communication protocols (POTRINO; RANGO; SANTAMARIA, 2019). Usually, such objects are embedded systems that consist of a highly-integrated hardware and software set that performs specific functions under computational constraints e.g. memory, processing, and energy supply (GRANJAL; MONTEIRO; SILVA, 2015). IoT has been the key to integrating these objects into new sets of applications, and it is often associated with Cloud Computing due to the inherent dependence on systems that demand Internet connectivity. Cloud computing is a model that enables ubiquitous, on-demand access to configurable sets of computing resources (e.g. networks, servers, storage, applications, and services). Cloud platforms are important for IoT applications providing powerful computing resources with a less expensive infrastructure (STERGIOU *et al.*, 2018).

As more smart applications become cloud-based and the number of constrained connected devices increases, the volume of data generated and the need for less costly solutions and with improved privacy increase as well. Cloud architectures are known for the large geographic distance between end devices and servers in the Cloud, which is often criticized regarding privacy issues. Although cloud services can potentially reduce infrastructure costs, they usually charge for processing time and data traffic, which may lead to high costs for applications that deal with high amounts of data. In many cases, this scenario makes applications that handle large amounts of data unfeasible.

A way to minimize centralized processing and cloud privacy problems is known as Fog Computing, as proposed by Cisco (BONOMI *et al.*, 2012). With solutions located closer to the network edge, Fog Computing makes room for a new profile of applications and services that can take advantage of geographically closer processing and improved data privacy. According to Bonomi *et al.* (2012), the main characteristics of Fog are low latency and location knowledge; widespread geographic distribution; mobility and heterogeneity; a large number of nodes; the predominant role of wireless access; and finally, a strong presence of streaming and real-time applications.

Artificial Neural Networks (ANNs) have been widely used in mobile devices and smart applications such as smart homes and cities, achieving promising results on various tasks. Current examples include object recognition (BANGARU *et al.*, 2021; LIAN *et al.*, 2022), computer vision, speech recognition, smart city (ALSAMHI *et al.*, 2021) and smart home (SKOCIR *et al.*, 2016; YU; ANTONIO; VILLALBA-MORA, 2022; VARDAKIS *et al.*, 2022) applications. ANN structures are composed of connected layers encompassing single processing units called neurons. Input data are processed by each layer until the last one outputs the calculated result (HAYKIN; NETWORK, 2004).

As more layers and neurons are used, more computational resources are demanded as well. A common solution for that is to deploy such networks in cloud computing services with

nearly unlimited computing resources. As previously commented, although highly available and scalable, cloud servers may experience issues such as high latency, and low privacy (YI *et al.*, 2015), which can negatively impact task requirements. Then, Fog Computing is an alternative to deal with online learning in the context of neural networks.

An online neural learning method is one capable of processing streaming data, piece-by-piece in a serial fashion, without having the entire input available from the start. In contrast, an offline neural learning method receives the whole problem data from the beginning and is required to output an answer which solves the problem at hand. Online and offline neural models have their own advantages and disadvantages (PUTTIGE; ANAVATTI, 2007). Offline models can handle large datasets, as computation time and memory are not critical to their functioning. They are robust to small variations but fail to adapt to larger changes in the system. Online models can adapt quickly to variations in the non-linear behavior of inputs but might be less accurate because of small sets of training data given as batches. Moreover, due to their usual low memory capability, forgetting large amounts of past data is frequent in the learning of online models.

By exploring offline ANN learning methods, the present work provides neural network models to classify activities in the context of smart homes. The proposed models include those based on shallow components, which are segments of the network with less hidden layers (not enough layers to be considered a deep model), those using autoencoders, which are models specialized in detecting outlier data, and those encompassing a Long Short-Term Memory (LSTM) (HOCHREITER; SCHMIDHUBER, 1997), a traditional recurrent deep model for time series. Aiming to mitigate cloud service issues, the work also proposes an alternative solution for online ANN learning which is based on fog computing. In the experiments, eight different classification problems are addressed; each one is described as the problem of classifying the activities that occur in a particular room of a smart home, based on the information provided by sensors distributed in the room. The results of proposed approaches running in offline mode are compared among each other. The best-proposed model for the most difficult room is selected to run in an online mode and the results under different setups are analyzed. Although other works have already explored the subject (SKOCIR *et al.*, 2016), the investigation of modular ANN online and offline learning in the context of smart homes and fog/cloud computing is, to the best of our knowledge, an unexplored field.

1.1 Objectives and Research Questions

This section outlines the research objectives, including the specific ones regarding two different contexts: offline and online learning. The section also presents the research questions or hypotheses being investigated.

1.1.1 General Objective

The work aims to evaluate artificial neural network-based models trained offline in the Cloud and online in the Fog to classify activities in each room of the addressed smart home.

1.1.2 Specific objectives

The specific objectives regarding the offline learning mode are:

- To evaluate the result of hybridizing autoencoder and multi-layer perceptron models;
- To compare the performance of different models (hybrid and non-hybrid) on solving eight addressed classification problems;
- To evaluate the influence of the number of classification levels on the models' performance;
- To test different architecture complexities in the proposed models;
- To compare the performance of a model based on LSTM with the ones based only on shallow components.

And the specific objectives regarding the online learning mode are:

- To compare the performance of online learning based on fog with online learning based on cloud computing;
- To explore different setup configurations for online learning and fog/cloud computing resources of one particular model (deepest/largest model considered the best one) running in a specific room (classified as the most difficult one);

1.1.3 Research Questions

1. Can a shallow standalone model solve the addressed problems?
2. Can two classification levels improve the solution provided by a unique level?
3. Can the hybridization of shallow models outperform the non-hybrid ones?
4. How is the performance of online learning based on fog computing compared with the one obtained through cloud computing?
5. What is a good trade-off between transmission rate and memory/processing capacity to run the largest model in online mode?

1.2 Main Contributions

Aiming to achieve the described objectives and answer the raised questions, the present work contributes by providing different models trained offline to solve the addressed smart home problems (classify activities in each room); and then by exploring different configurations of online learning and fog/cloud computing resources for one particular model running in a specific room. The work also contributes: a) by presenting shallow components that can be get together to provide a deeper model capable of solving the addressed smart home problem; b) by comparing pure Multi-Layer Perceptron (MLP) models and hybrid ones encompassing autoencoder and MLP in the context of offline learning; c) by expanding the area of application of fog computing to the context of online learning in smart homes.

The following publication (MAMANN; PIGATTO; DELGADO, 2022) in a relevant Brazilian conference (Qualis A4[1]) is a direct result of this dissertation:

- MAMANN, L. V. S. ; PIGATTO, D. F. ; DELGADO, M. R., . Offline and Online Neural Network Learning in the Context of Smart Homes and Fog Computing. In: Brazilian Conference on Intelligent Systems, 2022. Proceedings of BRACIS, 2022. v. 1. p. 357-372.

As an indirect contribution, the work resulted in the following publication (MAMANN *et al.*, 2021) in another relevant Brazilian conference (Qualis A4):

- MAMANN, L. V. S. ; SIMAO, J. M. ; DELGADO, M. R. ; PIGATTO, D. F. Paradigma Orientado a Notificações Aplicado à Programação de Microcontroladores. In: XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais, 2021, online. 2021: Anais Estendidos do XI SBESC, 2021. v. 1. p. 1-6.

1.3 Organization

The text structure is divided into five chapters. After this introduction, Chapter 2 presents the fundamentals of fog computing, artificial neural networks (shallow and deep models, recurrent and non-recurrent ones), and online *versus* offline learning. Chapter 3 describes the proposed approaches. Chapter 4 presents and discusses the obtained results. Finally, Chapter 5 concludes the work and discusses directions for future research.

[1] From https://www.gov.br/capes/pt-br/centrais-de-conteudo/documentos/avaliacao/09012022_RELATORIOQUALISEVENTOS20172020COMPUTACAO.PDF

2 BACKGROUND

This section outlines the basic concepts concerning fog computing and artificial neural networks, including their training aspects, and provides a foundation for understanding the significance and contributions of the proposed approaches and the results.

2.1 Fog Computing

The popularity of IoT has increased over the years, which has led to an unprecedented amount of data being generated every second, due to the number of devices connected to the Internet. According to CISCO (2015), billions of devices generate data that need to be processed somehow to make IoT-based solutions work. However, as the number of devices and volume of data scale, people start to face challenges that need to be tackled.

One of the most common solutions is to integrate IoT with cloud services, which is justified by some characteristics of these services that are beneficial to IoT, such as scalability and high availability of resources, nearly unlimited processing capabilities, storage capabilities, and others (DHANARAJ *et al.*, 2021). Yet, there are many aspects of the Cloud that make it not ideal for all cases. For instance, there are problems such as ensuring the security of data for data-sensitive applications, ensuring the latency is low for time-sensitive processes, and connecting IoT devices to cloud services, either because vendors often do not follow the same standards and protocols (DHANARAJ *et al.*, 2021) or because it is costly to provide Internet access to where the device is placed (OPENFOG, 2017). That does not mean the Cloud should be replaced by another service, but that it could benefit from other structures or architectures. This is where the fog comes into play.

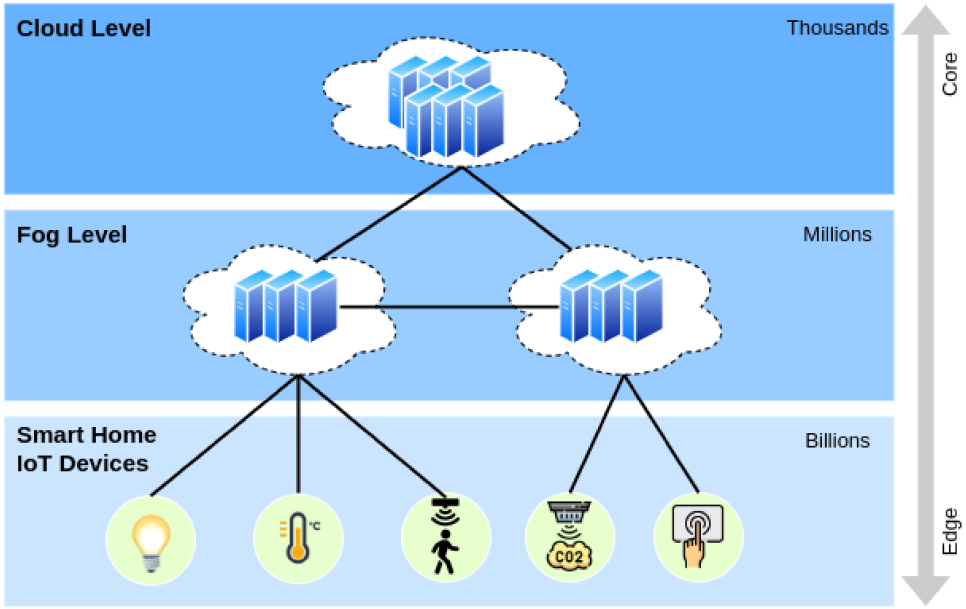
Fog is an extension of the Cloud. It is an intermediary level of the network that brings cloud resources closer to the devices that generate data and to the devices that use these data (CISCO, 2015; MATT, 2018). In order to extend the Cloud, fog systems use nodes that could be any type of device that can provide some level of processing and storage capabilities and that is connected to the network. These nodes are usually placed somewhere geographically closer to the IoT devices, which could be any place, even remote locations with limited internet access such as offshore oil platforms (CISCO, 2015), oil pipelines, or roads (OPENFOG, 2017). The task of fog nodes is to process and filter data, especially what is time sensitive, and later send to the cloud only what needs big data analysis (MATT, 2018).

There are a few points that can be improved when adding a fog layer to the project structure. First of them is latency, as the fog nodes are closer to the end devices and are often connected to managed local networks, thus communicating faster with IoT devices and even a small improvement in latency might be crucial depending on the application. Another key point is security, as many applications deal with business-critical data that cannot be exposed to the

public Internet. With the fog running on a local network, it is easier to apply policies that ensure the data are properly encrypted and secured and are not being intercepted by attackers (CISCO, 2015; OPENFOG, 2017). The performance and reliability of the network can also be improved, since the fog can do some level of processing and can filter data that need to be sent to the Cloud, thus the system can keep a large amount of data locally avoiding consuming all bandwidth unnecessarily (CISCO, 2015). There are still other benefits that might come with the use of fog, some of them are more application-specific such as minimization of costs on the network structure or on the size of the cloud structure (OPENFOG, 2017).

As mentioned before, the fog is not a replacement for cloud services. It can act as a middleware between IoT devices and the Cloud. As such, one of the main responsibilities of fog nodes is to decide what can be processed locally and what should be sent to the Cloud. What drives this decision-making process are the characteristics of the system and data. Usually, time-sensitive processes benefit from fog capabilities, because they need a fast analysis and response. Having a lower latency, the fog can make a big difference in such cases. However, the Cloud has a much larger amount of resources and storage space, which makes it more appropriate for cases when larger volumes of data need to be stored or when bigger computing power is needed, for instance when analyzing data (CISCO, 2015; MATT, 2018).

Figure 1 – A taxonomy of Cloud, Fog and Smart Home levels



Source: Author

2.2 Artificial Neural Networks

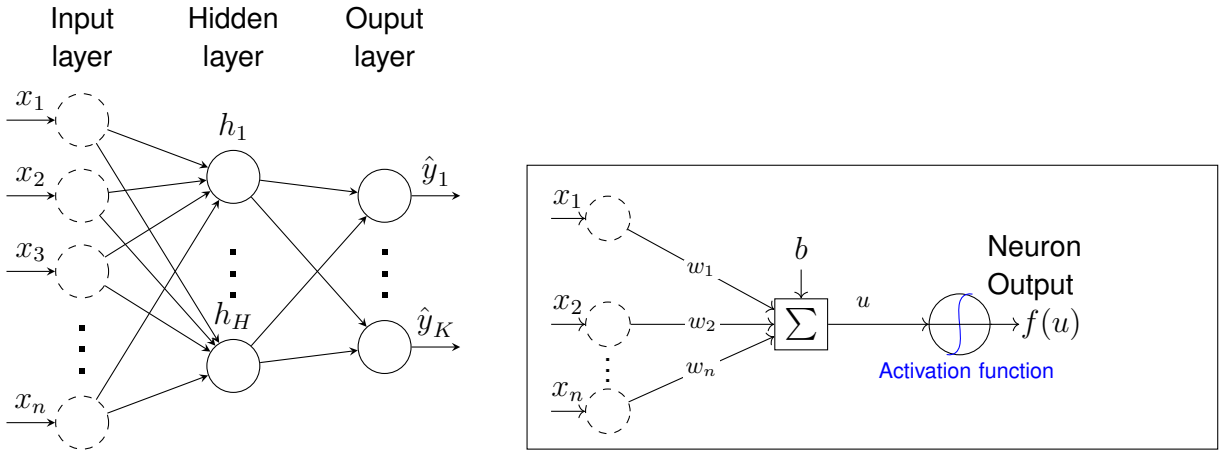
The ability to learn representations has made neural networks (HAYKIN; NETWORK, 2004) one of the most popular among machine learning methods for function approximation and regression (FERNÁNDEZ-DELGADO *et al.*, 2019) but mainly for classification (ZHANG, 2000).

Classification tasks, in particular, have been leveraged by the emergence of deep models (SCHMIDHUBER, 2015; ALZUBAIDI *et al.*, 2021). This section provides neural network concepts that are necessary to understand the basic components the proposed models are composed of, especially Multi-Layer Perceptrons, Autoencoders, and recurrent networks, in particular the Long Short-Term Memory model.

2.2.1 Multi-Layer Perceptrons

In 1986, Frank Rosenblatt proposed the MLP (RUMELHART; HINTON; WILLIAMS, 1986), a model that, after a long period of discredit, reintroduced Artificial Neural Networks as a hot topic in Artificial Intelligence research. Used in a standalone way or as a final component of deep models, MLP is still useful in many applications. In a typical configuration, the neurons of such a model are arranged as a feedforward layered structure, as shown in Figure 2.

Figure 2 – An overview of an ANN shallow model



(a) multi-layer perceptron with one hidden layer.

(b) detailed functioning of a neuron in the hidden layer.

Source: Adapted from https://tikz.net/neural_networks/

Figure 2 (a) shows an MLP with one hidden layer, capable of dealing with non-linear separability in data. MLP models represent an improvement over single-layer networks like perceptrons (ROSENBLATT, 1958), i.e., those models with no hidden layer, which can only handle linearly separable problems (MINSKY; PAPERT, 1969; MINSKY; PAPERT, 2017).

In a more detailed view of the neuron functioning depicted in Figure 2 (b), we have a node with a bias b; n inputs[1] $x_1, x_2, \ldots, x_n$, associated with weights $w_1, w_2, \ldots, w_n$, that emulate the role of the synapse by reinforcing or decreasing the importance of the respective neuron input to the neuron output; and an *activation potential* u that is computed based on a dot product. The

[1] The dashed circles represent identity functions in which the inputs are available to be directly used by the neural networks.

activation potential is calculated as described by Equation 1.

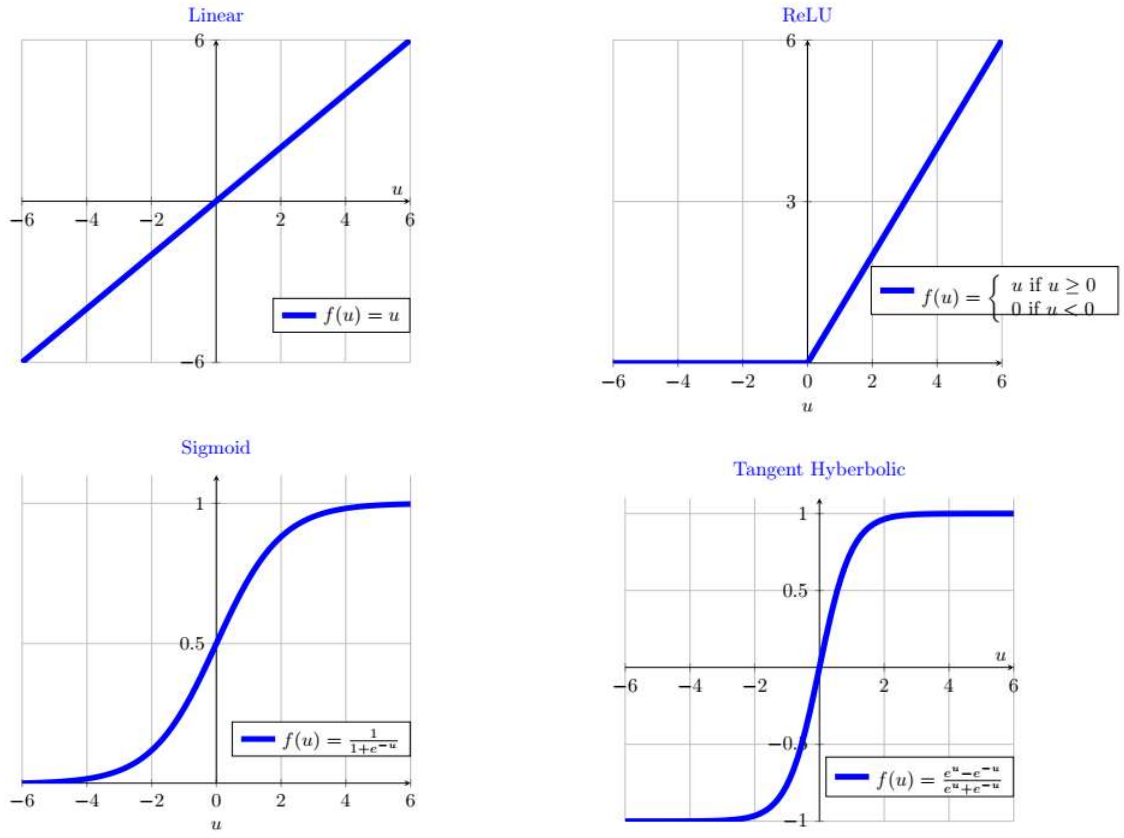
$$u = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

The neuron output is computed as a function of the *activation potential* u (from Equation 1), as given by Equation 2.

$$f(u) = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{2}$$

where f is the *activation function* that except for specific choices (e.g. Linear), always poses nonlinear characteristics in the neuron's behavior (GRAVES, 2012). Figure 3 shows the most usual activation function found in the literature.

Figure 3 – The most usual activation functions



(a) Top left - Linear ($f(u) \in [-\infty, +\infty]$); (b) Top right - ReLU ($f(u) \in [0, +\infty]$); (c) Bottom left - Sigmoid ($f(u) \in [0, +1]$); (d) Bottom right - Tangent hyperbolic ($f(u) \in [-1, +1]$).
Source: Author

A linear activation function can be used in neurons at the output level of the neural network structure to provide linear combinations of non-linear functions f computed in the pre-

vious layers ($\hat{y}_k = u_k = \sum_{j=1}^{H} w_{jk} f(u_j)$). Besides that, there are several types of activation functions such as sigmoid, hyperbolic tangent, and Rectified Linear Unit (ReLU), the latter one widely used in domains involving images. The choice of a particular *activation function* is usually restricted by its derivative properties, which allow or not its use in most traditional training algorithms.

In Section 3.1 we show MLP-based components and MLP standalone models used in the proposed approaches. In Section 4.2, we describe the main parameters used in the experiments for those models.

2.2.2 Autoencoders

Firstly used as unsupervised pre-training of ANNs in (BALLARD, 1987), an autoencoder is a neural network that when properly modeled as well as properly pre-processed can extract information from obvious inputs (HAWKINS *et al.*, 2002; DONG *et al.*, 2018; ZHAI *et al.*, 2018). Figure 4 shows an autoencoder that turns a high-dimensional input into a latent low-dimensional representation (encoder), and then performs a reconstruction of the input with this latent code (the decoder).

Although using multiple hidden layers can boost the performance by extracting slightly higher level features (COATES; NG; LEE, 2011), a model with a single hidden layer enables extracting important features and can be used as an outlier detection in many applications (HAWKINS *et al.*, 2002).

As illustrated in Figure 4 (a) and (b), in the simplest case, the encoder receives the input $\mathbf{x} \in \mathbb{R}^n$ and maps it through the *activation function* f^e, to the hidden layer. Applying Equation 2 to this layer we get its output, given by Equation 3.

$$y_j = f^e(\mathbf{w}_j^e.\mathbf{x} + b_j^e), \quad j = 1,...,H \tag{3}$$

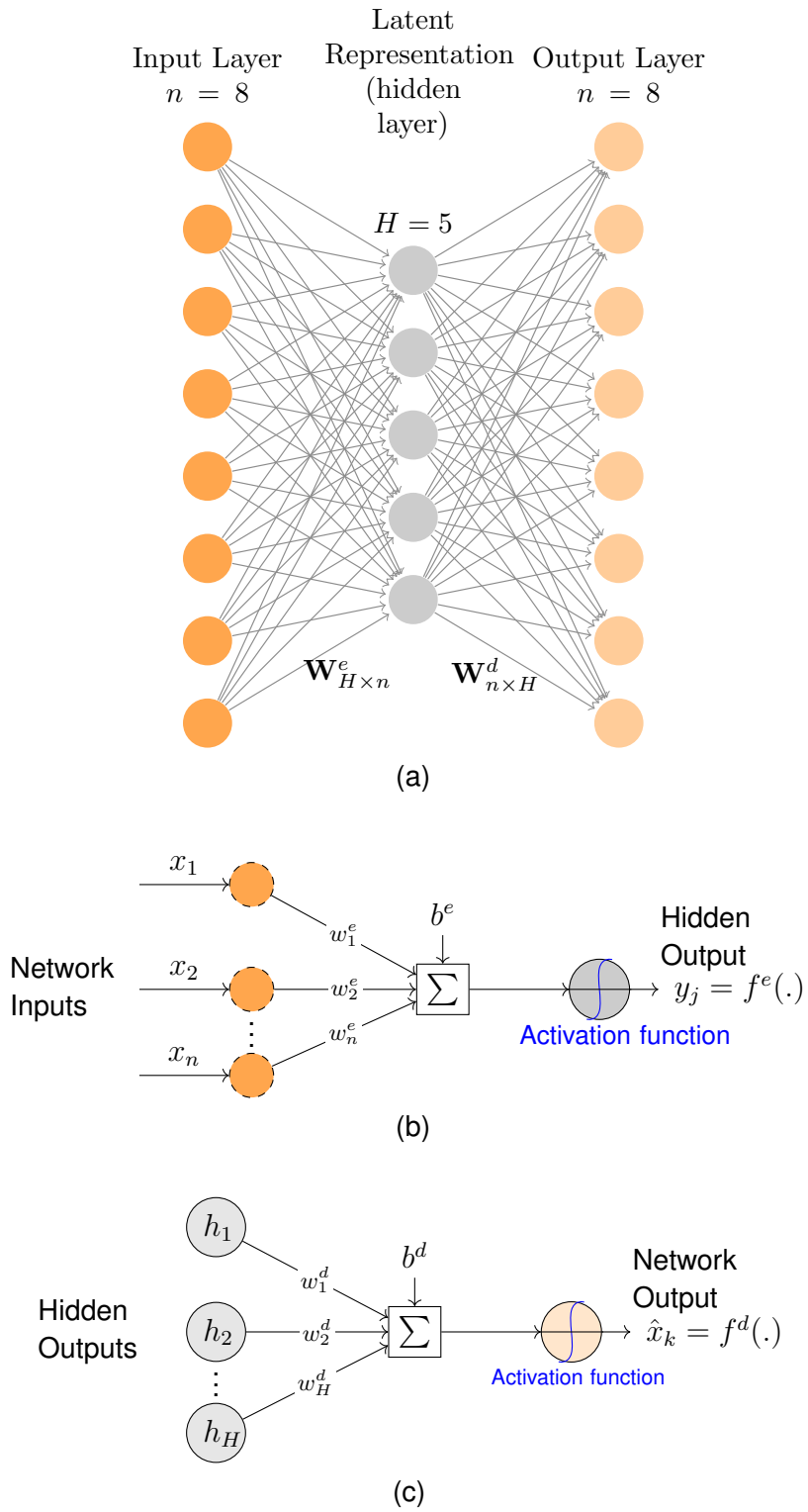
where $\mathbf{w}_j^e$ is the vector representing encoder weights of the j^{th} neuron out of the H neurons in the hidden layer and b_j^e is its bias.

After that (see Figure 4 (a) and (c)), the decoder stage maps the hidden layer to the reconstruction $\hat{\mathbf{x}} \in \mathbb{R}^n$. To get the reconstructed output of the network, we apply Equation 2 to the output layer, resulting in Equation 4.

$$\hat{x}_k = f^d(\mathbf{w}_k^d.\mathbf{y}^{\text{hidden}} + b_k^d), \quad k = 1,...,n \tag{4}$$

where $\mathbf{w}_k^d$ is the vector representing decoder weights of the k^{th} neuron out of the n neurons in the output layer and b_k^d is its bias. Notice that f^d, $\mathbf{w}_k^d$ and $b^d{}_k$ for the decoder may be unrelated to their encoder counterparts.

Figure 4 – An overview of a simple autoencoder



(a) an example of a model with n inputs/outputs neurons and a single hidden layer (latent representation) with H neurons, $\mathbf{W}^e$ and $\mathbf{W}^d$ are the matrices of encoder and decoder weights, respectively; (b) detailed functioning of a neuron in the encoder layer; (c) detailed functioning of a neuron in the decoder layer.

Source: Adapted from https://tikz.net/neural_networks/

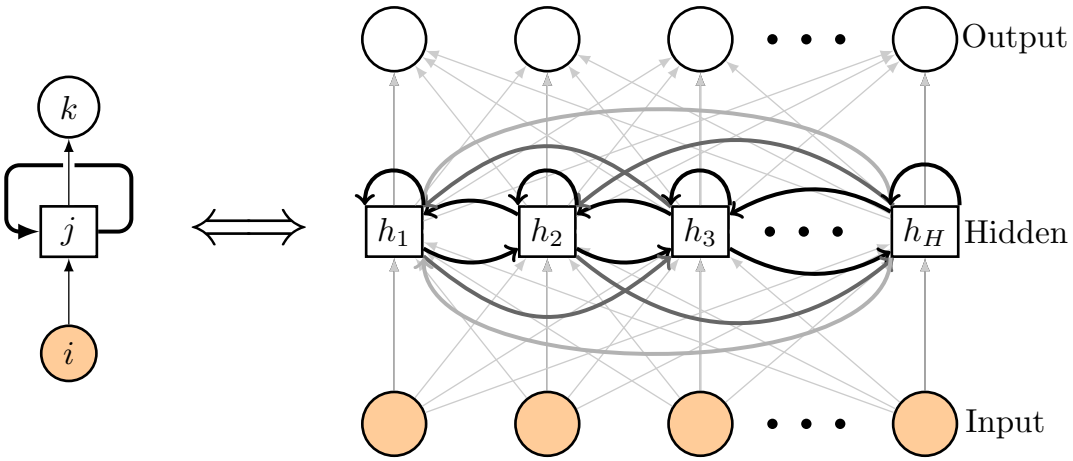
Although there are several variants taken from the model depicted in Figure 4 (DONG *et al.*, 2018; ZHAI *et al.*, 2018), in the present work we consider only the basic model, in an attempt to identify novelty or outliers in a simple way. In the performed experiments, outliers represent rare activities occurring in each room of a smart residence.

2.2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are special types of networks in which a neuron can connect to the next, previous, or its own level, forming cyclic connections. While a non-recurrent network can only map from input to output vectors, an RNN can in principle map from the entire history of previous inputs to each output (GRAVES, 2012). The key point is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state and thereby influence the network output.

Figure 5 shows a neural network with an external input layer, a recurrent hidden layer, and an external output layer.

Figure 5 – A recurrent neural network with H neurons in the hidden layer (right), where i, j, and k (left) represent neurons in the input, hidden and output layers, respectively.



Source: Author.

As depicted in Figure 5, the cyclic connections make the previous outputs also network inputs. Consequently, the network takes information from previous states to construct the current state. This is a characteristic of a system with memory. According to Hochreiter and Schmidhuber (1997), the recurrent connections can be interpreted as short-term memory, as their weights change more quickly. This characteristic is crucial for time series problems, such as the one discussed in this work, where not only the last data but also all historical data are relevant.

The input u received by an output unit k at time t can be calculated from the hidden activations similarly to the feedforward networks (see Equations 1 and 2) as given by Equation 5.

$$u_k^t = \sum_{j=1}^{H} w_{jk} y_j^t \tag{5}$$

Both the input and output of a neuron j at a recurrent hidden layer can be calculated similarly to the non-recurrent layers. The only difference is the fact that the input of a recurrent layer presents an extra factor, relative to the influence of the previous states, which is the recurrent connection. The input and output are represented by equations (6) and (7).

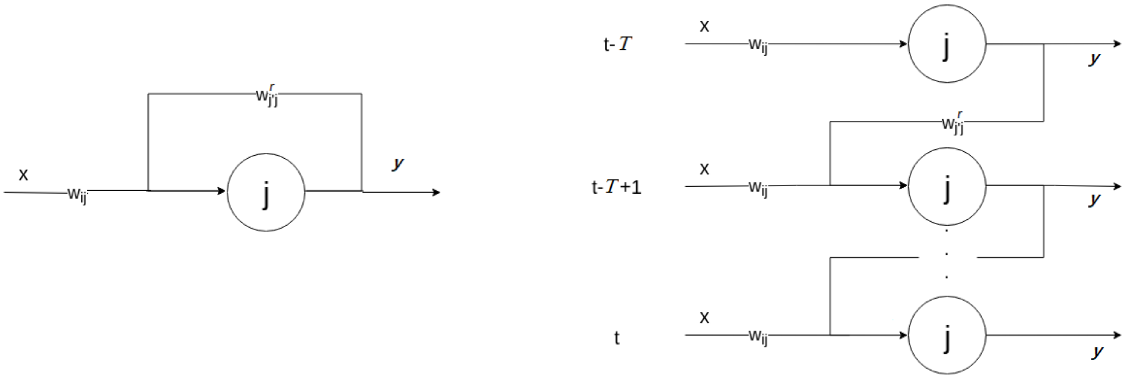
$$u_j^t = \sum_{i=1}^{n} w_{ij} x_i^t + \sum_{j'=1}^{H} w_{j'j}^r y_{j'}^{t-1} \tag{6}$$

$$y_j^t = f\left(u_j^t\right) \tag{7}$$

where u is the hidden unit input, the index t indicates an instant of time and j a neuron, y is the hidden unit output, $\mathbf{x}^t = (x_1^t,..,x_n^t)$ is the external input vector at time t, $\mathbf{w}_j = (w_{1j},...,w_{nj})$ is the weight vector connecting neuron j with the external inputs, $\mathbf{w}_j^r = (w_{1j}^r,...,w_{Hj}^r)$ is its recurrent weight vector, n and H indicate, respectively, the total number of inputs and neurons of a hidden layer and f is a nonlinear, differentiable activation function (GRAVES, 2012).

A useful way to visualize RNNs is to consider the graph formed by 'unfolding' the network along the input sequence. Figure 6 illustrates the functioning of a unit j and its unfolded representation for a window of T observations. It is showing that an output at time t, can be influenced by inputs at any time $t - T$, with $T \in \mathbb{Z}^*$

Figure 6 – Recurrent neural network unit



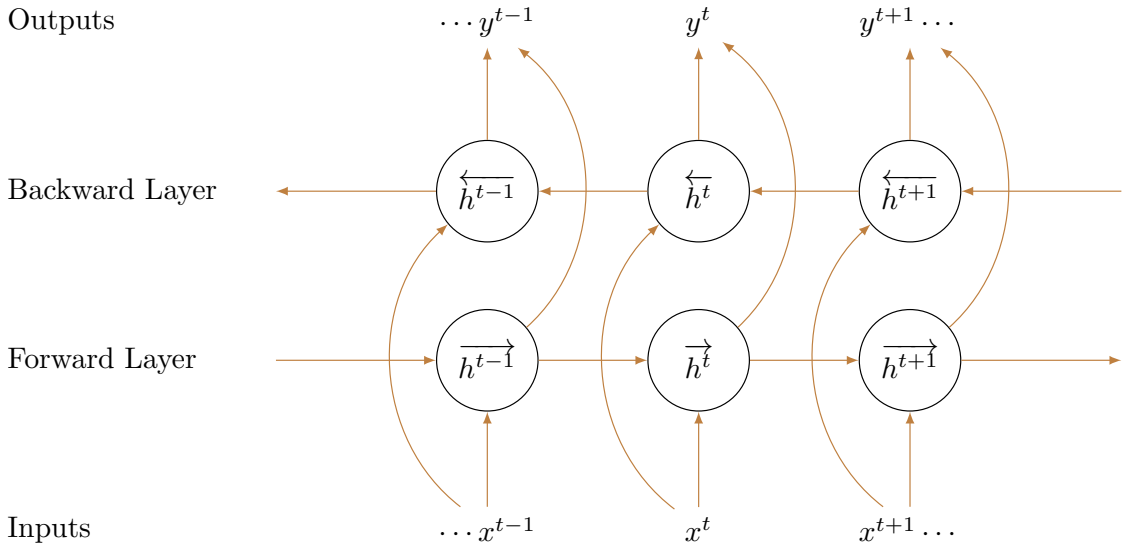
(a) Left - folded view of the recurrent neural network; (b) Right - unfolded view of the network
Source: Adapted from Graves (2012).

2.2.4 Bidirectional Networks

Basic neural networks receive inputs in only one direction, *i.e.*, the oldest data are presented before the most recent data. For recurrent networks, a single direction of data means that the network only has a memory of past contexts. However, for applications such as time series, it is also interesting for the network to know future contexts. The concept of bidirectional networks can be used to implement this idea. For the scenario used in this work though, it is not possible to use actual future data, since this is the data that the network is trying to predict. However we can still use the idea of future and past context within a time window containing the data that is available to the network.

As depicted in Figure 7, bidirectional networks have two hidden (recurrent) layers, which are independent of each other but connected to the same output layer. For one of the layers, data are presented in the natural order of the events, while for the other, data are presented in the reverse sequence. The responses from the two hidden layers are then concatenated and sent to the output layer. This way, one layer analyzes the inputs considering past contexts and another analyzes the inputs considering future contexts (GRAVES, 2012).

Figure 7 – Unfolded bidirectional recurrent neural network



Source: Adapted from https://tikz.net/neural_networks/.

According to Graves (2012), the mathematical formulation of bidirectional networks is the same as other networks, since they are just the junction of two layers. The only modification that occurs is the fact that one of the layers receives the inputs in reverse sequence.

The bidirectional layer could be composed of any kind of recurrent model. In this work, we use LSTM cells to build the bidirectional layer. Thus, the input of the network will be presented, simultaneously, to two LSTM layers, one layer will receive the data within a time window in natural order, whereas the other layer will receive the same data in reverse order. The outputs of both

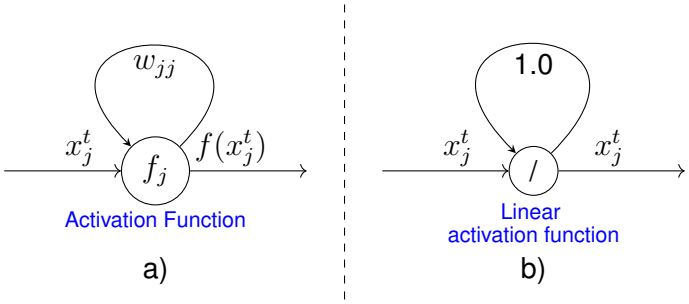
LSTM layers will be combined by a set of neurons generating a single array of outputs, as if it had been processed by a conventional LSTM layer.

2.2.5 LSTM Architecture

In their work on the LSTM architecture, Hochreiter and Schmidhuber (1997) present an issue of recurrent networks with respect to error signals and propose a solution, along with an appropriate learning algorithm. According to them, when using conventional algorithms for training recurrent ANNs, such as Back-Propagation Through Time (BPTT) or Real-Time Recurrent Learning (RTRL), the error signal tends to explode or vanish. When the error grows exponentially, the weights can start oscillating and the network becomes unstable. On the other hand, when the error decreases exponentially, the network cannot learn.

As the error variation is the problem, Hochreiter and Schmidhuber (1997) suggest keeping the error sign constant, with the method called Constant Error Carousel (CEC). This is the central idea of LSTM networks. The CEC approach can be understood considering a single self-recurring unit as depicted in Figure 8a) with a linear activation function, and the weight value fixed as 1.0 for the recurrent connection, as depicted in Figure 8b).

Figure 8 – Example of a recurrent unit



(a) Generic recurrent unit j with a weight w_{jj} connecting the neuron j to itself; b) CEC approach for a recurrent unit.

Source: Author

As shown in Figure 8 a), the recurrent connection of a unit can produce a backpropagated error given by Equation 8.

$$\epsilon_j^t = f_j'\left(x_j^t\right) w_{jj} \epsilon_j^{t+1} \tag{8}$$

where e is the error sign that is used in the backpropagation step, x_j^t is the input at time t directly connected to the neuron j and f' is the derivative of the unit activation function. In Equation (8), to achieve a constant error, i.e. $\epsilon_j^t = \epsilon_j^{t+1}$, we assume

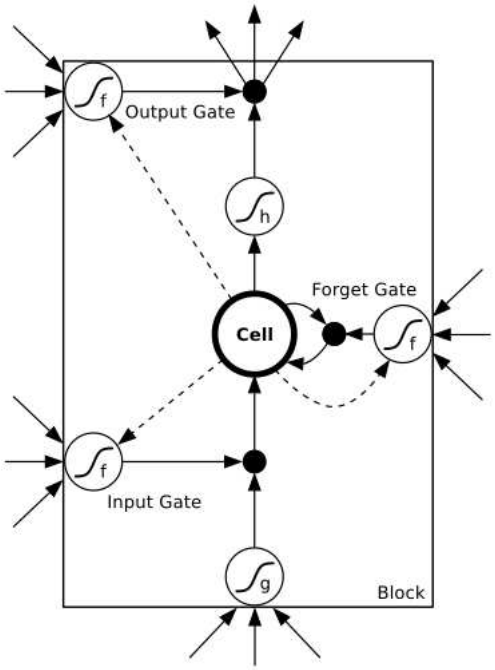
$$f_j'\left(x_j^t\right) w_{jj} = 1 \tag{9}$$

Integrating Equation 9, we get Equation 10.

$$f_j\left(x_j^t\right) w_{jj} = x_j^t \tag{10}$$

As shown in Figure 8 b), in practice we consider $w_{jj} = 1$, which leads us to a linear activation function ($f(u) = u$) (HOCHREITER; SCHMIDHUBER, 1997). That is, the output of the unit is the input itself and the output is reintroduced to the input. In this way, a pulse-type input would cause the cell to keep the same value indefinitely at its input and output.

Figure 9 – Representation of the structure of a memory cell



Source: Graves (2012).

According to Graves (2012), the set of *gates* and activation functions, shown in Figure 9, is called a memory cell. The unit called *cell*, in Figure 9, is the self-recurring unit shown in Figure 8 b), responsible for keeping the internal state of the cell. The three gates are nonlinear summation units that collect activations from inside and outside the block, and control the activation of the cell via multiplications (small black circles). The input and output gates multiply the input and output of the cell while the forget gate multiplies the cell's previous state. No activation function is applied within the cell. The gate activation function 'f' is usually the logistic sigmoid, so that the gate activations are between 0 (gate closed) and 1 (gate open). The cell input and output activation functions ('g' and 'h') are usually tanh or logistic sigmoid, though in some cases 'h' is the identity function. The weighted connections from the cell to the gates are shown with dashed lines. All other connections within the block are unweighted (or equivalently, have a fixed weight of 1.0). The only outputs from the block to the rest of the network emanate from the output gate multiplication.

According to Graves (2012), Hochreiter and Schmidhuber (1997), the memory cell output at instant t is given by Equation 11, with *input*, *forget*, and *output gates* providing, respectively, the outputs y^t_{inG}, y^t_{forgG}, and y^t_{outG}.

$$y^t_c = y^t_{\text{outG}}\ h\left(y^t_{\text{forgG}}\ s^{t-1}_c + y^t_{\text{inG}}\ g\left(\sum_{i=1}^{n} w_{ic}x^t_i + \sum_{j=1}^{H} w^r_{jc}y^{t-1}_j\right)\right) \tag{11}$$

where, c is the CEC cell with activation functions g and h for its input and output, respectively; H is the set of memory cells.

Equation (11) demonstrates that the current state s^t_c kept by the cell, i.e. the argument of the function h, and given by Equation 12.

$$s^t_c = y^t_{\text{forgG}}\ s^{t-1}_c + y^t_{\text{inG}}\ g\left(\sum_{i=1}^{n} w_{ic}x^t_i + \sum_{j=1}^{H} w^r_{jc}y^{t-1}_j\right) \tag{12}$$

is the same as the previous state, however, the percentage of retained information is controlled by the output of *forget gate* at time t. The current cell input goes through the g transformation and is weighted by the output value of the *input gate* at time t. In this way, it can be noticed that the closer the *forget gate* is to 1, the greater the part of the previous information that will be kept; and, the closer the *input gate* is to 1, the more new information will be incorporated into the current state of the cell.

Functions g and h, shown in Figure 9, are two differentiable activation functions that play the role of resizing the input and output of the cell, respectively (HOCHREITER; SCHMIDHUBER, 1997). The function g reduces the dimensionality of the input, as the cell receives a vector of inputs, weighted by weight values and delivers only a scalar value to the current state. On the other hand, after function h, the dimensionality of the cell's output increases, as it receives a scalar value, i.e., the current state of the cell, and delivers the output value, weighted by the weight values, to the other units of the network. Therefore, these functions are responsible for making the input and output dimensions compatible with the cell's internal state through its connections and activations.

The final three components of the memory cell shown in Figure 9 are the *gates*, which have similar functions and formulations.

The output of each *gate* is its activation function f applied to its input. Equations 13, 14 and 15 describe the outputs of input, forget and output gates, respectively.

$$y^t_{\text{inG}} = f\left(\sum_{i=1}^{n} w_{\text{inG}i}x^t_i + \sum_{j=1}^{H} w^r_{\text{inG}j}y^{t-1}_j + \sum_{c} w^s_{\text{inG}c}s^{t-1}_c\right) \tag{13}$$

$$y^t_{\text{forgG}} = f\left(\sum_{i=1}^{n} w_{\text{forG}i} x^t_i + \sum_{j=1}^{H} w^r_{\text{forG}j} y^{t-1}_j + \sum_c w^s_{\text{forG}c} s^{t-1}_c\right) \tag{14}$$

$$y^t_{\text{outG}} = f\left(\sum_{i=1}^{n} w_{\text{outG}i} x^t_i + \sum_{j=1}^{H} w^r_{\text{outG}j} y^{t-1}_j + \sum_c w^s_{\text{outG}c} s^t_c\right) \tag{15}$$

where s^t_c is the current state of cell c, $w_{\text{inG}}*$, $w_{\text{forgG}}*$ and $w_{\text{outG}}*$ are the weight values for *input*, *forget* and *output* gates, respectively, which are connected to the external input, other hidden units or to the cell state (GRAVES, 2012).

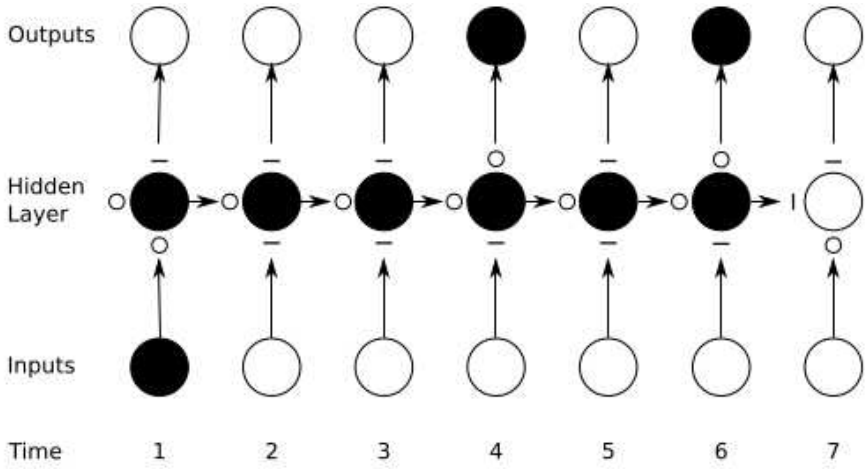
Analyzing Equations (11) to (15), we notice that the gates are responsible for controlling the amount of information that enters (*input gate*), leaves (*output gate*), or remains (*forget gate*) in the cell. Such control is done by multiplying, as in Equation (11), the *gate* output, i.e., its activation value, by the information to be controlled. As the activation value f of the *gates* is usually fixed as sigmoid, it varies between 0 and 1, and the result of this multiplication will always be a percentage of the original value of the information.

It is important to note that the constant error signal occurs only in the CEC cell as it has a recurrent weight fixed as 1.0 and a linear activation function. The weights of the *input*, *forget* and *output* gate ($w_{\text{inG}}*$, $w_{\text{forgG}}*$ and $w_{\text{outG}}*$) are updated in a standard way during the *backpropagation* step, otherwise, the network would never be able to learn, as the error would never be reduced.

The idea of having a constant error is just to ensure that the internal state of a cell is able to store information for countless instants of time. Likewise, when necessary, according to what the *gates* learned during training, the cell is able to release this information and/or allow more information to be stored. However, the internal state of the cell does not modify (loses) the information being stored, due to an eventual reduction of the error signal. The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby mitigating the exponential increase or decrease of the influence of recurrent inputs on the network's output, known as the vanishing gradient problem, when the gradient approximates to zero very fast, and maybe completely stopping the neural network from further training (BASODI *et al.*, 2020).

Figure 10 illustrates the preservation over time of gradient information by an LSTM unit. In this example, as long as the input gate remains closed (i.e. has an activation near 0), the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate. In the image, the large circles represent the information being presented to the cell (labeled as 'Inputs'), the information being kept in the cell's internal state (labeled as 'Hidden Layer') and the information leaving the cell (labeled as 'Outputs'). Black circles represent one possible value, e.g. off or zero, and white circles represent another possible value, e.g. on or one. The small circles around the Hidden Layer's cell indicate the status of each gate: input positioned below

Figure 10 – Information gradient preservation by LSTM



Source: Graves (2012).

the cell, output positioned above the cell or forget position at the left side of the cell. Circle signs indicate that the gate is open, and the dash signs indicate that the gate is closed. This diagram is showing a simplification of the aforementioned concept, where information can flow into the cell when the input gate is open (Hidden Layer's input has a small circle in it), and that same information remains in the cell for as long as the forget gate is open (Hidden Layer's connection to the next time step has a small circle in it). On the output side, we can notice that the output gate can control when the current cell state will be passed forward (small circle on the output) or not (dash sign on the output).

2.3 Online *versus* Offline learning of neural models

Offline learning, also known as batch learning, involves training machine learning models using the entire dataset or large batches of data. Offline learning is suitable when the entire dataset is available beforehand or when the computational resources allow the processing of larger batches efficiently. The traditional gradient descent method is an example of this type of learning. Many machine learning paradigms, particularly neural network models, often work in a batch learning or offline learning fashion. In this approach, the model processes data samples in batches and updates the weights based on the average gradient computed across each batch.

Used mainly in supervised training of MLPs, traditional offline learning updates model parameters by some learning algorithm from an entire training dataset at once (it might also consider validation processes), and then the model is deployed for inference without performing any update afterward. Such learning methods suffer from expensive re-training costs when dealing with new training data, and thus are poorly scalable for real-world applications. In the era of big data, traditional batch learning paradigms become more and more restricted, especially when live data grow and evolve rapidly (HOI *et al.*, 2021).

Besides facing huge volumes of data, computational systems are exposed to a continuous flow of information when dealing with some practical problems and thus are required to learn from dynamic data distributions. Learning capabilities are therefore crucial for such systems and autonomous agents interacting in the real world and processing continuous streams of information (PARISI *et al.*, 2019).

Online learning, also known as incremental learning, involves updating model parameters continuously as new data become available. In this approach, the model learns from individual data samples in real time. The model is updated incrementally after each sample, adapting to new information and potentially adjusting its predictions.

According to Hoi *et al.* (2021), depending on the types of learning tasks and the forms of feedback information, the existing online learning works can be classified into three major categories: (i) online supervised learning where full feedback information is always available, this is the type adopted in the present work; (ii) online learning with limited feedback, and (iii) online unsupervised learning where no feedback is available.

In the context of ANNs, online learning is used for updating the weights of neural models after processing each training sample. However, online learning remains a challenge for ANNs since the continual acquisition of incrementally available information from non-stationary data distributions generally leads to catastrophic forgetting or interference, i.e., training a model with new information interferes with previously learned knowledge (PARISI *et al.*, 2019).

Mini-batch gradient descent is a compromise between online which trains using a unique sample at a time and offline learning which uses the entire dataset. It processes the data in small batches, typically ranging from a few tens to a few hundreds of samples, to balance the benefits of processing in parallel and update efficiency. This is the approach adopted in the present work in the offline learning mode even when training non-deep models (e.g. autoencoders and MLPs).

There are several studies on ANN using either online or offline learning; discussing each of these works is out of the scope of this work. Although less frequent, works exploring both methods are likewise not new (KESKINOCAK, 1998). Puttige and Anavatti (2007) compare the methods when training two network models used to calculate lateral and longitudinal dynamics of an unmanned aerial vehicle. Bessa, Miranda and Gama (2009) adopt entropy concepts to the training of neural networks aimed at predicting wind power based on speed and direction characteristics for wind parks connected to a power grid. Even though the authors do not compare both methods, they use them to evaluate the benefits of introducing the entropy concepts. More recently, Zhang, Bengio and Liu (2017) propose new benchmarks for both online and offline handwritten Chinese character recognition, as well as a deep learning model to solve them. Less related to ANN learning, but also in the deep context, Lee *et al.* (2022) propose a method that balances online and offline information aiming to improve sample efficiency and final performance of fine-tuned robotic agents on various locomotion and manipulation tasks.

It is important to point out that the choice between online and offline learning depends on factors such as the available computational resources, the nature of the data stream, and

the desired trade-off between responsiveness and accuracy. Different learning algorithms and variations can be employed within each approach to optimize training performance.

2.3.1 Training Feedforward Neural Models

Training a neural network is the process in which the free parameters of the network, e.g. weights and bias, are changed by the continuous stimulation caused by the environment the network is inserted in; that is, the network parameters are iteratively adjusted. In this work, all models are used in a supervised way where there is one (or more) target output(s) for each training input pattern.

MLP networks most of the time are trained through the backpropagation algorithm. In backpropagation, for each pattern or a set of patterns presented at iteration t, the output produced by the neural network is compared with the desired outcome through a *loss function* $\mathcal{L}$, which measures how good the model performs in terms of being able to predict the expected outcome – in case of hidden layers such an outcome is estimated based on the backpropagated error.

Then, for each layer, the adjustments $\Delta(\mathbf{w}(t))$ in the weights $\mathbf{w}$ can be calculated to minimize the error in the output according to the gradient descent (Equation 16).

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta(\mathbf{w}(t)) = \mathbf{w}(t) - \eta\nabla\mathcal{L} \tag{16}$$

where η is the step size of each update, and $\nabla\mathcal{L}$ is the gradient of the cost or loss function $\mathcal{L}$ as shown in Equation 17.

$$\nabla\mathcal{L} = \left(\frac{\partial\mathcal{L}}{\partial w_{1j}}, \frac{\partial\mathcal{L}}{\partial w_{2j}}..., \frac{\partial\mathcal{L}}{\partial w_{ij}}, ...\right) \tag{17}$$

with a component of the gradient vector associated with w_{ij}, i.e., the weight connecting neuron i with neuron j, given by Equation 18.

$$\frac{\partial\mathcal{L}}{\partial w_{ij}} = \frac{\partial\mathcal{L}}{\partial y_j}\frac{\partial y_j}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}} = \epsilon_j f'(u_j)\text{input}_i = \delta_j\text{input}_i \tag{18}$$

where ϵ_j is the output error or the error backpropagated from the next layer to the current layer; $\delta_j = \epsilon_j f'(u_j)$; and input_i is the input directly connected to the weight w_{ij}.

We can also adopt the version of backpropagation with momentum (Equation 19):

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta(\mathbf{w}(t)) + \zeta\Delta(\mathbf{w}(t-1)) \tag{19}$$

where ζ is the momentum factor (GRAVES, 2012).

In autoencoder models, weights and biases are usually randomly initialized and then updated iteratively during training through backpropagation, which is performed just like other feedforward neural networks. In other words, it aims to minimize the reconstruction error measured by a "loss" function $\mathcal{L}$, such as the squared error (Equation 20).

$$\mathcal{L}(\mathbf{x},\hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|_p^2 = \|\mathbf{x} - f^d(\mathbf{W}^d(f^e(\mathbf{W}^e\mathbf{x} + \mathbf{b}^e)) + \mathbf{b}^d)\|_L^2 \tag{20}$$

where $\|.\|_L^2$ is the squared error dependent on the L-norm, $\mathbf{x}$ is the input vector, $\hat{\mathbf{x}}$ is the output vector (from Equations 3 and 4), $\mathbf{b}^e = (b_1^e,...,b_H^e)$, $\mathbf{b}^d = (b_1^d,...,b_n^d)$, are the vectors encompassing all the biases of encoder and decoder neurons respectively, $\mathbf{W}^e = [\mathbf{w}_1^e,\mathbf{w}_2^e,...,\mathbf{w}_H^e]'$ is the matrix of encoder weights, $\mathbf{W}^d = [\mathbf{w}_1^d,\mathbf{w}_2^d,...,\mathbf{w}_n^d]'$ is the matrix of decoder weights (DONG *et al.*, 2018).

2.3.2 Training Recurrent Neural Models

In recurrent neural networks, time t is important for the adjustments. Then neuron inputs and outputs are now identified by the time step t. There are different algorithms proposed to train RNNs, in this work, we adopt the BPTT algorithm. It assumes that the same weights are reused at every time step. Then, we have Equation 18 adapted to sum over the whole sequence to get the derivatives with respect to the network recurrent weight w_{ij}^r connecting neurons i and j as Equation 21.

$$\frac{\partial \mathcal{L}_p}{\partial w_{ij}^r} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_p}{\partial y_j^t} \frac{\partial y_j^t}{\partial u_j^t} \frac{\partial u_j^t}{\partial w_{ij}^r} = \sum_{t=1}^{T} \epsilon_{jbkp}^t f'(u_j^t) \text{input}_i^t = \sum_{t=1}^{T} \delta_j^t \text{input}_i^t \tag{21}$$

where $\delta_j^t = \epsilon_{jbkp}^t f'(u_j^t)$ and $\text{input}_i^t = y_j^t$ due to the recurrence.

Equation 21 shows that like standard backpropagation, BPTT consists of a repeated application of the chain rule. The difference is that, for recurrent weights, the error backpropagated to neuron j at instant t (ϵ_{jbkp}^t) depends not only on the output layer but also on the hidden layer at the next time step, as shown in Equation 22.

$$\delta_j^t = f'(u_j^t)\epsilon_{jbkp}^t = f'(u_j^t) \left(\sum_{k=1}^{K} \delta_k^t w_{jk} + \sum_{j'=1}^{H} \delta_{j'}^{t+1} w_{jj'} \right) \tag{22}$$

The complete sequence of δ_j^t can be calculated by starting at $t = T$ and recursively applying Equation 22, decrementing t at each step and assuming $\delta_j^{T+1} = 0$, since no error is received from beyond the end of the sequence.

The main steps performed by BPTT algorithm are described below.

1. LSTM *feedforward pass*:

   For every **cell** in the hidden layer

- for $t = 1,..,T$ in Forward layer (F) and $t = T,..,1$ in Backward layer (B).
- calculate the output of the current **cell** for each pair of elements in F or B.

Calculate LSTM output for pattern p encompassing all the **cell**.

2. LSTM *backward pass*:
   Calculate the errors and δs for cells and gates;
   Calculate the adjustment for each weight due to each pattern in each memory cell;
   For every layer, accumulate the adjustments and update the weights for the current batch.

In Section 3.2.1 we detail the steps performed in the present work in both phases of the BPTT algorithm, adapted to feedforward information from a bidirectional LSTM model to an MLP positioned next in the proposed pipeline.

3 METHODOLOGY

In the present work, ANN-based components are proposed to compose MLP-based models and pipelines of hybrid models, aimed at solving a classification problem resulting from the smart home problem formulation. In summary, each problem has at most 5 classes: up to 4 classes indicating activities in a particular room at the smart home being addressed and NA indicating no activity in the room. Additionally, two LSTM-based models are also proposed, one encompassing only one biLSTM model and another with two.

3.1 The Proposed Approaches

The proposed approaches aim to answer the questions raised in the introduction. For this, we build the six different approaches described in Table 1.

Table 1 – Description of proposed approaches

Name	Description
Model1	a pure multi-layer perceptron model (see Figure 11);
Model2	a hierarchical model composed of two MLP-based classifiers used in the pipeline (see Figure 12);
Model3	hybrid and hierarchical model encompassing an autoencoder and two MLP classifiers in its pipeline (see Figures 13);
Model4	hybrid and hierarchical model encompassing an autoencoder and two MLP classifiers in its pipeline (see Figures 13);
Model5	a model containing a bidirectional LSTM layer and MLP layers (see Figure 15);
Model6	a hierarchical model containig two classifiers composed by a bidirectional LSTM layer and MLP layers (see Figure 16).

Although there is no consensus on how many layers a model should encompass to be considered deep, we assume that, except for the first one, all the remaining proposed models involve learning a sequence of representations via composite functions or modules. Therefore, they could be considered deep models built up from shallow components; the two last approaches are classified as deep recurrent models.

Table 2 shows the list of components that are used by the models described in the following sections.

3.1.1 Pure Multilayer Perceptron Model

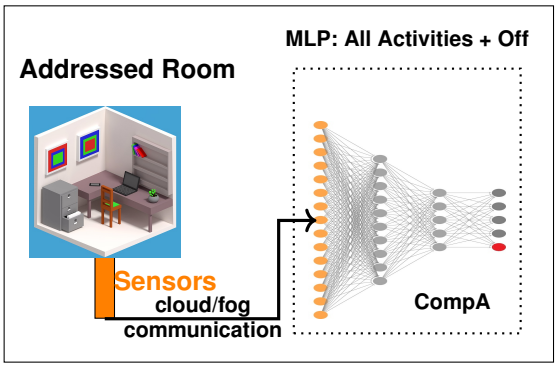
This section describes the Pure MLP model (**Model1**) depicted in Figure 11. It is composed of a unique module encompassing a Multi-class plus 1 component (**CompA**), which is responsible for the whole classification process. The CompA component has an input layer with n neurons (n is #S), 2 hidden layers with $H_1 > H_2$ neurons, an f=tanh activation function, and

Table 2 – Description of components used by the models

Name	Description
CompA	an MLP component composed of multiple layers, with the output having one neuron for each class plus a 'no activity' neuron;
CompB	an MLP composed of multiple layers, with the output having only 2 neurons, representing on and off states;
CompC	an MLP component composed of multiple layers, with the output having only 2 neurons, representing on and off states. This component has more neurons per layer than CompB;
CompD	an MLP component composed of multiple layers, with the output having one neuron for each class (without a neuron for 'no activity');
CompE	an MLP component composed of multiple layers, with the output having one neuron for each class (without a neuron for 'no activity'). This component has more neurons per layer than CompD;
CompF	an autoencoder component composed of only one hidden layer, and with the same number of neurons in the input and output layers;
CompG	a component composed of a bidirectional LSTM layer and multiple MLP layers, with the output having one neuron for each class plus a 'no activity' neuron;
CompH	a component composed of a bidirectional LSTM layer and multiple MLP layers, with the output having only 2 neurons, representing on and off states;
CompI	a component composed of a bidirectional LSTM layer and multiple MLP layers, with the output having one neuron for each class (without a neuron for 'no activity');
$\mathcal{L}$**mod**	a component to calculate the loss value.

an output layer with $\#C + 1$ neurons using f=tanh+softmax; each neuron is associated with a particular class or activity and the last one indicates a non-activity output (the red output depicted in Figure 11). This proposal aims to answer the question regarding the possibility of a simple standalone MLP being capable of solving the whole classification problem.

Figure 11 – Model1: a proposed model composed of an MLP classifier (C types of activities plus one with no activity) with 2 hidden layers.



Source: Author

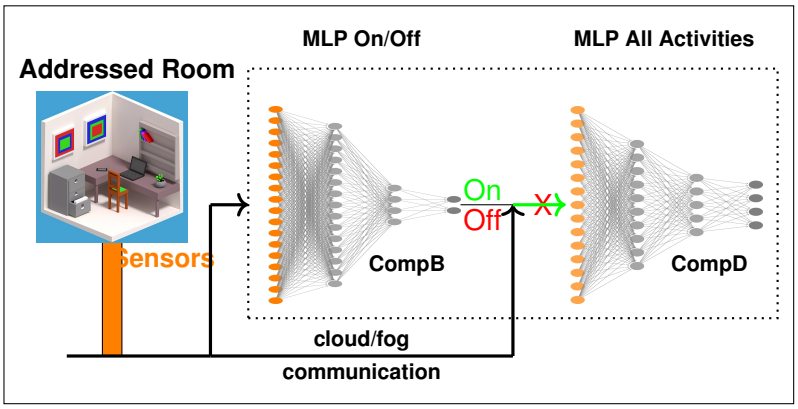
3.1.2 Hierarchical Multilayer Perceptron Model

This section describes the hierarchical model (**Model2**) depicted in Figure 12. It encompasses two components:

1. **CompB (an MLP on/off)** - input layer with $n = \#S$ neurons, 2 hidden layers with $H1 > H_2$ neurons, f=tanh, and an output layer with 2 neurons, f=tanh+softmax, indicating if there is (On) or there is not (Off) an activity occurring in the addressed room.

2. **CompD (an MLP for all activities)** - input layer with $n = \#S$ neurons, 2 hidden layers with $H_1 > H_2$ neurons, f=tanh, and an output layer with $\#C$ neurons, f=tanh+softmax, each neuron associated with a particular activity.

Figure 12 – Model2, a proposed model with two Hierarchical MLPs : an MLP classifier (on/off) with 2 hidden layers and an MLP classifier (C types of activities) also with 2 hidden layers.



Source: Author

3.1.3 Hybrid Small Model

This section describes the Hybrid with Small components model (**Model3**) depicted in Figure 13. It encompasses four modules (three ANN shallow components and a Loss module) in its pipeline CompF $\rightarrow \mathcal{L}$mod $\rightarrow$ CompB $\rightarrow$ CompD:

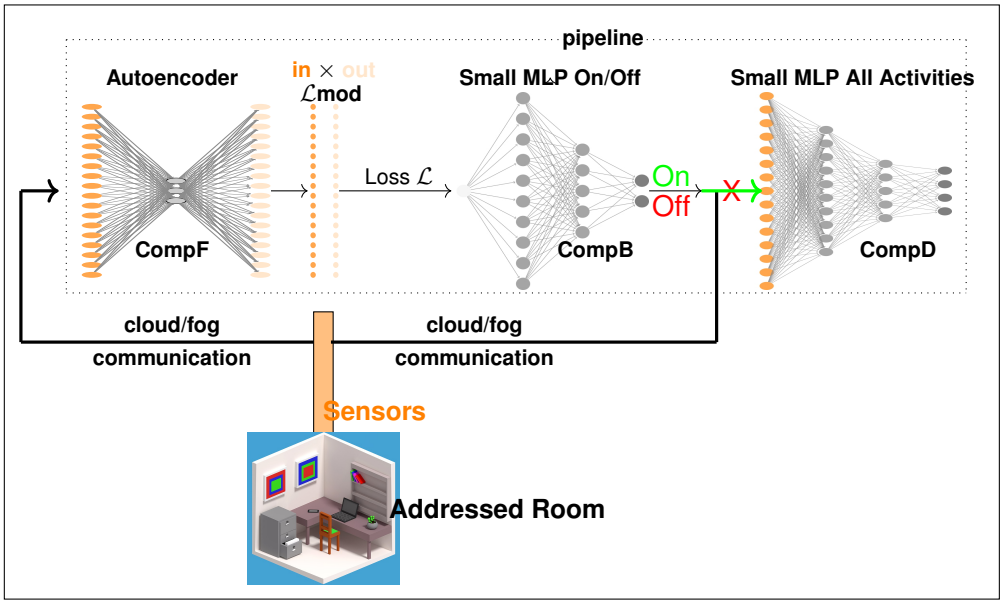
1. **CompF (Autoencoder)** - input layer with $n = \#S$ neurons, 1 hidden layer with 3 neurons, an output layer also with n neurons.

2. **$\mathcal{L}$mod (loss module)** - it calculates the difference between the autoencoder input $\mathbf{x}$ and its output $\hat{\mathbf{x}}$; then it uses it to update the autoencoder weights and also to feed the next component. It might update the i^{th} input importance ξ_i to the target output to provide the loss function whenever the weighted version is adopted.

3. **CompB (small MLP on/off)** - input layer with 1 neuron (i.e. the output received from $\mathcal{L}$mod), 2 hidden layers with $H_1 > H_2$ neurons, f=tanh, and an output layer with 2 neurons, indicating if there is (On) or there is not (Off) an activity occurring in the addressed room.

4. **CompD (small MLP all activities)** - an input layer with $n = \#S$ neurons, f=tanh, 2 hidden layers with $H_1 > H2$ neurons, and an output layer with $\#C$ neurons, f=tanh+softmax, each one associated with a particular class or activity.

Figure 13 – Model3, a model composed of an autoencoder with #S inputs/outputs plus a module that calculates how good is the reconstruction, an MLP classifier (on/off) with 2 hidden layers and finally, an MLP classifier (types of activities) also with two hidden layers encompassing small MLP components.



Source: Author

3.1.4 Hybrid Large Model

This section describes the Hybrid with Large components model (**Model4**) depicted in Figure 14. Its pipeline also encompasses four modules CompF $\rightarrow$ $\mathcal{L}$mod $\rightarrow$ CompC $\rightarrow$ CompE of the previous model, except for the numbers of neurons in each hidden layer of components CompC and CompE that are quite larger than the previous model.

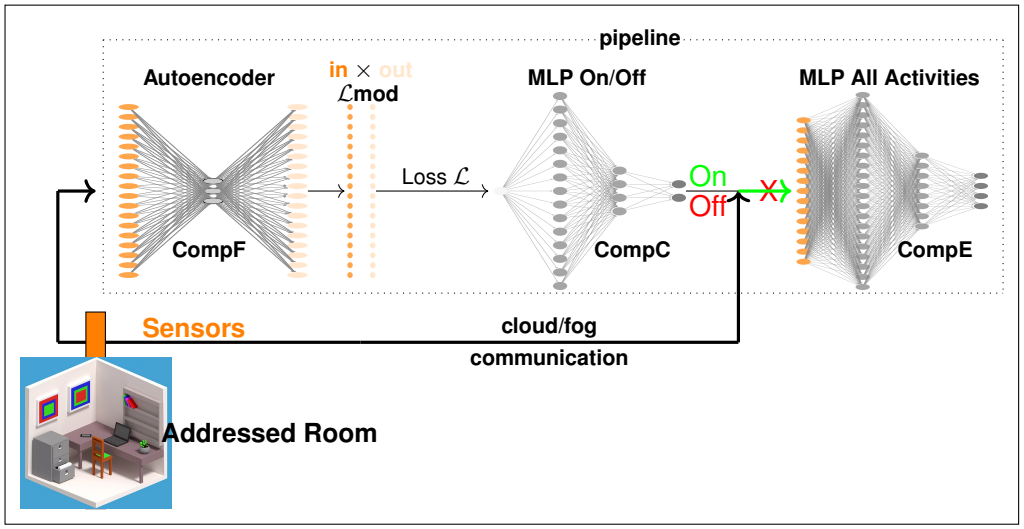
1. **CompF (Autoencoder)** - the same component described in the previous model.

2. **$\mathcal{L}$mod (loss module)** the same module previously described.

3. **CompC (an MLP on/off)** - the same component described in the first two models.

4. **CompE (an MLP all activities)** - the same component described in the first two models.

Figure 14 – Model4, a model composed of an autoencoder with #S inputs/outputs plus a module that calculates how good is the reconstruction, an MLP classifier (on/off) with 2 hidden layers and finally, an MLP classifier (types of activities) also with two hidden layers encompassing large MLP components.



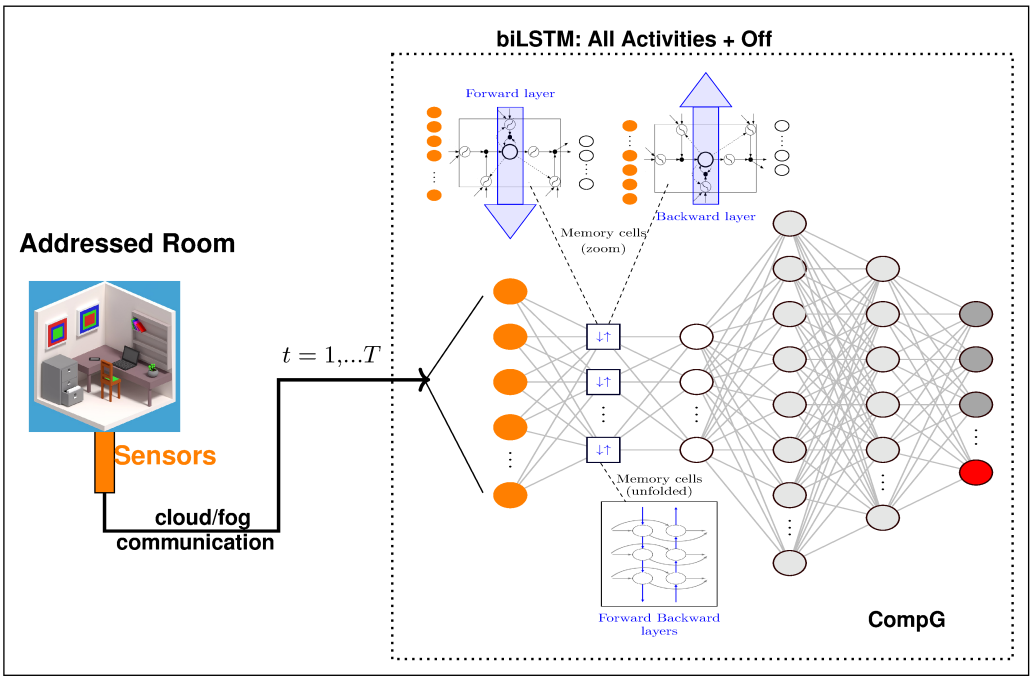
Source: Author

3.1.5 Simple Bidirectional LSTM

This section describes the Bidirectional LSTM model (**Model5**) depicted in Figure 15. It is composed of a unique module encompassing a recurrent Multi-class plus 1 component (**CompG**), which is responsible for the whole classification process.

- **CompG** component can be described as a hybrid model encompassing a bidirectional LSTM and an MLP model. It is composed of an input layer with $\#S$ neurons, each one receiving from the sensor s an input sequence of size T. This way, the entire input of the network is composed of a matrix $X_{T \times \#S}$, i.e. a matrix with T rows, and $\#S$ columns, indicating all the $\#S$ sequences (one for each sensor) of size T. Each hidden layer (feedforward or backward) of the bidirectional LSTM is composed of H recurrent units (memory cells) each using $f^{\iota} = ReLU$ as activation function, and whose characteristics and unfolded version are depicted in detail in the top of Figure 15. The outputs of the recurrent units in the hidden layer of the biLSTM model are used as inputs for an MLP with 2 hidden layers composed of $H_1 > H_2$ neurons using f=softmax as activation function, and an output layer with $\#C + 1$ neurons using f=softmax; each output neuron is associated with a particular class or activity and the last one indicates a non-activity output (the red output).

Figure 15 – Model5: a proposed model based on a hybrid classifier (C types of activities plus one in red with no activity) with a biLSTM plus an MLP with 2 hidden layers.



Source: Author

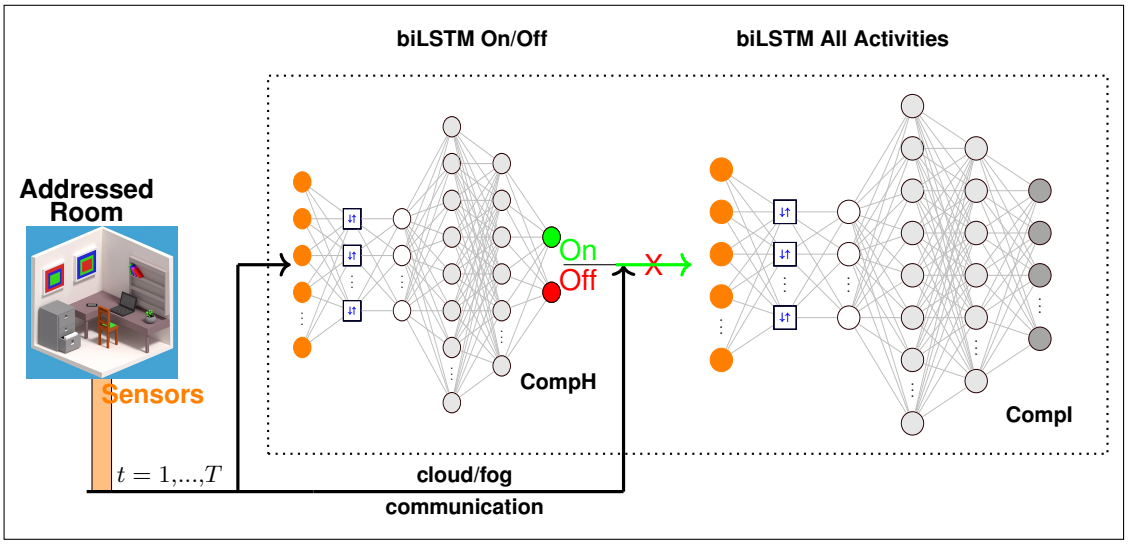
3.1.6 Hierarchical Bidirectional LSTM

This section describes the LSTM hierarchical model (**Model6**) depicted in Figure 16. It is composed of two components:

1. **CompH (a bidirectional LSTM for on/off activity)** - input layer with $n = \#S$ neurons, a bidirectional LSTM hidden layer with H units and with $f^{\iota} = ReLU$ and $f^{g} = tanh$, linked with two MLP hidden layers with $H_1 > H_2$ and f=softmax, and finally an output layer with 2 neurons and f=softmax, indicating if there is (On) or there is not (Off) an activity occurring in the addressed room.

2. **CompI (a bidirectional LSTM for all activities)** - Similarly to the model described in the previous section, here we have a hybrid component encompassing a bidirectional LSTM and an MLP model. It is also composed of an input layer with $\#S$ neurons, each one receiving from the sensor s an input sequence of size T. Each hidden layer (feedforward or backward) of the bidirectional LSTM is composed of H recurrent units (memory cells) with $f^{\iota} = ReLU$ and $f^{g} = tanh$ as the activation and recurrent activation function, respectively. The outputs of the recurrent units in both hidden layers of the biLSTM model are used as inputs for an MLP with 2 hidden layers composed of $H_1 > H_2$ neurons using f=softmax as activation function, and an output layer with $\#C$ neurons using f=softmax; each neuron is associated with a particular class or activity.

Figure 16 – Model6, a proposed model with two hybrid LSTMs: a biLSTM classifier (on/off) with one bidirectional hidden layer whose outputs are inputs of an MLP and a biLSTM classifier for C types of activities, also with one bidirectional hidden layer whose outputs are inputs of the final MLP in the pipeline.



Source: Author

3.2 Training the shallow components

The four shallow components used by the MLP-based proposed approaches are:

- **CompF**: an autoencoder with one single hidden layer to detect outliers (e.g., sparse room activities);
- **CompC** and **CompB**: MLP binary classifiers conceived to separate activities from non-activities;
- **CompE** and **CompD**: MLP classifiers designed with a softmax final layer to separate among the different possible activities in a particular room;
- **CompA**: an MLP classifier designed with a softmax final layer to perform the entire classification task (i.e. to separate among all possible classes: up to 4 different possible activities in the room plus non-activity NA).

Besides the first three shallow components (CompF; CompB or CompC; CompD or CompE), the hybrid approaches (Model3 and Model4) also encompass:

- $\mathcal{L}$**mod**: a component that compares the autoencoder input and the reconstructed output.

The LSTM models use the following shallow components:

- **CompH**: a bidirectional LSTM followed by an MLP with two hidden layers and two output neurons for on/off activity;

- **CompI**: a bidirectional LSTM followed by an MLP with two hidden layers and $\#C$ output neurons for all activities;
- **CompG**: a bidirectional LSTM followed by an MLP with two hidden layers, $\#C + 1$ output neurons for all activities plus a non-activity NA.

One training parameter of paramount importance for all models is the *loss function* $\mathcal{L}$. The $\mathcal{L}$ setting depends on the type of component being considered and the learning mode taking place, which can be conducted in an offline or online way. In the present work, we adopt three different loss functions ($\mathcal{L}$), each one being used by one or more of the components as described in Table 5:

- cross-entropy: $\mathcal{L}(\mathbf{y},\hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \cdot \log \hat{y_k}$
- squared error: $\mathcal{L}(\mathbf{y},\hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \sum_{k=1}^{K}((y_k - \hat{y}_k))^2$
- weighted squared error: $\mathcal{L}_\xi(\mathbf{y},\hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_{2\xi}^2 = \sum_{k=1}^{K}(\xi_k(y_k - \hat{y}_k))^2$

where $\mathbf{y}$ and $\hat{\mathbf{y}}$ are, respectively, the vectors of target and estimated outputs of the considered component, K is the number of neurons in the output layer with $K \in \{2, \#C, \#C + 1\}$; and ξ_k is estimated before training based on the correlation input $\times$ output. Each single output $\hat{y}_k$ in $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2,...\hat{y}_K)$ is given by $\hat{y}_k = f(u)$ where f is the activation function and u is the *activation potential* given by Equation 23, derived from Equation 1.

$$u = \mathbf{w}_k \cdot \mathbf{y}^{\text{hidden}} + b_k = \sum_{j=1}^{H} w_{jk} y_j + b_k \tag{23}$$

Notice that u depends on the neuron bias b_k, weights $\mathbf{w}_k$ and $\mathbf{y}^{\text{hidden}}$, i.e., the output of the last hidden layer which is composed of H neurons. It measures the compatibility between the H values $y_1, y_2, \ldots, y_H$ received by the kth output neuron and its H weights $w_{1k}, w_{2k}, \ldots, w_{Hk}$.

As depicted in Figures 13 and 14, in the hybrid proposed approaches (Model3 and Model4), the rule of *loss function* can extrapolate the training phase, as it can also support the inference process. In our hybrid models, it is used as an input of CompB or CompC components in addition to setting the weights/bias updates.

In the learning of shallow components, the target output depends on which component is being trained. Considering the whole set of training data $\{(\mathbf{x},\mathbf{y})_p\}_{tr}$, whose cardinality is given by $P = |\{(\mathbf{x},\mathbf{y})_p\}_{tr}|$, we have the inputs and target outputs defined as:

- CompF : $\{(\mathbf{x}_p,\hat{\mathbf{x}}_p)\}$, $p = 1,...,P$ with $\hat{x}_{ip} = x_{ip}$, $i = 1,..,n$;

and for all the other components we have one-hot encoding:

- CompB, CompC CompH: $\{(\mathbf{x}_p,\mathbf{y}_p)\}$, with $\mathbf{y}_p = \begin{cases} (1,0) & \text{if the target class is on} \\ (0,1) & \text{if the target class is off} \end{cases}$
- remaining ones: $\{(\mathbf{x}_p,\mathbf{y}_p)\}$, with

  $\mathbf{y}_p = (y_{1p},...,y_{Kp})$,

  $c_k \in \{c_1,c_2,...c_{\#C}\}$,

  $Kp = \#C$ and

  $y_{kp} = \begin{cases} 1 & \text{if the target class is } c_k \\ 0 & \text{otherwise} \end{cases}$

For both modes of learning (offline and online), independently of which type of component, the weights and biases are randomly initialized, and then iteratively updated through the supervised learning performed by the chosen optimizer. All shallow components use an optimizer whose basis is the backpropagation algorithm.

3.2.1 Offline Learning

In the offline learning conducted in the present work, the whole set of training data $\{(\mathbf{x},\mathbf{y})_p\}_{tr}$, $p = 1,...,P$, is available and the testing phase using $\{(\mathbf{x},\mathbf{y})_\varsigma\}_{ts}$, $\varsigma = 1,...,S$ occurs only when the model finishes its training. Aiming to diminish the dependence of results on the data partition, a 5-fold cross-validation process is also considered. In this case, we consider a validation phase using $\{(\mathbf{x},\mathbf{y})_v\}_{vl}$, $v = 1,...,V$ to decide when the training should stop.

The training data have been divided into batches $\mathcal{B}$, each one with size $|\mathcal{B}| > 1$ (for offline learning experiments). At each iteration and for each pattern p in the current batch $\mathcal{B}$, the output vector $\hat{\mathbf{y}}_p = (\hat{y}_{1p},\hat{y}_{2p},...,\hat{y}_{Kp})$, with the k-th element given by $\hat{y}_{kp} = f(\mathbf{w}_k \cdot \mathbf{y}_p^{hidden} + b_k)$ with bias b_k and weight vector $\mathbf{w}_k$, is obtained in the *feedforward pass* as Equation 24.

$$\hat{\mathbf{y}}_p = f(W^{out} \cdot \mathbf{y}_p^{hidden} + \mathbf{b}^{out}) \tag{24}$$

where W^{out} is the matrix of output weights with K lines and H_{hidden} columns, each line corresponding to the vector $\mathbf{w}_k$.

The vector $\mathbf{y}_p^{hidden} = (y_{1p}^{hidden},y_{2p}^{hidden},...,y_{Hp}^{hidden})$ at the output of the last hidden layer is described by Equation 25.

$$\mathbf{y}_p^{hidden} = f(W^{hidden} \cdot \mathbf{y}_p^{hidden-1} + \mathbf{b}^{hidden}) \tag{25}$$

where W^{hidden} is a matrix $H_{hidden} \times H_{hidden-1}$, and $y_{jp}^{hidden} = f(\mathbf{w}_j \cdot \mathbf{y}_p^{hidden-1} + b_j)$ is the k-th element of $\mathbf{y}_p^{hidden}$. The *backward pass* is repeated until $hidden - 1$ is the input layer, in this case, $\mathbf{y}_p^{hidden-1} = \mathbf{x}_p$.

The output vector $\hat{\mathbf{y}}_p$ is then compared with the target output vector $\mathbf{y}_p$, and the prediction error ϵ_p is calculated through the *loss function* $\mathcal{L}_p(\mathbf{y}_p,\hat{\mathbf{y}}_p)$[1]. For every layer, the adjustments (Equation 26) are calculated to minimize the error in the output according to the mini-batch gradient descent.

$$\Delta_{\mathcal{B}}(\mathbf{w}(t)) = -\eta\nabla(\frac{1}{|\mathcal{B}|}\sum_{p=1}^{|\mathcal{B}|}\mathcal{L}_p) = -\eta\frac{1}{|\mathcal{B}|}(\sum_{p=1}^{|\mathcal{B}|}\nabla\mathcal{L}_p) \tag{26}$$

where η is the learning rate and $\nabla\mathcal{L}_p = \left(\frac{\partial\mathcal{L}_p}{\partial w_{1j}}, \frac{\partial\mathcal{L}_p}{\partial w_{2j}}, ..., \frac{\partial\mathcal{L}_p}{\partial w_{ij}}, ...\right)$ for $\mathbf{w} = (w_{1j},w_{2j},..., w_{ij},...)$ is the vector of all weights connected to a neuron j. For every component w_{ij} of the gradient vector, the adjustment is given by Equation 27.

$$\frac{\partial\mathcal{L}_p}{\partial w_{ij}} = \frac{\partial\mathcal{L}_p}{\partial y_{jp}}\frac{\partial y_{jp}}{\partial u_{jp}}\frac{\partial u_{jp}}{\partial w_{ij}} = \epsilon_{jp}f'(u_{jp})\text{input}_{ip} \tag{27}$$

where $\frac{\partial\mathcal{L}_p}{\partial y_{jp}} = \epsilon_{jp}$ is the error for neuron j for pattern p, $\frac{\partial y_p}{\partial u_{jp}} = \frac{\partial f(u_{jpp})}{\partial u_{jp}} = \frac{df(u_{jp})}{du_{jp}} = f'(u_{jp})$ is the derivative of activation function of neuron j with respect to u_{jp}, and $\text{input}_{ip} = \frac{\partial u_{jp}}{\partial w_{ij}}$ is the p-th input directly connected to w_{ij}.

For every weight associated with a neuron in the output layer, the error ϵ_p can be easily calculated based on the difference between the target and estimated outputs. However, for the weights in all other layers, the error depends on neurons in the following layers the weight is connected to. In this case, the error is referred to as retro-propagated error (ϵ_{bkp}), and the adjustment of the weight w_j of a neuron j in the current layer is given by Equation 28.

$$\frac{\partial\mathcal{L}_p}{\partial w_j} = \frac{\partial\mathcal{L}_p}{\partial y_{jp}}\frac{\partial y_{jp}}{\partial u_{jp}}\frac{\partial u_{jp}}{\partial w_j} = \epsilon_{jbkp}f'(u_{jp})\text{input}_{jp} \tag{28}$$

where $\epsilon_{jbkp} = \sum_k \delta_{kp}w_{jk}$, with w_{jk} indicating a weight connecting a neuron j in the current layer and k in the next layer, $\delta_{kp} = \epsilon_{kp}f'(u_{kp})$, and input_{jp} is the input directly connected to w_j.

Then, the value of the gradient for each pattern ($\nabla\mathcal{L}_p$) is accumulated to provide $\Delta_{\mathcal{B}}(\mathbf{w}(t))$ and the weight updates occur only when each batch is complete. Based on backpropagation with momentum (Equation 19), the weight update will be calculated as per Equation 29.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta_{\mathcal{B}}(\mathbf{w}(t)) + \zeta\Delta_{\mathcal{B}}(\mathbf{w}(t-1)) \tag{29}$$

where ζ is the momentum parameter.

[1] for hidden layers such an error is estimated based on the backpropagated error.

For autoencoder components, we consider $n = \#S$ and one hidden layer with $n >> H$, as an attempt to detect outliers in the sensor's activities.

The training data have also been divided into batches $\mathcal{B}$, each one with size $|\mathcal{B}|$. At each iteration and for each pattern p in the current batch $\mathcal{B}$, every component of the vector $\mathbf{y}_p^{hidden} = (y_{1p}^{hidden},y_{2p}^{hidden},...,y_{Hp}^{hidden})$ produced by the neural network in the hidden layer, is obtained in the *feedforward pass* as described by Equation 30.

$$y_{jp}^{hidden} = f^e(\mathbf{w}_j^e \cdot \mathbf{x}_p + b_j^e) \tag{30}$$

where b_j^e and $\mathbf{w}_j^e$ are the bias and the weight vector of neuron j in the encoder layer, respectively, and $\mathbf{x}_p$ is the input vector; the output vector $\hat{\mathbf{x}}_p = (\hat{x}_{1p},\hat{x}_{2p},...,\hat{x}_{np})$ has components given by Equation 31.

$$\hat{x}_{kp} = f^d(\mathbf{w}_k^d \cdot \mathbf{y}_p^{hidden} + b_k^d) \tag{31}$$

where b_k^d and $\mathbf{w}_k^d$ are the bias and the weight vector of neuron k in the decoder layer, respectively.

The adjustments are also calculated based on equations 26 to 29. However, they aim to minimize the reconstruction error measured by the loss function $\mathcal{L}$ given by Equation 32.

$$\mathcal{L}_p(\mathbf{x}_p,\hat{\mathbf{x}}_p) = \|\mathbf{x}_p - f^d(\mathbf{W}^d(f^e(\mathbf{W}^e\mathbf{x}_p + \mathbf{b}^e)) + \mathbf{b}^d)\|_2^2 \tag{32}$$

where $\|.\|_2^2$ is the squared error dependent on the Euclidean norm; f^e and f^d are the activation functions of encoder and decoder neurons; $\mathbf{b}^e = (b_1^e,...,b_H^e)$, $\mathbf{b}^d = (b_1^d,...,b_n^d)$, are the vectors encompassing all the biases of encoder and decoder neurons respectively, $\mathbf{W}^e = [\mathbf{w}_1^e,\mathbf{w}_2^e,...,\mathbf{w}_H^e]'$ is the matrix of encoder weights, $\mathbf{W}^d = [\mathbf{w}_1^d,\mathbf{w}_2^d,...,\mathbf{w}_n^d]'$ is the matrix of decoder weights.

Therefore, in the case of autoencoders, we have Equation 27 adapted, resulting in Equation 33.

$$\frac{\partial \mathcal{L}_p}{\partial w_q} = \frac{\partial \mathcal{L}_p}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial w_q} = \frac{\partial \mathcal{L}_p(\mathbf{x}_p,\hat{\mathbf{x}}_p)}{\partial x} f'(u)\text{input}_q = \epsilon_{rp} f'(u)\text{input}_q \tag{33}$$

where $\epsilon_{rp} = \frac{\partial \|\mathbf{x}_p - \hat{\mathbf{x}}_p\|_2^2}{\partial x}$ is the reconstruction error for pattern p.

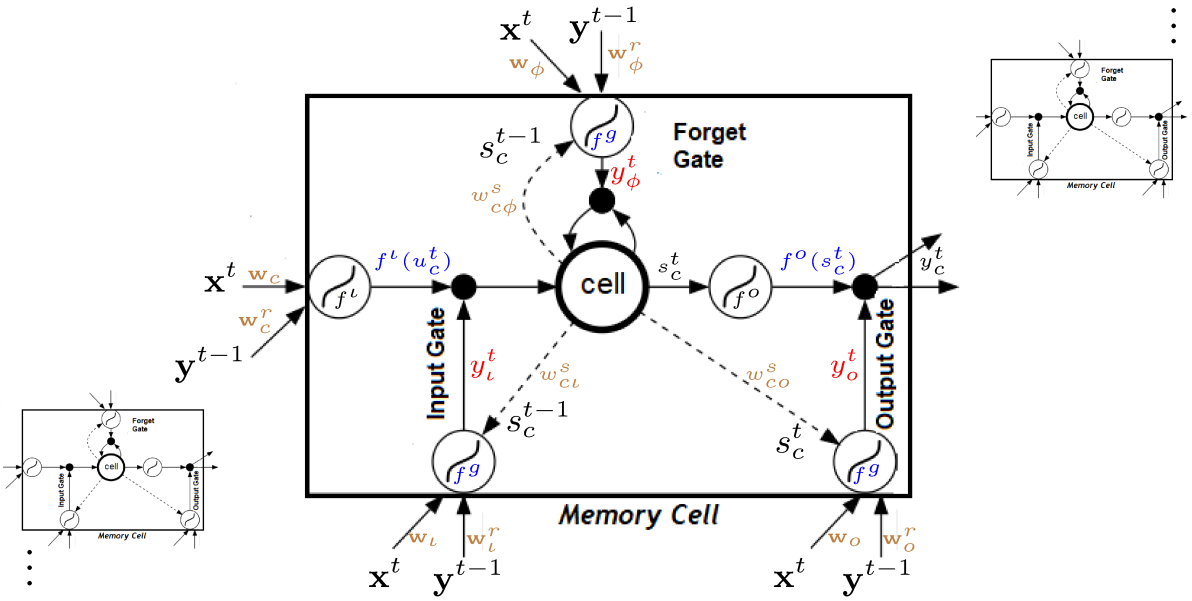
Finally, for bidirectional LSTM learning, we have to consider the particularities of the proposed models. As depicted in Figures 15 and 16, the outputs of every memory cell (squared nodes in the figures) are multiplied by the weights of its output layer (white nodes in the figures), this way, providing the outputs of LSTM models. These outputs become the inputs of an MLP that is positioned in the sequence of the pipeline and used to classify the activity performed by the user in the addressed room. Moreover, different from the remaining approaches, in LSTM models, the time is important and the patterns are collected as sequences with T steps. Therefore, input data encompass a matrix $X_{T \times \#S}$, i.e. a matrix with T rows, and $\#S$ columns, indicating all the $\#S$ sequences (one for each sensor) of size T.

When training the proposed recurrent approaches, first, we have to perform the *feedforward pass* by flowing information from the LSTM input toward the MLP output. It is worth mentioning that bidirectional LSTMs have two hidden layers - Forward and Backward, that must be considered differently in the *feedforward pass*. In the Forward layer, for example, the sequence is considered from $t = 1,...,T$ while in the Backward layer, the sequence is presented in a reverse order $t = T,...,1$. Afterward, we calculate the errors at the MLP output and perform, during the *backward pass*, the classic backpropagation exactly as described in Equations 26 to 29. The classic backpropagation is performed until we reach the LSTM output, as for this model, there are some modifications in the weight updates due to the recurrences.

Aiming to better understand the activation (forward pass) and BPTT gradient calculation (backward pass) of our biLSTM, we need to scrutinize the set of *gates* and activation functions present in the memory cell.

Figure 17 shows a set of three memory cells, zooming one of them. With the **cell** positioned in the main box center, it illustrates the principal components of the emphasized memory block: the cell's input, output, and the three control gates - input, forget, and output.

Figure 17 – Scrutinizing the structure of the memory cells



Source: Adapted from Graves (2012).

Equations 34 and 35 show that the overall input (u_c^t) of the **cell** is similar to the classical RNN and its internal state (s_c^t) depends on the forget (ϕ) and input (ι) gates.

$$u_c^t = \sum_{i=1}^{n} w_{ic} x_i^t + \sum_{j=1}^{H} w_{jc}^r y_j^{t-1} = \mathbf{w}_c.\mathbf{x}^t + \mathbf{w}_c^r.\mathbf{y}^{t-1} \tag{34}$$

$$s_c^t = y_\phi^t s_c^{t-1} + y_\iota^t f^\iota(u_c^t) \tag{35}$$

where $n = \#S$ in our case and H is the set of memory cells (fixed with the same value for both directional layers); f^ι is the input activation function of the **cell**, y_ι^t and y_ϕ^t are the outputs at instant t of *input gate* (ι) and *forget gate* (ϕ), respectively.

Equation (34) indicates that the input of the **cell** at instant t is composed by the current input vector $\mathbf{x}^t = (x_1^t,...,x_n^t)$, weighted by vector $\mathbf{w}_c = (w_{1c},...,w_{nc})$; and also by the outputs of all memory cells at the previous time instant, $\mathbf{y}^{t-1} = (y_1^{t-1},...,y_H^{t-1})$, weighted by the recurrent weight vector $\mathbf{w}_c^r = (w_{1c}^r,...,w_{Hc}^r)$. That is, the cell receives the new inputs ($\mathbf{x}^t$) and, due to network recurrence, it also receives the system's response at the previous instant ($\mathbf{y}^{t-1}$).

As depicted in Figure 17, each *gate* receives three inputs: the current input vector ($\mathbf{x}^t$), the vector of previous outputs of all units in the hidden layer ($\mathbf{y}^{t-1}$), and the state of the cell itself (s^t or s^{t-1} depending on the *gate*).

Assuming, as shown in brown in Figure 17, that $\{\mathbf{w}_\iota,\mathbf{w}_\iota^r,w_{c\iota}^s\}$ is the set of weights associated with the *input gate* (ι), $\{\mathbf{w}_\phi,\mathbf{w}_\phi^r, w_{c\phi}^s\}$ is the set of weights associated with the *forget gate* (ϕ) and $\{\mathbf{w}_o,\mathbf{w}_o^r,w_{co}^s\}$ is the set of weights associated with the *output gate* (o), the formulations for the inputs for all the *gates* are the ones presented in Equations (36) to (38).

$$u_\iota^t = \sum_{i=1}^{n} w_{i\iota} x_i^t + \sum_{j=1}^{H} w_{j\iota}^r y_j^{t-1} + \sum_c w_{c\iota}^s s_c^{t-1} = \mathbf{w}_\iota.\mathbf{x}^t + \mathbf{w}_\iota^r.\mathbf{y}^{t-1} + w_{c\iota}^s s_c^{t-1} \tag{36}$$

$$u_\phi^t = \sum_{i=1}^{n} w_{i\phi} x_i^t + \sum_{j=1}^{H} w_{j\phi}^r y_{j'}^{t-1} + \sum_c w_{c\phi}^s s_c^{t-1} = \mathbf{w}_\phi.\mathbf{x}^t + \mathbf{w}_\phi^r.\mathbf{y}^{t-1} + w_{c\phi}^s s_c^{t-1} \tag{37}$$

$$u_o^t = \sum_{i=1}^{n} w_{io} x_i^t + \sum_{j=1}^{H} w_{jo}^r y_j^{t-1} + \sum_c w_{co}^s s_c^t = \mathbf{w}_o.\mathbf{x}^t + \mathbf{w}_o^r.\mathbf{y}^{t-1} + w_{co}^s s_c^t \tag{38}$$

From equations (36) to (38), we notice that current inputs, previous outputs, and previous states (or the current one, in the case of the *output gate* which has no time delay) influence the flow of information in the network.

The output of each *gate* is its activation function applied to its input, as described by Equations 39 to 41.

$$y_\iota^t = f^g(u_\iota^t) \tag{39}$$

$$y_\phi^t = f^g(u_\phi^t) \tag{40}$$

$$y_o^t = f^g(u_o^t) \tag{41}$$

The memory cell output is given by Equation 42.

$$y_c^t = y_o^t f^o(s_c^t) \tag{42}$$

where f^o is the cell output activation function and y_o^t is the output of *output gate* (o).

Figure 17 and Equations (36) to (38) show that, due to the recurrence, the memory cell output y_c^t becomes the input of another cell in the next time step.

Moreover, as depicted in Figure 7, in the unfolded view of a memory cell, each output pair (from the Forward layer $\mathbf{y}_{cF} = (y_{cF}^1,y_{cF}^2,...,y_{cF}^T)$ and Backward layer $\mathbf{y}_{cB} = (y_{cB}^T,y_{cB}^{T-1},...,y_{cB}^1)$, which are independent of each other) is weighted by w_{ck}^F and w_{ck}^B and further joined in the output layer of the bidirectional LSTM model as described by Equation 43.

$$y_k^t = f(w_{ck}^F.y_{cF}^t + w_{ck}^B.y_{cB}^t) \tag{43}$$

where w_{ck}^F is the weight connecting cell c to the kth output neuron. Then, the LSTM output vector for all K output neurons at instant t is given by Equation 44.

$$\mathbf{y}_{LSTM}^t = \left(y_1^t, y_2^t..., y_K^t\right) \tag{44}$$

In our proposals, LSTM output has the same dimension of the hidden layer, then $K = H$. Moreover, the LSTM ouputs are feedforwarded to the MLP considered in the pipeline of the proposed classifiers to transform a time series prediction into a classification problem.

Figure 18 illustrates the flow of information performed by the backward pass of BPTT algorithm.

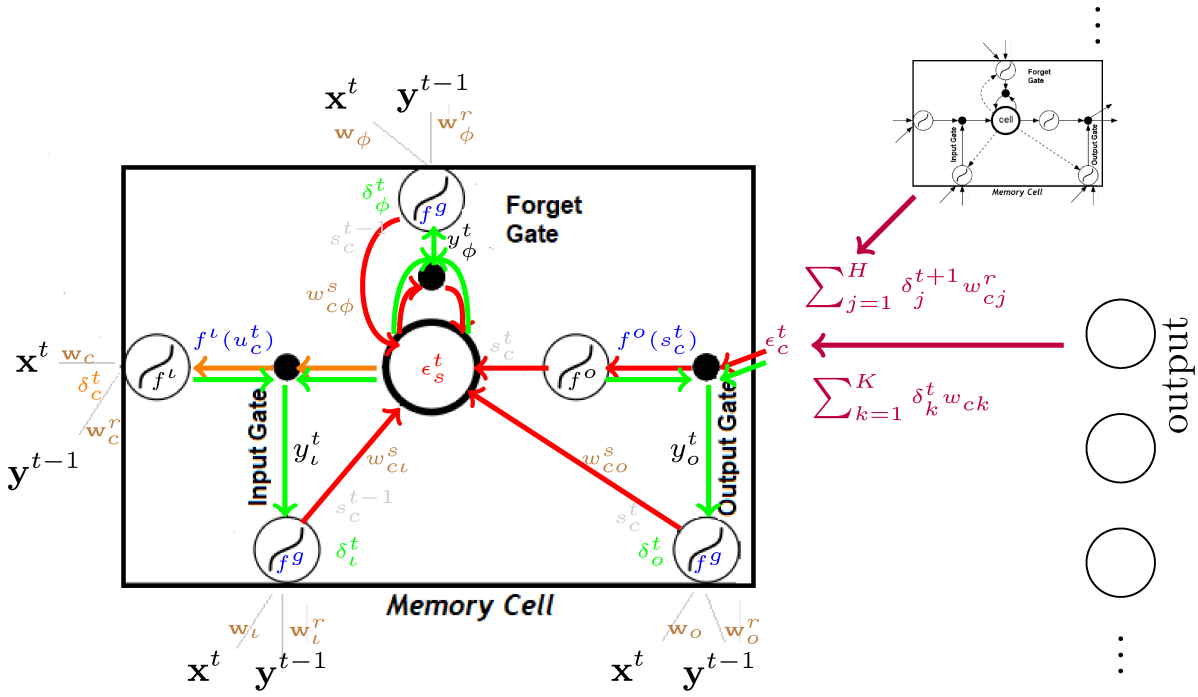
As depicted in Figure 18, BPTT demands the definition of two elements that appear in the adjustments of all the weights in the memory cell: ϵ_c^t, that is the error backpropagated to the cell output (at the right side of the emphasized memory cell) and ϵ_s^t that is the error backpropagated to the cell state (in red at the center of the emphasized memory cell), calculated by Equations 45 and 46, respectively.

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial y_c^t} = \sum_{k=1}^{K} \delta_k^t w_{ck} + \sum_{j=1}^{H} \delta_j^{t+1} w_{cj}^r \tag{45}$$

$$\epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_c^t} = f^{o\prime}(s_c^t) y_o^t \epsilon_c^t + y_\phi^{t+1} \epsilon_s^{t+1} + w_{c\iota}^s \delta_\iota^{t+1} + w_{c\phi}^s \delta_\phi^{t+1} + w_{co}^s \delta_o^t \tag{46}$$

where, w_{ck} is the weight of the connection between c and the k-th output, w_{cj}^r is the recurrent weight of the connection between c and the j-th neuron in the hidden layer (including itself), $w_{c\iota}$ is the weight of the connection between the internal state and the *input gate*, $w_{c\phi}$ is the

Figure 18 – Scrutinizing the flow of backward pass in the memory cells



Source: Adapted from Graves (2012).

weight of the connection between the internal state and the *forget gate* and w_{co} is the connection weight between the internal state and the *output gate*. Equation 46 demonstrates that the error backpropagated (see the red flow in the figure) is influenced by ϵ_c^t, ϵ_s^t and the *gates* themselves.

Then we can compute the elements δ for cells and gates, using Equation 47.

$$\delta_c^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial u_c^t} = y_\iota^t f^{\iota'}(u_c^t)\epsilon_s^t \tag{47}$$

where δ_c^t is the error weighting factor of the internal state unit (depicted in orange at the left size of the emphasized memory cell). From Equation 47, we see that the internal state unit error is related to the internal state itself and to the inputs.

For the *output*, *forget* and *input* gates we calculate the weighting factors using Equations 48 to 50.

$$\delta_o^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial u_o^t} = f^{g\prime}(u_o^t) f^o(s_c^t)\epsilon_c^t \tag{48}$$

$$\delta_\phi^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial u_\phi^t} = f^{g\prime}(u_\phi^t) s_c^{t-1}\epsilon_s^t \tag{49}$$

$$\delta_\iota^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial u_\iota^t} = f^{g\prime}(u_\iota^t) f^\iota(u_c^t)\epsilon_s^t \tag{50}$$

where δ_o, δ_ϕ^t and δ_ι are weighting factors of *input*, *output*, and *forget gate*, respectively.

From Equation 50 we see that the *input gate* error depends on the internal state of the cell and the input scaled by f^ι. Equation 48 indicates that the *output gate* error depends both on the output error and on the internal state scaled by f^o. On the other hand, Equation 49, shows that *forget gate* error depends only on the internal states of the cell. They show, in fact, the error going the opposite way (flow in green in Figure 18) to the flow of information shown in Figure 17.

In the following we describe the general steps of the BPTT algorithm adapted to our proposed approach.

For every epoch e in the total of epochs E

. For every batch $\mathcal{B}$ in the current epoch e

. For every pattern b in the batch $\mathcal{B}$

1. LSTM *feedforward pass*:
   For every **cell** in the hidden layer
   - for $t = 1,..,T$ in Forward layer (F) and $t = T,..,1$ in Backward layer (B)
   - calculate the output of the current **cell** for each pair of elements in F or B

   Calculate LSTM output for pattern p encompassing all the **cell**s

2. MLP *feedforward pass*:
   for all patterns p, calculate the activations for each layer

3. MLP *Backward pass*:
   Calculate the backpropagated errors for each layer
   Calculate the adjustment for each weight in the layer

4. LSTM *backward pass*:
   Calculate the errors and δs for cells and gates
   Calculate the adjustment for each weight due to each pattern b in each memory cell
   For every layer, accumulate the adjustments for $\mathcal{B}$ update the weights

The details of the previous steps are described in the following.

For every pattern b in the batch $\mathcal{B}$ and epoch e

1. LSTM *feedforward pass*:
   For every **cell** in the hidden layer
   - for $t = 1,..,T$ in Forward layer (F) and $t = T,..,1$ in Backward layer (B)
     a) Input Gates: $y_\iota^t = f^g\left(\mathbf{w}_\iota.\mathbf{x}^t + \mathbf{w}_\iota^r.\mathbf{y}^{t-1} + \mathbf{w}_\iota^s.\mathbf{s}^{t-1}\right)$
     b) Forget Gates: $y_\phi^t = f^g\left(\mathbf{w}_\phi.\mathbf{x}^t + \mathbf{w}_\phi^r.\mathbf{y}^{t-1} + \mathbf{w}_\phi^s.\mathbf{s}^{t-1}\right)$
     c) Cells: $s_c^t = y_\phi^t s_c^{t-1} + y_\iota^t f^\iota(\mathbf{w}_c.\mathbf{x}^t + \mathbf{w}_c^r.\mathbf{y}^{t-1})$
     d) Output Gates: $y_o^t = f^g\left(\mathbf{w}_o.\mathbf{x}^t + \mathbf{w}_o^r.\mathbf{y}^{t-1} + \mathbf{w}_o^s.\mathbf{s}^t\right)$
     e) Cells outputs: $y_c^t = y_o^t f^o(s_c^t)$
     f) Store $\mathbf{y}_{cF} = (y_{cF}^1, y_{cF}^2, ..., y_{cF}^T)$ and $\mathbf{y}_{cB} = (y_{cB}^T, y_{cB}^{T-1}, ..., y_{cB}^1)$
   - for every pair p of elements, calculate $y_p^{\textbf{cell}} = f(w_{ck}^F.y_{cF}^p + w_{ck}^B.y_{cB}^p)$

   LSTM output: $\mathbf{y}_p^{LSTM} = \left(y_p^{\textbf{cell}1}, y_p^{\textbf{cell}2}, ..., y_p^{\textbf{cell}H}\right)$

2. MLP *feedforward pass*: for $p = 1,..,T$
   a) Hidden 1: $\mathbf{y}_p^{H1} = f(W^{H1}\mathbf{y}_p^{LSTM} + \mathbf{b}^{H1})$
   b) Hidden 2: $\mathbf{y}_p^{H2} = f(W^{H2}\mathbf{y}_p^{H1} + \mathbf{b}^{H2})$
   c) Output: $\hat{\mathbf{y}}_p = f(W^{out}\mathbf{y}_p^{H2} + \mathbf{b}^{out})$

3. MLP *Backward pass*:
   a) Output: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{jk}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{jk}} = \sum_{p=1}^T \epsilon_{kp} f'(u_{kp}) y_{jp}^{hidd2}$, $k = 1,..,K$
   b) Hidden2: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{hj}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{hj}} = \sum_{p=1}^T \epsilon_{jbkp} f'(u_{jp}) y_{hp}^{hidd1}$, $j = 1,..,H_2$
   c) Hidden1: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{ih}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{ih}} = \sum_{p=1}^T \epsilon_{hbkp} f'(u_{hp}) x_{ip}$, $h = 1,..,H_1$

4. LSTM *backward pass*:
   a) Cells' output errors: ϵ_c^t given by Equation (45).
   b) Output Gates: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{*o}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{*o}} = \sum_{t=1}^T \delta_o^t i_{*o}^t$, δ_o^t by Equation (48)
   c) Cells' state errors: ϵ_s^t given by Equation (46)
   d) Cells: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{*c}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{*c}} = \sum_{t=1}^T \delta_c^t i_{*c}^t$, δ_c^t by Equation (47)
   e) Forget Gates: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{*\phi}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{*\phi}} = \sum_{t=1}^T \delta_\phi^t i_{*\phi}^t$, δ_ϕ^t by Equation (49)
   f) Input Gates: $\nabla\mathcal{L}_b = \left(..., \frac{\partial\mathcal{L}_b}{\partial w_{*\iota}}, ...\right)$, $\frac{\partial\mathcal{L}_b}{\partial w_{*\iota}} = \sum_{t=1}^T \delta_\iota^t i_{*\iota}^t$, δ_ι^t by Equation (50)

   For every layer
   $\mathbf{w}(iter+1) = \mathbf{w}(iter) - \eta\frac{1}{|\mathcal{B}|}(\sum_{b=1}^{|\mathcal{B}|}\nabla\mathcal{L}_b) + \zeta\Delta_{\mathcal{B}}(\mathbf{w}(iter-1))$

3.2.2 Online Learning

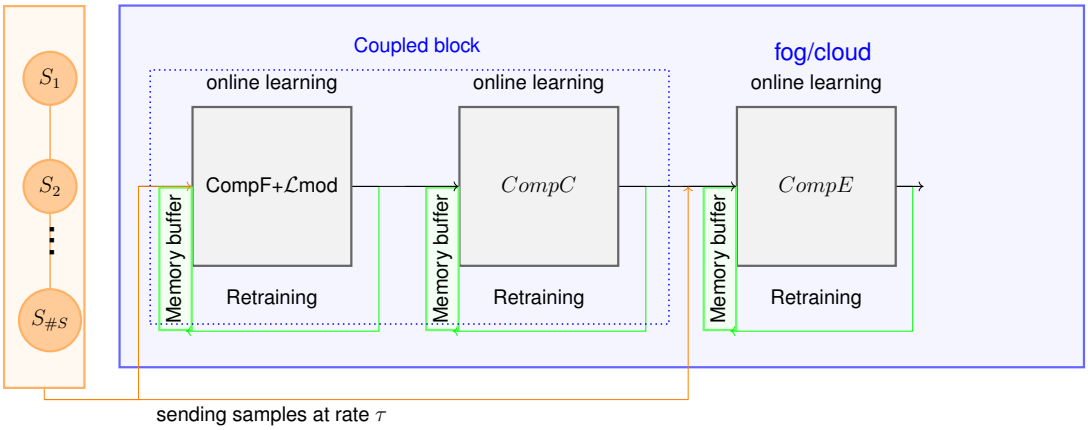
Concerning the online learning performed in the present work, the main differences compared to offline mode comprise:

1. the role played by each pattern - it can be used for both, first testing and then training, since all data matter;
2. the concept of epoch or iteration that completely changes in the performed online learning;
3. the two novel control parameters:
    - the rate τ in which data are sent from sensors to cloud/fog systems, where the shallow components are built in;
    - the budget of memory M considered for each component, which might be less restrictive in the cloud compared to fog computing.

Besides these two control parameters, there are two additional factors that impact the system's performance, nevertheless, they are not under control. First, there is a delay that naturally occurs in communication systems, particularly for cloud computing, where communication occurs mainly through the Internet. Second, there is the actual processor capacity of systems that operate with multiple users in cloud computing. Although not directly controlled, it is usually assumed superior to the one available for fog computing.

Figure 19 shows an overview of the online learning of a complete pipeline built in and running on fog or cloud computing. The figure could represent any proposed model, where the proposed Model1 model would encompass only CompE, Model2 would not include CompF+$\mathcal{L}$mod and the hybrid models would be composed of all components depicted in the figure.

Figure 19 – Online learning for a complete pipeline CompF + $\mathcal{L}$mod $\rightarrow$ CompC $\rightarrow$ CompE that could represent the Model3 or Model4 proposed models.



Source: Author

In the case of online learning, the batch size is $|\mathcal{B}| = 1$, which means the Stochastic gradient descent, where the weights are updated for each pattern p at every iteration t. Then, for the backpropagation with momentum, we need to use Equation 51.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta_p(\mathbf{w}(t)) + \zeta\Delta_p(\mathbf{w}(t-1)) \tag{51}$$

where the weight adjustment element is given by Equation 52.

$$\Delta_p(\mathbf{w}(t)) = -\eta\nabla\mathcal{L}_p \tag{52}$$

Training time is an important criterion as the system runs on small values of sample time. Thus, differently from offline learning, when adapting weights/bias in an online way, the proposals use each sample p as soon as it arrives and past experience can be completely lost since data streaming can provide new data when there is not enough memory to store all patterns. As in the offline case, some components could be trained separately - e.g. the coupled block (CompF+$\mathcal{L}$mod$\rightarrow$CompC) independently from the CompE component - but it is necessary to synchronize them in a way that whenever the output of the coupled block is 'on' it enables the direct communication between the inputs (i.e. sensors information) and the CompE component as both must process the same input. Due to their longer pipelines, synchronization is more critical in deeper models (Model3 and Model4).

4 EXPERIMENTS AND RESULTS

This chapter describes the experiments conducted to evaluate the proposed approach and presents the corresponding results. It begins by describing the experimental design including the problem definition (Section 4.1), and the setup phase (Section 4.2) for topology and training parameters. Next, the section presents the results of the experiments in Section 4.3 divided into results from offline learning and results from online learning.

4.1 The addressed problem

A classification problem can be formally defined as the task of estimating the output label $y \in Y = \{c_1, c_2, ...\}$ of an n-dimensional input vector $\mathbf{x}$. Most of the time, particularly in the ANN context, input variables have to be real-valued, i.e., $\mathbf{x} \in \mathbb{R}^n$. In the present work, we address eight different classification problems each one described as the problem of classifying the activities that occur in a particular room of a smart home (see Figure 20 as an example), based on the information provided by sensors distributed in the room.

Figure 20 – A possible perspective of the addressed smart home

Source: This image has been generated by Artificial Intelligence.

In this work, we use Orange4Home as the dataset for the experiments, built by Cumin *et al.* (2017). This dataset brings the readings of sensors and labeled activities, recorded in an instrumented smart house during 4 weeks, 5 days per week and 8 hours per day. The house was

inhabited by a single occupant and this person was performing routine activities in all rooms of the house.

Each room of the house had a different set of sensors, and there were also global sensors, that could be shared by all rooms. The sensors could generate binary, real, integer or categorical data. Examples of these sensors are: presence, door, luminosity, CO_2 and noise sensors, switches, voltage, power and water consumption, appliance working modes. As for the global sensors, they measured values that applied to the entire house, such as weather conditions, time and total power consumption.

The activity labels in the dataset were generated by the occupant of the house. They would annotate the start and end time, as well as the activity category during the day. Since there was a single person in the house, there can be only one label at any time, and only one room has activity, while the other rooms receive a virtual 'no activity' label. There is a predefined list of possible activities, and not all of them can be executed in all rooms.

The following list summarizes the number of sensors ($\#S$) in each room (total of sensors, combining room-specific and global sensors), and the total of classes ($\#C$), i.e., the total of possible activities in each room performed by the smart home user:

1. **Entrance** with $\#S$=55 and $\#C$=2: $\mathbf{x} \in \mathbb{R}^{55}$, $Y = \{c_1,c_2\}$, where c_1 is *entering the house* and c_2 is *leaving the house*;
2. **Staircase** with $\#S$=51 and $\#C$=2: $\mathbf{x} \in \mathbb{R}^{51}$, $Y = \{c_1,c_2\}$; c_1 is *going upstairs* and c_2 is *going downstairs*;
3. **Bathroom** with $\#S$=68 and $\#C$=4: $\mathbf{x} \in \mathbb{R}^{68}$, $Y = \{c_1,c_2,c_3,c_4\}$; c_1 is *showering*, c_2 is *using the sink*, c_3 is *using the toilet* and c_4 is *cleaning*;
4. **Livingroom** with $\#S$=80 and $\#C$=4: $\mathbf{x} \in \mathbb{R}^{80}$, $Y = \{c_1,c_2,c_3,c_4\}$; c_1 is *watching TV*, c_2 is *using the computer*, c_3 is *eating* and c_4 is *cleaning*;
5. **Toilet** with $\#S$=47 and $\#C$=1: $\mathbf{x} \in \mathbb{R}^{47}$, $Y = \{c_1\}$; c_1 is *using the toilet*;
6. **Office** with $\#S$=62 and $\#C$=3: $\mathbf{x} \in \mathbb{R}^{62}$, $Y = \{c_1,c_2,c_3\}$ where c_1 is *watching TV*, c_2 is *using the computer* and c_3 is *cleaning*;
7. **Kitchen** with $\#S$=94 and $\#C$=4: $\mathbf{x} \in \mathbb{R}^{94}$, $Y = \{c_1,c_2,c_3,c_4\}$; c_1 is *preparing food*, c_2 is *cooking*, c_3 is *washing the dishes* and c_4 is *cleaning*;
8. **Bedroom** with $\#S$=76 and $\#C$=4: $\mathbf{x} \in \mathbb{R}^{76}$, $Y = \{c_1,c_2,c_3,c_4\}$; c_1 is *cleaning*, c_2 is *dressing*, c_3 is *reading* and c_4 is *napping*.

As mentioned before, the Orange4Home dataset has different types of values. To be able to feed this data into the neural networks, it was necessary to preprocess the data. All values were normalized to a scale of real numbers ranging from 0 to 1, i.e., binary numbers

were converted to 0 or 1, integer and real numbers were rescaled, and categorical values were assigned to evenly spaced numbers between 0 and 1. As for the activity classes, they were converted to a one-hot encoding.

In this smart home application, the number of constrained connected devices is high, and the volume of data generated is also high. First, we consider a solution running offline in the Cloud, which presents as its main characteristics the large geographic and logical distance between end devices and servers. Next, we select the best approach to also run in the Fog under online learning aiming to evaluate its latency and response time.

4.2 Setup for the experiments

In this section, we describe the setup for the experiments. First, we separate data from each room of the addressed smart home and then we describe the parameters of the neural models and training setup.

4.2.1 Smart home dataset

Table 3 shows data distribution according to the rooms: number of sensors #S, number of activities or classes #C (excluding NA class), the total of samples, and the percentage of samples that represent an output class other than NA (non-activity), of each room separately.

Table 3 – A summary of Dataset used in the experiments

Room	#S	#C	Total of Samples	Activity	Difficulty Average Rank
Kitchen	94 (1)	4 (1)	69417 (3)	10.46% (4)	2.3
Livingroom	80 (2)	4 (1)	120598 (1)	18.83% (5)	2.3
Bedroom	76 (3)	4 (1)	45405 (4)	19.46% (5)	3.3
Bathroom	68 (4)	4 (1)	36860 (5)	18.82% (5)	3.8
Office	62 (5)	3 (3)	104290 (2)	84.24% (6)	3.8
Entrance	55 (6)	2 (3)	30216 (7)	2.34% (2)	4.5
Staircase	51 (7)	2 (3)	33629 (6)	2.83% (3)	4.8
Toilet	47 (8)	1 (4)	28947 (8)	0.33% (1)	6.7

The numbers in parenthesis represent the classification regarding the challenge posed by each room. Lower numbers indicate higher difficulty (higher number of sensors, classes, samples, and more unbalanced classes - rare activities). The last column shows the average rank.

4.2.2 Neural model parameters for the topologies

The experiments conducted in the present work compare the six proposed approaches described in Section 3.1 (Model1, Model2, Model3, Model4, Model5 and Model6), whose parameters are shown in Table 4.

Table 4 – Model Parameters

Model	Component	parameter	Value
Model1	CompA	number of inputs	#S
		number of outputs	$\#C + 1$
		hidden layers [neurons per layer]	2 [10, 5]
Model2	CompB	number of inputs	#S
		number of outputs	2
		hidden layers [neurons per layer]	2 [10, 5]
	CompD	number of inputs	#S
		number of outputs	#C
		hidden layers [neurons per layer]	2 [10, 5]
Model3	CompF	number of inputs	#S
		number of outputs	#S
		hidden layers [neurons per layer]	1 [3]
	CompB	number of inputs	1
		number of outputs	2
		hidden layers [neurons per layer]	2 [10, 5]
	CompD	number of inputs	#S
		number of outputs	#C
		hidden layers [neurons per layer]	2 [10, 5]
Model4	CompF	number of inputs	#S
		number of outputs	#S
		hidden layers [neurons per layer]	1 [3]
	CompC	number of inputs	1
		number of outputs	2
		hidden layers [neurons per layer]	2 [100, 25]
	CompE	number of inputs	#S
		number of outputs	#C
		hidden layers [neurons per layer]	2 [200, 50]
Model5	CompG	number of inputs	#S
		number of outputs	#C + 1
		hidden layers [neurons per layer]	3 [10, 200, 100]
Model6	CompH	number of inputs	#S
		number of outputs	2
		hidden layers [neurons per layer]	3 [10, 200, 100]
	CompI	number of inputs	#S
		number of outputs	#C
		hidden layers [neurons per layer]	3 [10, 200, 100]

4.2.3 Setup parameters for training

The learning process can occur in two different modes: offline and online. In the online mode, neural network models can be built in two computing systems: cloud and fog. First, we deploy each model to learn, in an offline way, the behavior of the smart home user in the eight different rooms described in Section 4.1. Then we evaluate, in a particular room (kitchen), the performance of Model4 model when receiving streaming data in an online learning scheme.

In the offline learning mode, we divide the total of samples in each room, i.e. sensor measures and the respective target output (one activity among the #C possible ones), into three different groups. Therefore in the offline learning we have $\{(\mathbf{x},y)\}$ for a particular room divided into training $\{(\mathbf{x},y)\}_{tr} = 64\%$, validation $\{(\mathbf{x},y)\}_{vl} = 16\%$ and testing $\{(\mathbf{x},y)\}_{ts} = 20\%$ groups of data. The set $\{(\mathbf{x},y)\}_{ts}$ changes according to each fold of a 5-fold cross-validation process performed in the room.

In online learning, there is no such a distinction, and the whole dataset for each room encompasses individual samples $(\mathbf{x},y)_{\tau t}$ that are sent to the fog/cloud system at a rate τ and received and processed at time t by the addressed model. Such a sample plays the role of a test sample first and then it is (re)trained as many times as the memory buffer size M allows it to.

Besides initialization of internal model parameters (weights and bias of every layer in the neural models), it is also necessary to set up the parameters that control the learning process. Table 5 describes the main *learning parameters* as well as their values considered in the experiments. The table separates parameters whose values are common to both learning modes (Opt, η, β_1, β_1), from others that are exclusive for each mode (E and T - exclusive of offline learning - and M - exclusive of online mode) or those that are common but have different values depending on the learning mode, i.e. parameters $\mathcal{L}$, $\mathcal{B}$ and Stp.

For online learning experiments, two environments are set up: one with higher processing capacity, to simulate a cloud platform; and another with less computing power, to simulate a fog system. The first environment runs on a computer with Intel(R) Core(TM) i7-8565U processor, having 8 CPUs @ 1.80GHz, to receive the streaming data and train the NNs. A Raspberry Pi 4 Model B, with a Cortex-A72 processor and 4 CPUs @ 1.5GHz, is used as the fog platform. To also take into account the traffic of data in the experiments, the cloud scenario has data being sent from US East (N. Virginia) region to a local computer in Brazil whilst in the fog scenario, data are exchanged between two computers within a local network.

4.3 Results

The results in this section are organized based on the research questions, aiming to discuss the significance of the findings.

Table 5 – Offline and Online Learning Parameters

Name/symbol	Description		
Adam Opt	Adaptive Moment Estimation (Adam) is an optimizer that combines two other approaches i) AdaGrad and ii) RMSProp, using moving averages of the first and second moments of the gradient (KINGMA; BA, 2014);		
learning rate η	controls how the weights are updated during the training;		
decay rates β_1, β_2	decay rates for the first and second moment estimates;		
loss function $\mathcal{L}$	measures how high is the (reconstruction) error		
epochs E	the number of times the learning algorithm will work through the entire training dataset;		
batch size $\|\mathcal{B}\|$	refers to the number of training samples propagated through the network before each weight update takes place;		
stop condition Stp	the rule established to finish the learning process;		
transmission rate τ	samples per second transmitted by sensors in the smart home;		
memory buffer M	size of the buffer of each component in the proposed models;		
input timesteps T	number of steps (inputs) back in time to be used along with the current input.		
Symbol (learning mode)	Autoencoder	MLP	LSTM
Opt (online & offline)	Adam	Adam	Adam
η (online & offline)	0.01	0.01	0.001
β_1,β_2 (online & offline)	β_1=0.9, β_2=0.999	β_1=0.9, β_2=0.999	β_1=0.9, β_2=0.999
$\mathcal{L}$ (offline)	$\mathcal{L}_\xi$: weighted Squared Error	$\mathcal{L}$: Cross Entropy	$\mathcal{L}$: Categorical Cross Entropy
$\mathcal{L}$ (online)	$\mathcal{L}$: Squared Error		
$\|\mathcal{B}\|$ (offline)	10	25	100
$\|\mathcal{B}\|$ (online)	1	1	N/A
Stp (offline)	achieve $E = 1000$ epochs	achieve $E = 200$ epochs	achieve $E = 100$ epochs
Stp (online)	Total of tested samples (4000)	Total tested of samples (4000)	N/A
τ (online)	{high,med,low}	{high,med,low}	N/A
M (online)	cloud={low,high} fog={low,high}	cloud={low,high} fog={low,high}	N/A
T (offline)	N/A	N/A	10

4.3.1 Offline learning results

In the first phase of the experiments, we assume that all the proposed approaches have been trained offline considering each room individually. Having the results with the highest average highlighted, Table 6 shows the performance (F-score average and standard deviation) from a 5-fold cross-validation process, where each non-overlapping set $\{(\mathbf{x},y)\}_{ts}$ is fixed as a fold test once in the five repetitions.

From Table 6, we notice that hybrid models perform quite differently depending on the size of their components (small or large). MLP performance also changes due to the hierarchy.

Table 6 – Offline: models' average F-score for each room

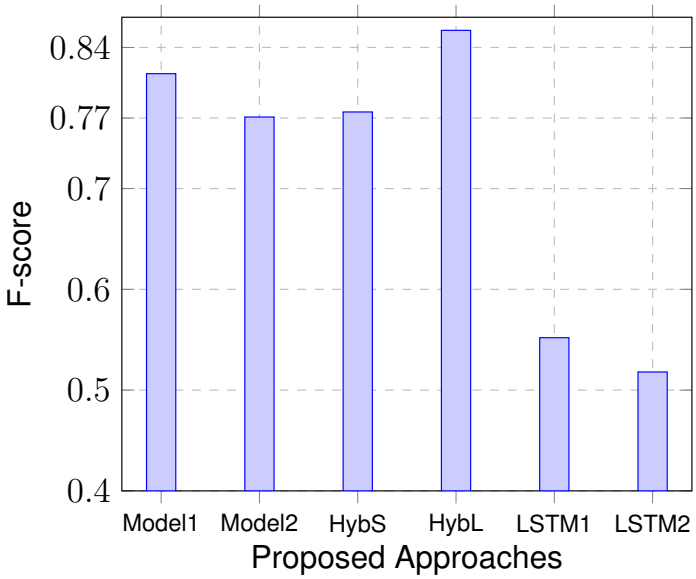
Room/Model	Fscore from 5-fold
Kitchen	Avg ± stdv
Model1	0.626 ± 0.258
Model2	0.791 ± 0.040
Model3	0.715 ± 0.245
Model4	0.884 ± 0.017
Model5	-
Model6	-
Livingroom	Avg ± stdv
Model1	0.910 ± 0.012
Model2	0.719 ± 0.286
Model3	0.551 ± 0.151
Model4	0.614 ± 0.101
Model5	0.502 ± 0.026
Model6	0.491 ± $2.6{\cdot}10^{-3}$
Bedroom	Avg ± stdv
Model1	0.795 ± 0.345
Model2	0.921 ± 0.050
Model3	-
Model4	-
Model5	-
Model6	-
Bathroom	Avg ± stdv
Model1	0.858 ± 0.018
Model2	0.695 ± 0.129
Model3	0.682 ± 0.228
Model4	0.911 ± 0.021
Model5	0.603 ± 0.175
Model6	0.583 ± 0.159

Room/Model	Fscore from 5-fold
Office	Avg ± stdv
Model1	0.768 ± 0.318
Model2	0.591 ± 0.216
Model3	0.857 ± 0.204
Model4	0.963 ± 0.021
Model5	0.710 ± 0.299
Model6	-
Entrance	Avg ± stdv
Model1	0.906 ± 0.006
Model2	0.924 ± 0.031
Model3	0.915 ± 0.011
Model4	0.908 ± 0.015
Model5	0.499 ± $3.2{\cdot}10^{-5}$
Model6	0.499 ± $3.2{\cdot}10^{-5}$
Staircase	Avg ± stdv
Model1	0.831 ± 0.009
Model2	0.858 ± 0.016
Model3	0.821 ± 0.021
Model4	0.854 ± 0.028
Model5	0.499 ± $6.2{\cdot}10^{-5}$
Model6	0.499 ± $6.2{\cdot}10^{-5}$
Toilet	Avg ± stdv
Model1	0.800 ± 0.171
Model2	0.817 ± 0.073
Model3	0.889 ± 0.025
Model4	0.865 ± 0.014
Model5	0.499 ± $2.2{\cdot}10^{-5}$
Model6	-

Considering the rooms' characteristics shown in Table 3, including the rank of difficulty posed by each one, we see from Table 6 that all approaches are performing well (F-score > 0.8) for simpler rooms, i.e., the last three in the right side of the table. The exceptions are the LSTM-based approaches that did not achieve good results for any of the experiments. Model4, in contrast with Model3, has the highest average performance for the most difficult problems: the rooms with a high number of sensors/classes (kitchen, bathroom, and office). Both, Model3 and Model4, have good performance in the room with the lowest percentage of activities (Toilet), highlighting the outlier (rare activity) detection capability of their autoencoders. Finally, the pure MLP model (Model1) is better for the room with the highest number of samples (Livingroom) and its hierarchical version (Model2) is performing well for two simple cases (Entrance and Staircase), and one difficult (Bedroom), in which neither of the hybrid approaches have converged. In the case of Model4, it is important to point out that, it is among the best ones for the five most difficult addressed rooms, and except for the Livingroom and Bedroom, it achieved a higher average performance than the baseline Model1 in all the addressed problems.

Aiming to investigate an overall performance for offline learning, we performed a comparison in terms of average performance for all the rooms where the approaches have converged. Figure 21 shows the overall average performance of all proposed approaches.

Figure 21 – Models' overall performance (F-score average for all rooms)



Source: Author.

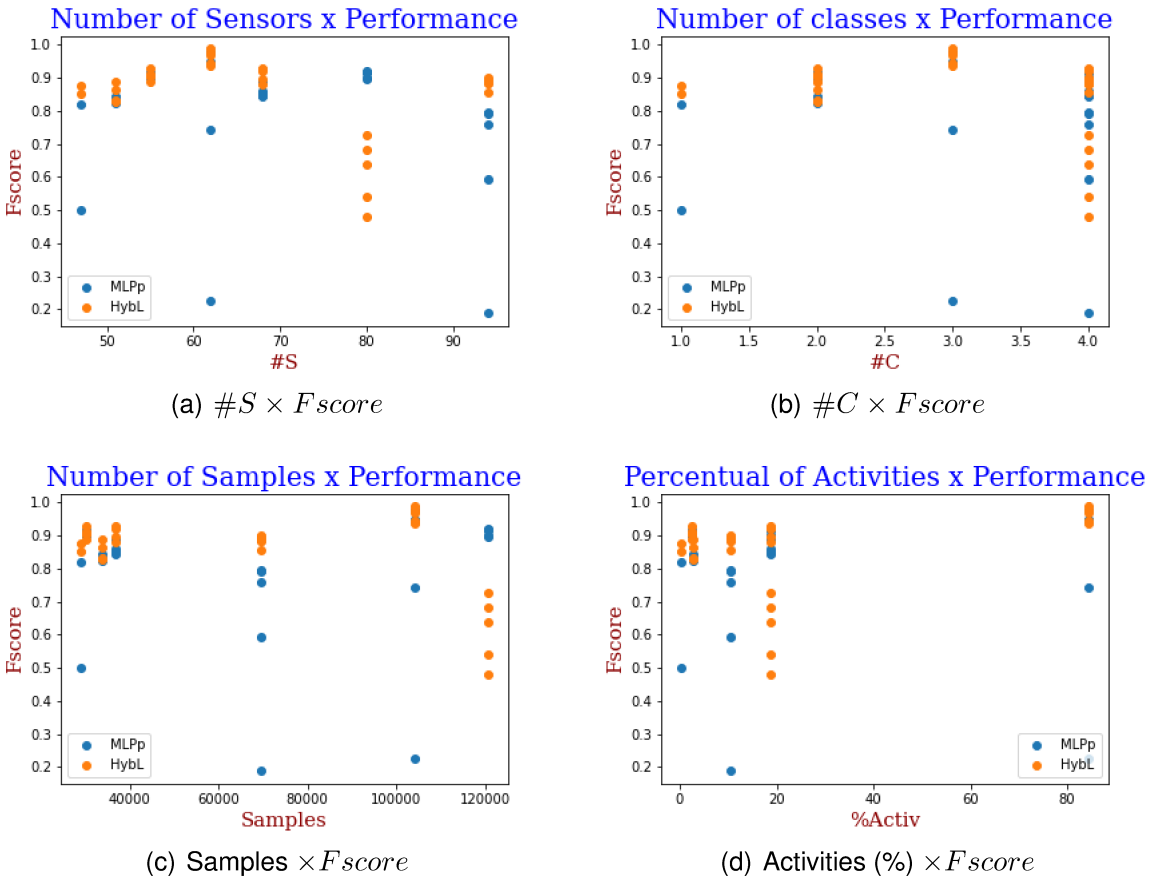
From the analysis of Figure 21, we can conclude that all the approaches based on shallow components (Model1, Model2, Model3 and Model4) outperform those based on LSTM. One possible justification for this poor performance could be some choices made for the biLSTM + MLP models. In the case of the activation functions, we chose tanh for the gates and softmax for the hidden layers of the MLP. With these functions the gates might not be properly blocking new information to flow through the network, and softmax could be forcing only one neuron to be active in each hidden layer. On the other hand, for the remaining models, the performance was good and the approaches with the best overall performance are the simplest and the largest ones.

Aiming to perform a deeper comparison among the approaches with the highest overall average performance (Model4 and Model1), we proceeded by investigating possible correlations between the performance of each model and the characteristics of each addressed room.

Figure 22 presents the scatter chart for the performance (F-score) regarding the different characteristics of each addressed room: number of sensors ($\#S$), number of classes ($\#C$), number of samples and percentage of activities in the input patterns.

Table 7 presents the Pearson correlation for each graphic depicted in Figure 22.

Figure 22 – Scatter chart for rooms' characteristics and model performance



(a) $\#S \times Fscore$ (b) $\#C \times Fscore$

(c) Samples $\times Fscore$ (d) Activities (%) $\times Fscore$

Source: Author

Table 7 – Pearson correlation index between rooms' characteristics and model performance

Model	#S × Fscore	#C × Fscore
Model1	-0.210	-0.025
Model4	-0.318	-0.291
Model	**Samples × Fscore**	**% Activities × Fscore**
Model1	-0.000	-0.060
Model4	**-0.477**	0.287

It is worth mentioning that all the performed tests present a weak linear correlation (values in range [-0.5, 0.5]); evaluating other types of correlation is out of the scope of the present work. Values tending to a moderate negative correlation (in bold) are observed only in the number of samples for the Model4 model of Figure 22 (c). In this case, the hybrid model is slightly and negatively affected by the increase in the number of training patterns (Samples). As the Model1 model presents no high negative correlation value, we can conclude that it is the most competitive approach for offline learning.

To run on fog/cloud systems while learning online the behavior of the smart home user through streaming data received from the sensors, we chose the Kitchen, i.e., the room classified

as the most challenging one. For this, Model4 was chosen as the evaluated approach since it achieved the best results for this room.

4.3.2 Online learning results: fog *versus* cloud computing

In the online mode we consider the Model4 proposed model running either on cloud or fog computing (see Figures 1 and 19 for more details). Aiming to compare both systems under different conditions of sensors and NN components, we consider six combinations of memory buffer sizes $M \in \{1000,100,10\}$ and transmission rates $\tau \in \{2.5,1.25,0.83\}$ (samples per second).

Table 8 shows average times (in seconds) spent by different topology setups of Model4 to process each test sample. As expected, the average time to process each sample in the

Table 8 – Online: results for the different combinations of setups

Setup Combinations	τ (smp/s)	cloud M	cloud time(s)	fog M	fog time(s)
High τ with Low M	2.5	100	1.43	10	2.43
High τ with High M	2.5	1000	1.14	100	2.34
Med τ with Low M	1.25	100	1.22	10	2.36
Med τ with High M	1.25	1000	1.10	100	2.27
Low τ with Low M	0.83	100	1.20	10	2.31
Low τ with High M	0.83	1000	1.31	100	2.28

fog system is higher (almost twice) than that in the Cloud. The time seems mainly affected by processing capacity, which is quite lower in the fog system. However, the fog system seems to be more robust to changes in M and τ than the Cloud. Whereas in the Cloud, when memory increases 10 times, time relative gains are $\{\cong 20\%, \cong 10\%, \cong -6\%\}$ for the three τ rates, in the Fog, the corresponding relative gains are smaller, and almost constant $\{\cong 4\%, \cong 4\%, \cong 1\%\}$ for all τ.

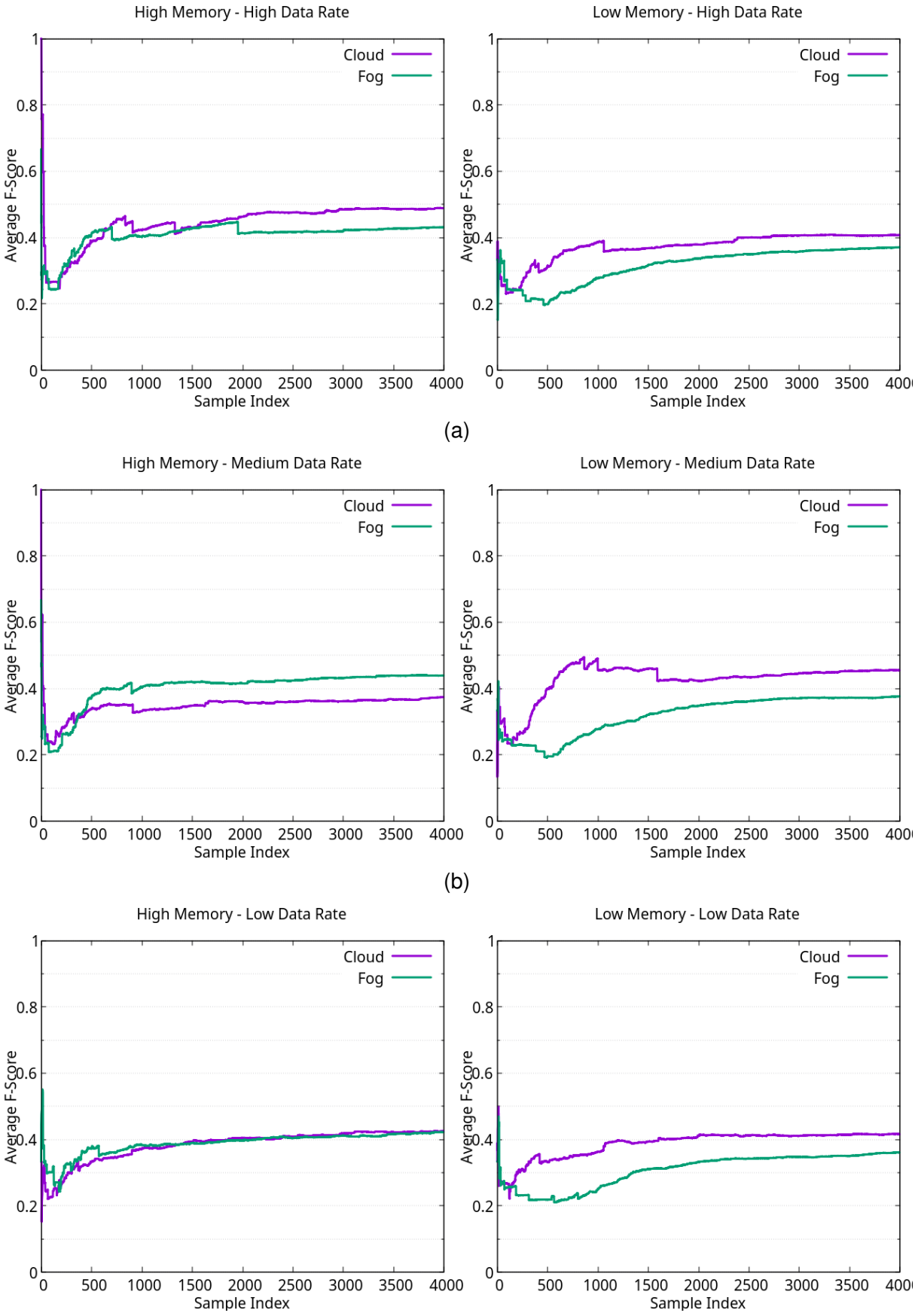
Figure 23 presents the curves for the average F-Score *versus* total of samples presented so far. They result from three runs, each one with Stp=4000 testing samples, as well as different weight/bias initialization and order of samples presentation. From Figure 23, we notice that in general fog performs quite similarly to the Cloud. The exceptions occur for Medium τ, where High M provides the best fog's performance and Low M, the worst one.

Figure 24 shows, in a log-scale to enable cloud $\times$ fog comparison, the average number of times each sample is retrained. It is important to highlight that every streaming sample is first tested and then (re)trained as many times as possible until the memory buffer M becomes full and the oldest samples start getting discarded. The gap (most visible for larger M sizes) in the average retraining curves is due to the time taken to fill the memory buffer, after which, discarding process starts and the number of retraining per sample can be computed. The initial peak on the graphs, indicating a higher number of repetitions, occurs because the first training

rounds take less time to complete since they have fewer examples to go through. Subsequently to this ‘warm-up’ period, the curves tend to become stable, and the stabilization F-score value represents the bottleneck of processor capacity for that hardware. As expected, in all scenarios the cloud environment is capable of training each sample many more times (4 to 10 times), because of its more powerful processor and larger memory.

Taking all results into account, we can notice that, even though ANNs executing in the fog take a larger time to process samples and to converge, they stabilize at a similar performance level. That means the amount of retraining has a low impact on the final result over time, i.e., less expensive fog hardware could be used as a replacement or as a support to a cloud solution and could still achieve comparable performance.

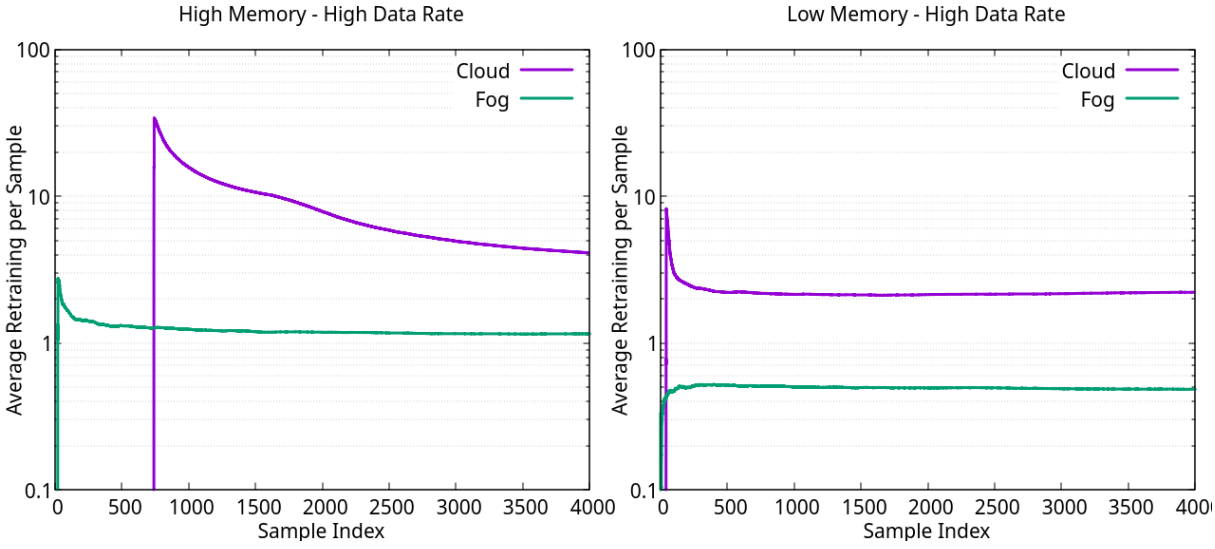
Figure 23 – Average F-Score vs. Tested Samples for online learning



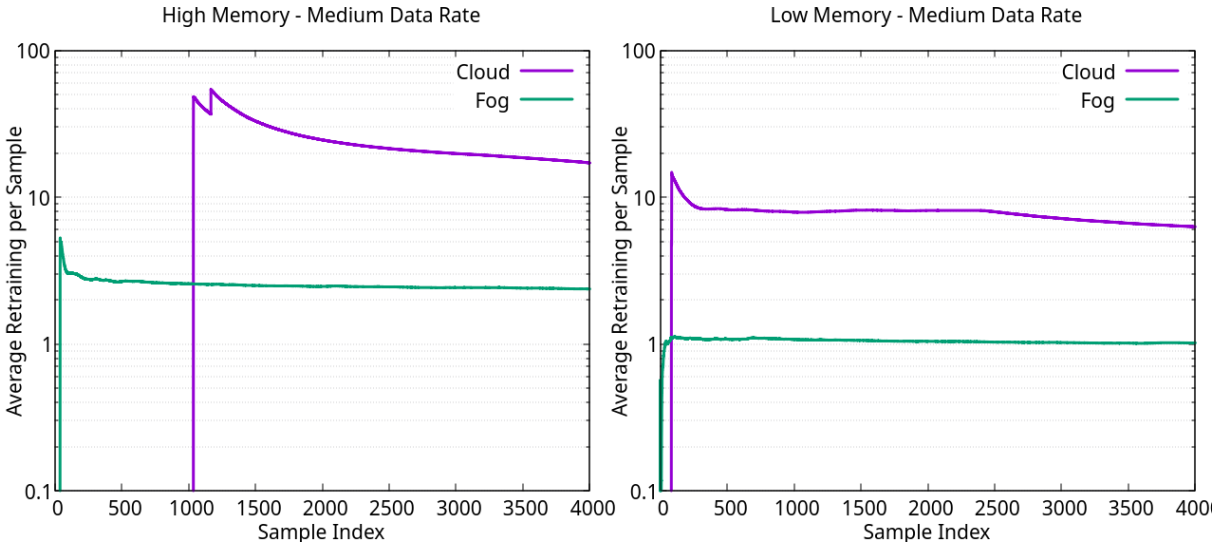
High and Low Memory sizes *versus* (a) High, (b) Medium, and (c) Low Data Rate.

Source: Author

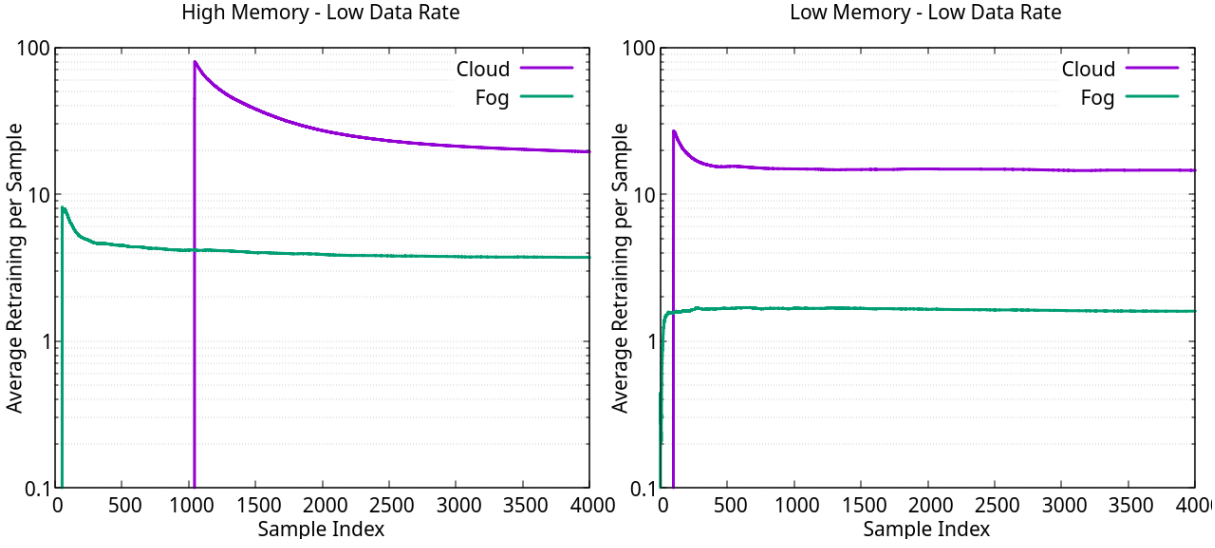
Figure 24 – Average Retraining per Sample vs. Tested Samples for online learning



(a)



(b)



(c)

High and Low Memory sizes *versus* (a) High, (b) Medium, and (c) Low Data Rate.

Source: Author

5 CONCLUSIONS

This work has investigated artificial neural network models under offline and online learning in the context of smart homes and fog/cloud computing. First, it has evaluated six different models trained offline to solve the addressed problems of classifying activities in each room of a smart home. Then the work has explored different configurations of streaming data in online learning and one particular model (the one with a good performance in offline mode) running in a specific room (the one considered the most challenging room) using fog/cloud computing resources.

Based on the achieved results, we note that in offline learning, adding an extra classification level to the neural architecture to split samples into activity/non-activity, before actually performing the activity classification, can improve the results only when it is coupled with an autoencoder and has a pipeline with a certain level of complexity. In our case, setting 10 and 5 neurons in the first and second hidden layers, respectively, was considered insufficient and the performance increased only when we set more than 100 neurons in the first hidden layer and 25 neurons in the second one. Otherwise, a simple multilayer perceptron is capable of performing the whole task. We noticed that the autoencoder is capable of learning, in an unsupervised way, the characteristics of activities that are addressed as “outliers” in the model. Another observation was that the deep recursive models failed to perform like classifiers when coupled with multilayer perceptions. After scrutinizing the functioning of our bidirectional long short time memory model we notice that it might be due to some particular configurations adopted for the gates’ and MLPs’ hidden layer activation functions. However, more investigation would be necessary to confirm this hypothesis. Moreover, the simplest model appeared as quite competitive, showing that the increase in complexity did not result in a significant performance improvement. In online learning, the fog proposed topology was capable of performing comparably with one using cloud computing even if it uses much less retrained samples. Yet the addressed model performed worse than in the offline learning way - what in fact was expected - when compared with the cloud results, the performance was not significantly affected by the lower computational resources available in the fog environment.

In future work, we intend to consider online learning for all models and rooms and include other online learning algorithms. Another topic for future studies is the modification of the neural models to solve time series prediction problems instead of classification ones, as already addressed in the current work. In this case, the LSTM model can benefit from the new context and suitable configuration. This, besides opening room for solving predictive control problems, could help eliminate the dependency on human feedback when determining the ground truth activity class during the training phase. Such a task can be quite challenging, particularly for online learning.

REFERENCES

ALSAMHI, S. H. *et al.* Predictive estimation of optimal signal strength from drones over iot frameworks in smart cities. **IEEE Transactions on Mobile Computing**, p. 1–1, 2021.

ALZUBAIDI, L. *et al.* **Review of deep learning: concepts, CNN architectures, challenges, applications, future directions.** 2021. 1-74 p.

BALLARD, D. H. Modular learning in neural networks. *In*: **Aaai**. [*S.l.*: *s.n.*], 1987. v. 647, p. 279–284.

BANGARU, S. S. *et al.* Ann-based automated scaffold builder activity recognition through wearable emg and imu sensors. **Automation in Construction**, Elsevier, v. 126, p. 103653, 2021.

BASODI, S. *et al.* Gradient amplification: An efficient way to train deep neural networks. **Big Data Mining and Analytics**, TUP, v. 3, n. 3, p. 196–207, 2020.

BESSA, R. J.; MIRANDA, V.; GAMA, J. Entropy and correntropy against minimum square error in offline and online three-day ahead wind power forecasting. **IEEE Transactions on Power Systems**, v. 24, n. 4, p. 1657–1666, 2009.

BONOMI, F. *et al.* Fog computing and its role in the internet of things. *In*: **Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12**. [*S.l.*: *s.n.*], 2012. p. 13.

CISCO. Fog computing and the internet of things: Extend the cloud to where the things are. *In*: **White Paper**. [*S.l.*: *s.n.*], 2015. v. 2015, p. 1–6.

COATES, A.; NG, A.; LEE, H. An analysis of single-layer networks in unsupervised feature learning. *In*: **Proceedings of the fourteenth international conference on artificial intelligence and statistics**. [*S.l.*: *s.n.*], 2011. p. 215–223.

CUMIN, J. *et al.* A dataset of routine daily activities in an instrumented home. *In*: SPRINGER. **Ubiquitous Computing and Ambient Intelligence: 11th International Conference, UCAmI 2017, Philadelphia, PA, USA, November 7–10, 2017, Proceedings**. [*S.l.*], 2017. p. 413–425.

DHANARAJ, R. K. *et al.* A review paper on fog computing paradigm to solve problems and challenges during integration of cloud with iot. *In*: IOP PUBLISHING. **Journal of Physics: Conference Series**. [*S.l.*], 2021. v. 2007, n. 1, p. 012017.

DONG, G. *et al.* A review of the autoencoder and its variants: A comparative perspective from target recognition in synthetic-aperture radar images. **IEEE Geoscience and Remote Sensing Magazine**, v. 6, n. 3, p. 44–68, 2018.

FERNÁNDEZ-DELGADO, M. *et al.* An extensive experimental survey of regression methods. **Neural Networks**, v. 111, p. 11–34, 2019.

GRANJAL, J.; MONTEIRO, E.; SILVA, J. S. Security for the internet of things: A survey of existing protocols and open research issues. **IEEE Communications Surveys Tutorials**, v. 17, n. 3, p. 1294–1312, 2015.

GRAVES, A. **Supervised Sequence Labelling with Recurrent Neural Networks**. [*S.l.*]: Springer Berlin, Heidelberg, 2012. (Studies in Computational Intelligence). ISBN 9783642247972.

HAWKINS, S. *et al.* Outlier detection using replicator neural networks. *In*: SPRINGER. **International Conference on Data Warehousing and Knowledge Discovery**. [*S.l.*], 2002. p. 170–180.

HAYKIN, S.; NETWORK, N. A comprehensive foundation. **Neural networks**, v. 2, n. 2004, p. 41, 2004.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, MIT press, v. 9, n. 8, p. 1735–1780, 1997.

HOI, S. C. *et al.* Online learning: A comprehensive survey. **Neurocomputing**, v. 459, p. 249–289, 2021.

KESKINOCAK, P. **On-line algorithms: How much is it worth to know the future**. [*S.l.*]: IBM Thomas J. Watson Research Division, 1998.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

LEE, S. *et al.* Offline-to-online reinforcement learning via balanced replay and pessimistic q-ensemble. *In*: **Conference on Robot Learning**. [*S.l.*: *s.n.*], 2022. p. 1702–1712.

LIAN, C. *et al.* Ann-enhanced iot wristband for recognition of player identity and shot types based on basketball shooting motion analysis. **IEEE Sensors Journal**, v. 22, n. 2, p. 1404–1413, 2022.

MAMANN, L. *et al.* Paradigma orientado a notificações aplicado à programação de microcontroladores. *In*: **Anais Estendidos do XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2021. p. 134–139. ISSN 2763-9002. Disponível em: https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/18505.

MAMANN, L. V. D.; PIGATTO, D. F.; DELGADO, M. R. Offline and online neural network learning in the context of smart homes and fog computing. *In*: SPRINGER. **Brazilian Conference on Intelligent Systems**. [*S.l.*], 2022. p. 357–372.

MATT, C. Fog computing: Complementing cloud computing to facilitate industry 4.0. **Business & information systems engineering**, Springer, v. 60, p. 351–355, 2018.

MINSKY, M.; PAPERT, S. Perceptrons: An introduction to computational geometry. **Cambridge tiass., HIT**, v. 479, p. 480, 1969.

MINSKY, M.; PAPERT, S. A. **Perceptrons, Reissue of the 1988 Expanded Edition with a new foreword by Léon Bottou: An Introduction to Computational Geometry**. [*S.l.*]: MIT press, 2017.

OPENFOG. Openfog reference architecture for fog computing. *In*: . [*S.l.*: *s.n.*], 2017.

PARISI, G. I. *et al.* Continual lifelong learning with neural networks: A review. **Neural Networks**, v. 113, p. 54–71, 2019.

POTRINO, G.; RANGO, F. D.; SANTAMARIA, A. F. Modeling and evaluation of a new iot security system for mitigating dos attacks to the mqtt broker. **2019 IEEE Wireless Communications and Networking Conference (WCNC)**, p. 1–6, 2019.

PUTTIGE, V. R.; ANAVATTI, S. G. Comparison of real-time online and offline neural network models for a uav. *In*: **2007 International Joint Conference on Neural Networks**. [*S.l.*: *s.n.*], 2007. p. 412–417.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, v. 65, n. 6, p. 386, 1958.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, Springer Science and Business Media LLC, v. 323, n. 6088, p. 533–536, out. 1986.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural Networks**, v. 61, p. 85–117, 2015.

SKOCIR, P. *et al.* Activity detection in smart home environment. *In*: **KES**. [*S.l.*: *s.n.*], 2016.

STERGIOU, C. *et al.* Secure integration of iot and cloud computing. **Future Generation Computer Systems**, v. 78, p. 964–975, 2018.

VARDAKIS, G. *et al.* Smart home: Deep learning as a method for machine learning in recognition of face, silhouette and human activity in the service of a safe home. **Electronics**, v. 11, n. 10, 2022.

YI, S. *et al.* Fog computing: Platform and applications. *In*: **2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)**. [*S.l.*: *s.n.*], 2015. p. 73–78.

YU, J.; ANTONIO, A. de; VILLALBA-MORA, E. Deep learning (cnn, rnn) applications for smart homes: A systematic review. **Computers**, v. 11, n. 2, 2022.

ZHAI, J. *et al.* Autoencoder and its various variants. *In*: **2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. [*S.l.*: *s.n.*], 2018. p. 415–419.

ZHANG, G. Neural networks for classification: a survey. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, v. 30, n. 4, p. 451–462, 2000.

ZHANG, X.-Y.; BENGIO, Y.; LIU, C.-L. Online and offline handwritten chinese character recognition: A comprehensive study and new benchmark. **Pattern Recognition**, v. 61, p. 348–360, 2017. ISSN 0031-3203.