

Methodologies for the prediction of network usage within the context of cellular hotspots

George Koutroumpas

A thesis presented for the degree of
Electrical and Computer Engineer



**ΠΟΛΥΤΕΧΝΕΙΟ
ΚΡΗΤΗΣ /
TECHNICAL
UNIVERSITY
OF CRETE**

Examination Committee:
Professor Sotiris Ioannidis
Professor Apostolos Dollas
Professor Michail Zervakis

December 2022

Abstract

Today we live in a society that relies upon technology. Everything we see around us has become more and more advanced with the addition of smart phones, smart cars and maybe even smart clothes. Many of those devices require the remote use of the world wide web to function properly. This fact, in conjunction with the ever increasing population in central living areas, creates severe issues to mobile service providers. Sudden demand bursts of their service can cause bottlenecks to the network infrastructure, resulting in performance issues to the cellular antennas. An interesting solution for this problem is forecasting when and where those performance drops will happen and re-calibrating the network parameters, effectively avoiding disaster. In this work I propose a neural network algorithm that will handle the forecasting task of those performance drops, referring to them as hotspots. To achieve this goal I am going to cooperate with the company Telefonica, which will provide essential information gathered from its networks antennas, as well as important feedback towards the final product. Using a combination of Gated Recurrent Units and Graph Convolution Networks the plan is to capture spatial and temporal dependencies that exist in the networks behaviour, effectively predicting most of the real performance drops in long prediction horizons. The focus of this work is to have accurate predictions of as many hotspots as possible and on the same time support a vast amount of antennas in the calculation.

Acknowledgements

First of all I would like to thank my supervisor Professor Sotiris Ioannidis for offering me this assignment. I would also like to express my gratitude to Ioannis Arapakis , project manager at Telefonica, for providing me with guidance in the model design process and the necessary data for this project. Cheers to coworker and co-student Konstantinos Zacharopoulos for the great cooperation and excellent partnership in the pursuit of our goals. An extra thanks to members of the MHL lab Konstantinos Georgopoulos and Ioannis Moriannos for their support.

Contents

Abstract	I
Acknowledgements	II
List of Figures	V
List of Tables	VI
1 Introduction	1
1.1 Dissertation theme	1
1.1.1 Object and motivation	1
1.1.2 Problem formulation	2
1.2 Project contribution	2
1.2.1 Approach	3
1.3 Chapter contents	3
2 Background	5
2.1 Introduction to Machine Learning	5
2.1.1 Artificial Neural Networks	5
2.1.2 Model training	7
2.1.3 Activation functions	7
2.1.4 Loss functions	9
2.1.5 Data preparation	9
2.1.6 Classification performance metrics	11
2.2 Relevant NN architectures	13
2.2.1 Convolutional Neural Networks(CNN)	13
2.2.2 Graph Convolutional Networks	14
2.2.3 Recurrent neural networks and Gated Recurrent Units	16
2.2.4 Autoencoders	17

3	State-of-the-art	20
3.1	Existing solutions to general hotspot prediction	20
4	Model architecture	22
4.1	Data preparation	22
4.1.1	Missing value imputation	22
4.1.2	Feature analysis and feature extraction using Autoencoders	24
4.2	Graph neighbour logic	26
4.3	Neural Network design principles	27
4.3.1	Importance of performance metrics and class imbalance	29
4.3.2	Detection of temporal dependencies	29
4.3.3	Detection of spatial patterns	30
4.3.4	Activation function choices	30
4.3.5	Loss function choices	31
4.4	Optimization and hyper-parameter tuning	32
4.4.1	Optimizer and learning rate	33
4.4.2	Loss weights	33
4.4.3	GCN and GRU configuration	34
4.4.4	Epochs	35
4.4.5	Graph logic optimization and comparison	36
5	Results	38
5.1	Final model tests and comparison to baselines	38
5.2	Class weights and precision/recall trade-off	40
6	Conclusions	42
6.1	General observations	42
6.2	Thesis conclusion	43
6.3	Alternative implementations/Future plans	43

List of Figures

2.1	Artificial neuron logic based on real neuron. [1]	6
2.2	Mathematical flow of an artificial neuron. [2]	6
2.3	Famous activation functions. [3]	8
2.4	Common data processing flow. [4]	12
2.5	Example of Convolutional Neural Network for character recognition, here max is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map [5]	14
2.6	Graph Convolutional Network node traversal. [6]	15
2.7	Basic Recurrent Neural Network. [7]	16
2.8	LSTM architecture. [8]	17
2.9	GRU architecture. [9]	17
2.10	Basic Autoencoder architecture. [10]	19
4.1	Simple data imputation [11]	23
4.2	Example of a node connected by knn with k=5. [12]	27
4.3	Basic diagram of the final proposed model.	28
4.4	Simple example of message passing on a target node. [13]	31

List of Tables

4.1	Autoencoder reconstruction accuracy using different compression ratios	25
4.2	Activation functions used and their derivatives.	32
4.3	GRU performance using prediction horizon of 12 hours ahead and class weights with the target of reaching high recall.	34
4.4	GCN performance using different layers and neurons. The neighbour logic used on the test was distance based with maximum neighbour distance 0.1 kilometers.	35
4.5	GRU-GCN model performance using different neighbour logic. The main parameters of the model are 24 hours of history, 3 GCN layers with 256 neurons and class weights optimized for recall.	37
5.1	Model and baseline performance with the target of high recall.	39
5.2	Original data versus Autoencoder output	40
5.3	Model and baseline performance with the target of higher precision at horizon of 12 hours ahead.	41

1 Introduction

1.1 Dissertation theme

1.1.1 Object and motivation

With the ever increasing power of computer hardware that has been observed in the past years, a very useful and resource hungry computer science section has been revived, machine learning. Machine learning is a branch of computer algorithms, whose goal is to detect behaviour patterns in data and use them to create a desirable output. These techniques are based on empirical data analysis and are used for data classification, future state prediction, system automation etc. The advantage of those methods over conventional artificial intelligence algorithms is that they are fully automated and can handle varieties of data with a more advanced scope than that of a human. This makes them perform especially well in real world applications, where the engineers scope may limit the performance of the automation algorithm.

Along with the rapid advancements in hardware technology there exists another interesting phenomenon, the rapid increase of mobile usage in large population centers. Today smart devices such as electric cars, smartphones, internet of things devices etc. create massive data traffic in mobile networks, that is growing at an exponential rate. Furthermore many of those devices are not static, meaning they will not apply their traffic load in one specific area. This creates an interesting problem for providers of telecommunication services, where the pattern recognition of traffic overload in several antenna regions remains to this date a high-value and challenging forecasting problem. At the same time it impacts the Quality of Experience (QoE) for the customer, since such overloads may cause low response times, packet loss, or even total service loss to

their device. I will refer to those problematic antenna areas as hotspots. In cooperation with the company Telefonica and fellow student Konstantinos Zacharopoulos we plan to tackle this problem by implementing two advanced automated machine learning algorithms. The first algorithm that is described in this thesis has the goal of forecasting with high recall using the full set of antennas, with a secondary goal of being light enough to include as many antennas as possible into one prediction while also investigating how the model behaves while changing the goal to achieve high precision. The second algorithm implemented in the thesis of Kostas Zacharopoulos aims to split the antennas into smaller regions and achieve predictions with high precision.

1.1.2 Problem formulation

The problem that is being addressed in this project is the prediction of the system overload on the networks cellular antennas, which happens due to the increased amount of the network service usage. In order to solve this issue I am going to create a machine learning algorithm that forecasts the future load state of an antenna using its key performance indicators. These indicators consist of sensor values that indicate the state of the device and measure the quality of the provided services to the users. These predictions will be given to the network engineers responsible for the targeted areas, who will in turn calibrate the network parameters and as a result be able to handle the load on the antennas. It is important for this goal to predict as many of the real hotspots, while still maintaining high accuracy in the total network predictions, meaning that I can tolerate low true positive to false positive ratio. Part of the main target is also the feasibility of the model to train on large scale mobile networks without overloading the used system resources.

1.2 Project contribution

To my knowledge, there only exists two works addressing the problem of long term prediction of antenna hotspots. The first one is the work of Serra et al. [14], where the authors propose the use of tree based models in order to find patterns in the networks data. There they observed

patterns both in the antenna location and in the temporal nature of the data. In the second, more recent work of Zhou et al. [15], an LSTM based model was successfully deployed to predict hotspots for a Chinese mobile network. In the approach of this thesis, based on the findings of those two precious works, I am going to construct a graph-based recurrent neural network (RNN) model so that I can take advantage of both spatial and temporal patterns that lie in the data. By exploiting the predictive power of such architectures I hope to have an increased prediction performance in long prediction horizons, while having a faster run-time compared to previous works and be able to process a more heavyweight antenna network.

1.2.1 Approach

As mentioned earlier, my model consists of a graph-based RNN model. Essentially, I am going to embed a Gated Recurrent Unit (GRU) as an initial module, followed by a Graph Convolution Network (GCN). The role of the GRU is creating a temporal representation of the data, detecting such dependencies, later feeding them into a GCN module, where the effects of close-by antennas to their neighbours are being considered. The final output will be used as the prediction probability of a future hotspot in a given time horizon per antenna. The goal is to have accurate predictions on the full antenna graph, with a focus on recall over precision. One of the challenges is the graph size, which is naturally big since I am dealing with a big data problem on the full grid of London. It is also important to achieve a high ratio of true positives to total real positives, while maintaining high accuracy, since it is more important to predict most hotspot occurrences, even if there exist a fair amount of false positives.

1.3 Chapter contents

Here i am going to present a small overview of the chapter contents of this thesis. Chapter 1 contains a small introduction to the problem that this work is based on. On the second chapter I provide the theoretical background of how a machine learning algorithm works and how it has to

be designed in order to function properly. In chapter 3 the current state of the art is being showcased, including some works that either relate or even predate my state of affairs. In the next chapter I exhibit in depth the design process of the network and the required preparation, followed by chapter 5 where I post my experimental results. Finally chapter 6 includes the conclusion of my work, proposing future enhancements and possible additions.

2 Background

In this section I am going to disclose the theoretical background of the design process of a learning algorithm [16], while also introducing some useful neural network categories that the suggested model directly derives from.

2.1 Introduction to Machine Learning

2.1.1 Artificial Neural Networks

Machine learning (ML) is the study of creating algorithms capable of 'intelligent' thinking. They are used to recognize patterns in input data in order to solve classification tasks, target recognition , prediction or even new output creation. One of the more popular methods of machine learning algorithms are neural networks (NNs). NNs algorithms [17] are high capacity function approximators that work using a network of artificial neurons. Those neurons are based on the operation of an organic neuron (see Figure 2.1). They typically consist of a weighted sum function and an activation function, as shown in Figure 2.2. The first function processes the input data using learn-able weights and the second one applies non-linearity to them.

The goal of neural networks is to adjust the weights of the neurons in order to approximate the behaviour of the target into a mathematical function. This is done by adjusting the weights of the neurons based on a set of the input data, also referred to as training of the neural network model. The performance of the trained model is then rated based on a number of metrics, using a separate set of data(i.e. test set). After several training cycles, called epochs, when those metrics meet certain criteria the model is then ready for use.

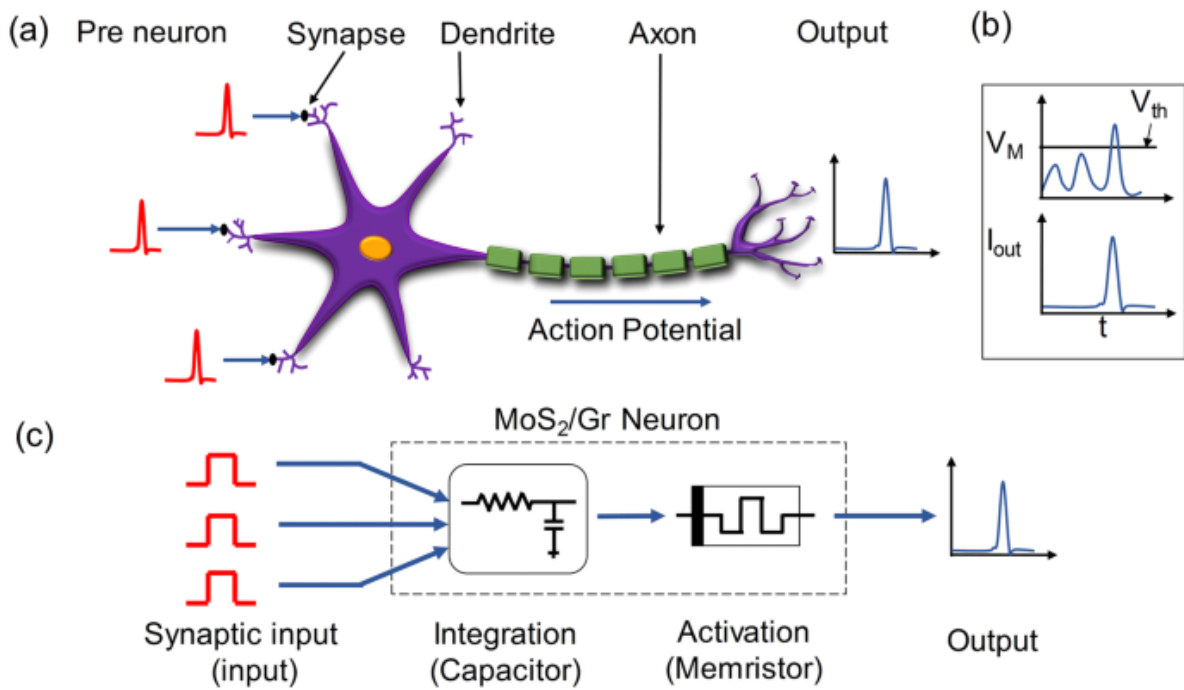


Figure 2.1: Artificial neuron logic based on real neuron. [1]

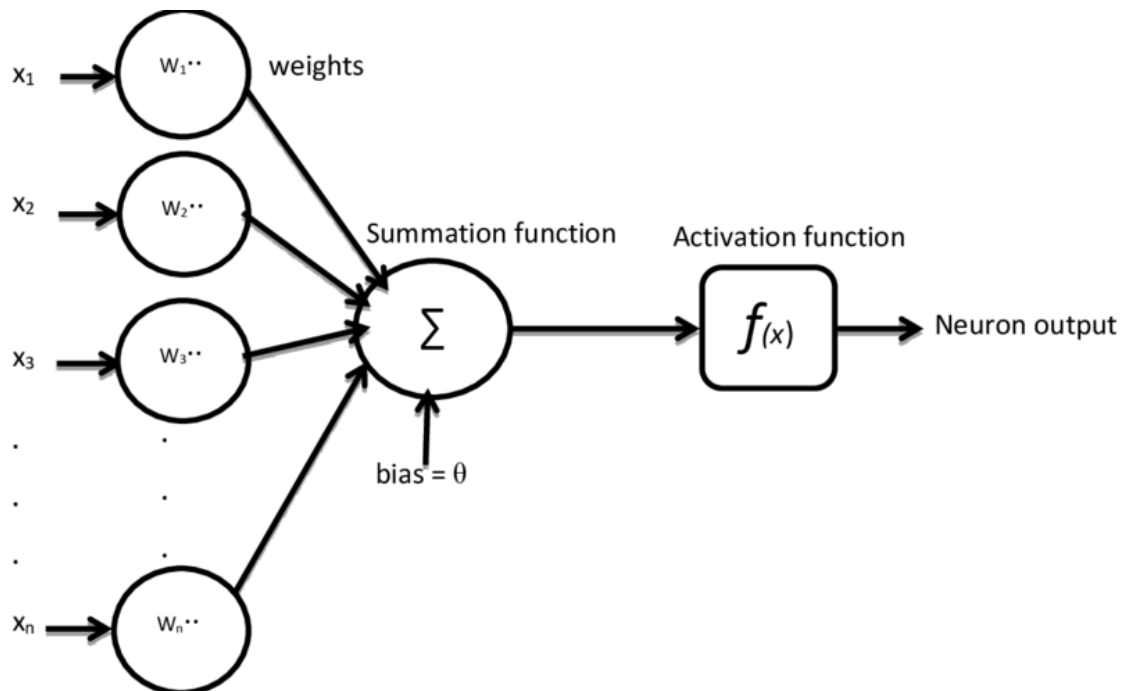


Figure 2.2: Mathematical flow of an artificial neuron. [2]

2.1.2 Model training

Training is a critical part of how neural network models work. As mentioned, in training the model uses a certain set of data to adjust its weights, so that it optimises its performance for a given task. In training, the input data is fed into a model with randomized weights. After all the calculations are done the output is being fed forward to a loss function. This function measures how much the output information deviates from the expected real output. Based on that measurement the neuron weights are being adjusted using back-propagation (see [18]). Back-propagation functions by computing the gradient of the loss function starting from the last layer of neurons to the first using the chain rule, re-calibrating the weights. There are many methods to train a model, based on the data at hand. The most common of them are:

- Supervise Learning: In supervised learning, the model is given both the input and the expected output, named label, and is expected to map the correct behaviour to meet our expectations. Supervised learning is commonly utilized for classification tasks.
- Unsupervised Learning: In unsupervised learning the data given is unstructured/unlabeled. The goal is to find a pattern in them and use it to fulfill a task. Unsupervised learning is used typically in statistics.
- Reinforcement Learning: In reinforcement learning the model operates in an environment, in which it learns by receiving a reward based on its actions. The input data consists usually of environmental variables based on the devices sensory. Reinforcement learning is mostly applied in automated agents.

2.1.3 Activation functions

In ML, an activation function is a way of deciding how the neuron will behave and whether it will "activate". Activation functions essentially control the domain of the values that the models output can take. They are placed in the end of a NN layer, adding non-linearity to the network. This non-linearity is what provides the current model the ability to learn

how to solve more complex problems. Some famous activation functions are the binary step, which activates to 1 only then the input x goes above a threshold, ReLu , which activates x when the input x is above 0 and the sigmoid which outputs a value between 0 and 1 depending on the input x , where $output = 1/(1+e^{-x})$. In the works of Chigozie Nwankpa et al. [19] and Shiv Ram Dubey et al. [20] there are some interesting comparisons in the performance of the above and more activation functions while used in deep learning problems. Figure 2.3 shows more famous activation functions along with their plot diagrams, equations, first derivatives and output range.









ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

Figure 2.3: Famous activation functions. [3]

2.1.4 Loss functions

Another important aspect of NN's are loss functions. These functions are responsible for fitting the model to the given training set. They operate by comparing the target and predicted output values and compute a new value called loss. The goal of the model is typically to minimize the loss and as a result bring the outputs closer to the target values.

There are many different loss functions depending on the use case. In regression, the most typical loss functions are mean squared error (MSE) and mean absolute error (MAE). These two functions operate exactly as they are named, MSE calculates the mean of all squared errors in an instance 2.1 and MAE the mean of the absolute deviation per instance 2.2. In classification tasks, where the output is a probability, cross entropy is typically used, which compares the distribution of the prediction probability to the target values. In binary classification cross entropy follows Equation 2.4 and in multi-class problems the Equation 2.3.

$$MAE = \sum_{i=1}^D |x_i - y_i| \quad (2.1)$$

$$MSE = \sum_{i=1}^D (x_i - y_i)^2 \quad (2.2)$$

$$CrossEntropy = - \sum_{c=1}^M y_{o,c} \log(p_{o,c}) \quad (2.3)$$

$$BinaryCrossEntropy = -(y \log(p) + (1 - y) \log(1 - p)) \quad (2.4)$$

2.1.5 Data preparation

One of the key factors for the proper operation of NNs, is the preparation of the input data. A good in-depth analysis of data preparation can be found in [21], as shown in Figure 2.4. Initially, the data that is going to be analysed needs to be checked for missing or corrupt information. When working with real-world data, it is very common for phenomena like sensor failures to occur, leading to incomplete data capture. There are two basic ways to handle those areas, either remove the data tuple

that has blank spots, or fill it using an imputation technique, that is, using a statistical or a learning approach to guess what is missing.

After having filled the missing spots in the data set, excess information has to be removed. Usually such data exists in points of data that do not provide useful patterns to the model. These can be columns or features that contain the same value across the whole column, or columns with identical or very close values, meaning that they have close to 0 variance. Keeping those features in the data will only make the model more resource hungry, while at the same time possibly degrading its performance by introducing noise and increasing computations needed. Some simple categories of methods to battle this problem are feature selection and feature extraction. In feature selection the columns or features are being compared, and only the ones of high value are being kept in the input set. In feature extraction the raw data is fed into a transforming algorithm that calculates the new features. Feature extraction techniques typically apply a dimensionality reduction algorithm to the initial set and create a new one for the target model. This new set consists of new features, fewer in number than the initial set, that give a representation of the old tuples. Some well used algorithms for this function are the Autoencoder and Principal Component Analysis.

I also need to check the format of the data. In some columns there may be encountered data with high variance or with categorical values. Categorical values cannot be processed directly by a neural network, and data with high variance and big values can make it harder for the model to make accurate predictions. To deal with categorical values one-hot encoding is applied, where the non-integers are being encoded into integers. After that the data columns can either be normalized, giving them values between 0 and 1, or standardized, giving them a center value of 0 and variance at 1. This part of normalizing or standardizing the input is referred to as data scaling.

A final step of the data pre-processing is splitting it into a train group, a validation group and a test group. The purpose of splitting the data is to avoid phenomena such as over-fitting the classifier into a specific scenario, and to ensure its ability to generalize the given task. The split is done randomly to avoid any bias in the model training or testing. The

training set usually consists of the larger portion of the data, since this improves the model fitting into a more complex behaviour pattern. The validation set should be a small percentage, close to 10-20% the total data, and is used to validate the performance of the model in the end of each training cycle and adjust the hyper-parameters of the model. It also helps detect over-fitting and apply early stopping of the training function, if it is needed. Finally the testing step should be close to 15-20% of the total data, and is used to test the final performance of the model.

2.1.6 Classification performance metrics

To know how well the model fitted into the dataset I need a method of performance measurement. Here, I am going to introduce some of the important metrics that perform this task. These metrics are the standard Information Retrieval (IR) performance metrics of accuracy, precision, recall, f-score, area under curve and hit rate [22]. Their calculation relies on the use of a confusion matrix, which is a table that contains the number of true positive predictions TP, the number of true negative predictions TN, the number of false positive predictions FP and the number of false negative predictions FN. In non-binary classification tasks, the confusion matrix contains the TN, TP, FN, FP for each class. Using those values what each metric measures can be defined mathematically what each metric measures. Accuracy measures how accurate the model is by comparing the total number of correct predictions to the total number of predictions as in Equation 2.5. Precision calculates the ratio of true positives to the predicted positives as shown in Equation 2.6 and is a measure of how confident the model is in the calculations. Recall is the ratio of the true positives to the total positives as in the Equation 2.7 and expresses the models sensitivity to the detection of the target class. F-score is the harmonic mean of precision and recall as shown in the Equation 2.8 and considers both recall and precision. A version of F-score exists, called F-beta, which considers that recall is beta times as important as precision, with the Equation 2.9. Area under curve (AUC) calculates the relationship between the true positive rate and the true negative rate and gives the information of how well a model can differen-

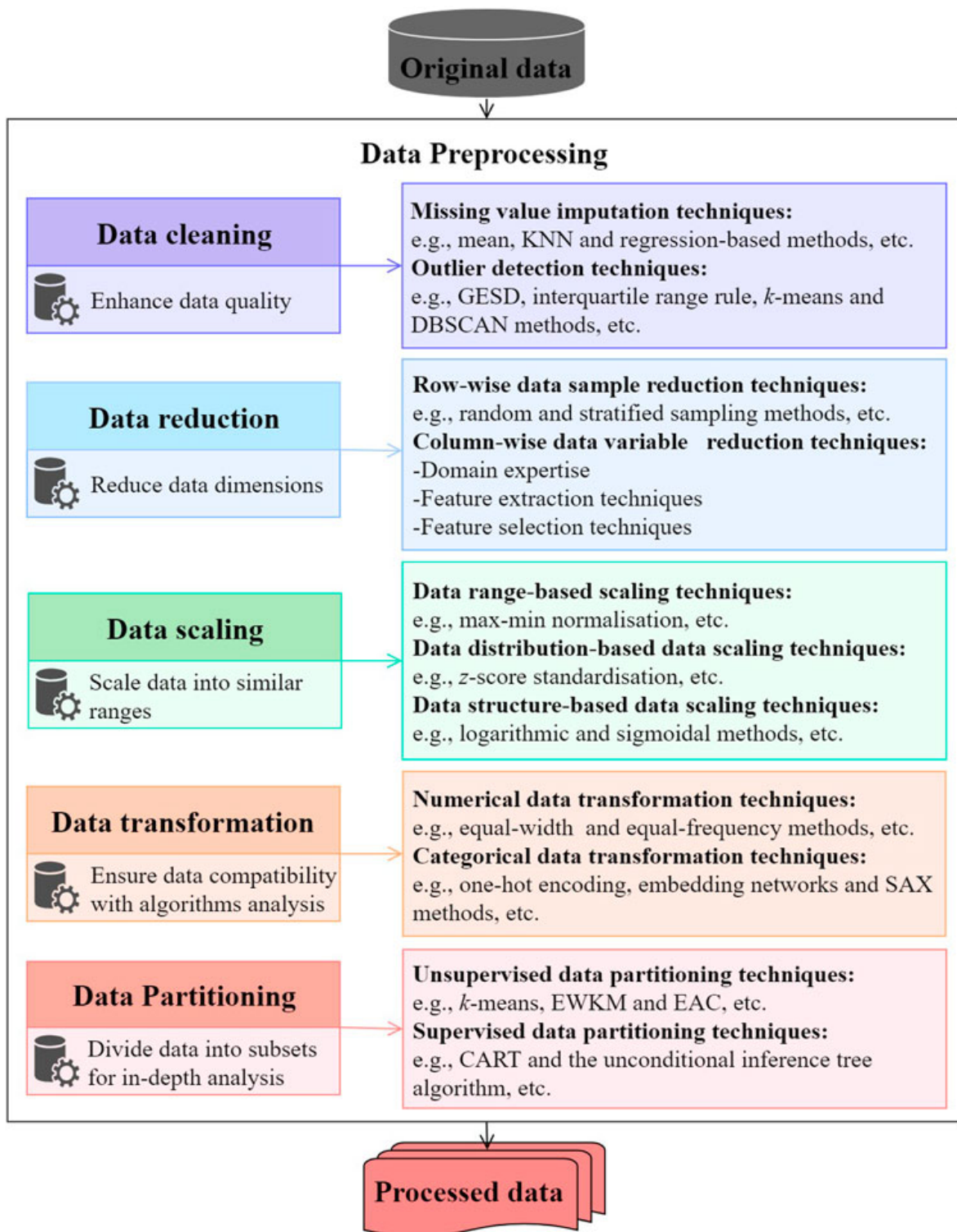


Figure 2.4: Common data processing flow. [4]

tiate between the classes. Finally hit rate at x contains the information of how well the precision was at the most probable x predictions and can help decide if and where to change the decision threshold. All of those metrics are useful tools in handling spatial cases, where the typical metric of accuracy may lead to false claims, like it is showcased in this example of Brian Liu et al. [23] where accuracy is put on the test versus precision.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (2.5)$$

$$Precision = \frac{TP}{TP + FP} \quad (2.6)$$

$$Recall = \frac{TP}{TP + FN} \quad (2.7)$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (2.8)$$

$$Fb = (1 + b^2) * \frac{2 * Precision * Recall}{b^2 * Precision + Recall} \quad (2.9)$$

2.2 Relevant NN architectures

2.2.1 Convolutional Neural Networks(CNN)

In this section I am going to describe some useful neural networks related to the project, starting from CNNs [24]. As their name implies CNNs are neural networks that apply a mathematical function called a convolution. At the point of the convolution those networks use a set of learn-able filters, also called kernels, to apply the convolution to the input data. Using those kernels CNNs are able to learn, store and identify patterns in the data. Additionally due to the abilities of the convolution, the pattern detection is invariant of the spatial location of the pattern. This means that a kernel that detects dots in an image can detect them wherever they are located and not a specific spot. The rest of their functionality is similar to the typical neural networks. Convolutional neural networks are intended for use with multidimensional data and as such are

typically applied to image and time-series related works. Convolutional neural networks have proved to be a very powerful type of network in such use cases and have been widely deployed for pattern recognition in images and time series data. Convolutional neural networks have been used in conjunction with LSTM type RNNs for traffic flow prediction. In the works of Zhao et al. and Zhuang et al. [25, 26] they have been proven to work effectively in short-term predictions. Also in the work [27] CNNs are able to handle sets with sizable feature vectors. Figure 2.5 show a well known application of CNNs, character recognition.

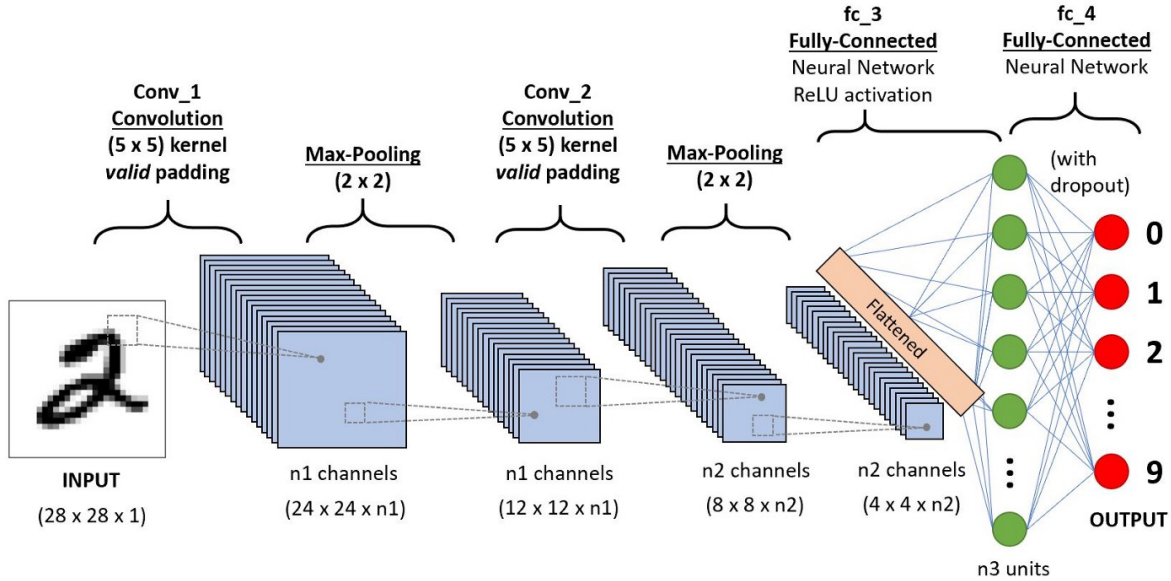


Figure 2.5: Example of Convolutional Neural Network for character recognition, here max is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map [5]

2.2.2 Graph Convolutional Networks

Graph Convolutional Networks, or GCNs , are a generalization of CNNs that also take advantage of the graph structure of the data. The GCNs rely on an adjacency or Laplacian matrix to represent the structure of a graph [28, 29]. The input of the GCNs is the data followed

by an adjacency matrix that describes the relation between each data point. Then message passing is applied to each target data point, where the data of the adjacent 'neighbour' nodes are being aggregated with the target data. The output of the aggregation is being fed into a CNN. GCNs allow taking the advantage of the spatial influence contained in close-by data points and thus help in detecting the wanted phenomenon. The main focus is on GCNs since they have been typically used in association with traffic forecasting problems containing important spatial patterns with high success rates.

In order to detect spatio-temporal dynamics of traffic data, GCNs have been used accompanied with RNNs. In the work of Zhao et al. [30] GCNs have been fused with GRU or LSTM components with the goal of traffic control. There, the combination with GRU outperformed LSTM, while also outperforming algorithms GCN, GRU, ARIMA and SVR in their prediction. Another project using GCN-LSTM architecture has shown to handle network scale data, similar to my interest [31].

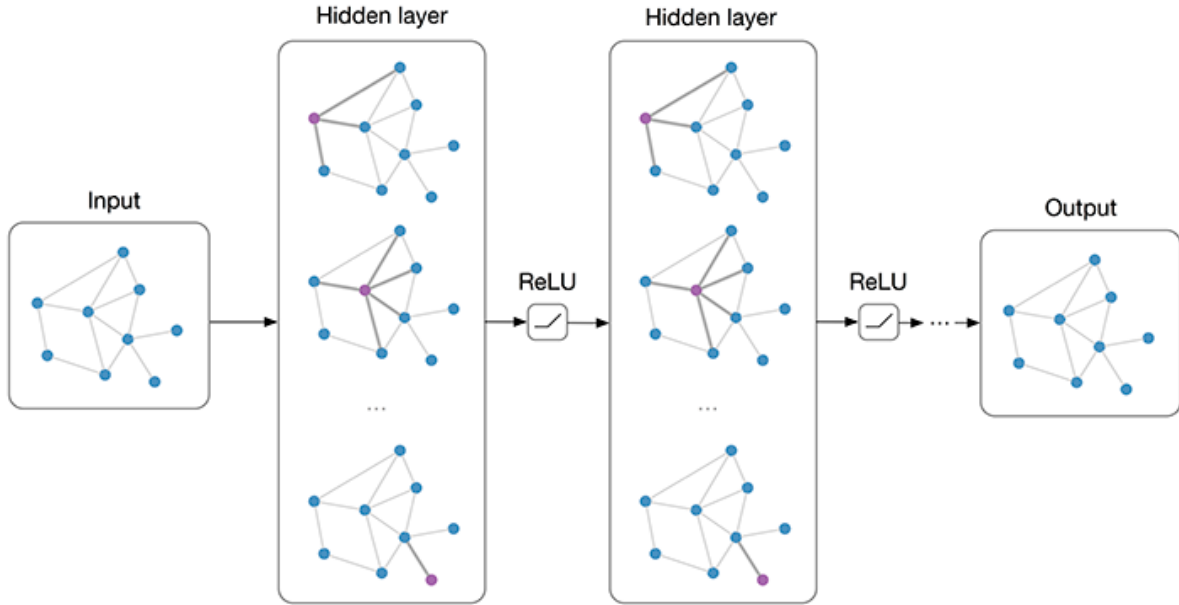


Figure 2.6: Graph Convolutional Network node traversal. [6]

2.2.3 Recurrent neural networks and Gated Recurrent Units

Another well-used category of algorithms, that I built upon, are recurrent neural networks (RNNs). This type of algorithm takes advantage of artificial memory in order to maintain information patterns through cyclical runs on the input data. Thanks to that memory, these algorithms usually achieve high accuracy. They are also invariant to the sequence length. This behaviour makes them suitable for recognition using time-series as inputs, like speech recognition. RNNs have been widely used as part of a traffic forecasting model to predict traffic speed [32], travel time [33], and traffic flow [34, 35, 36].

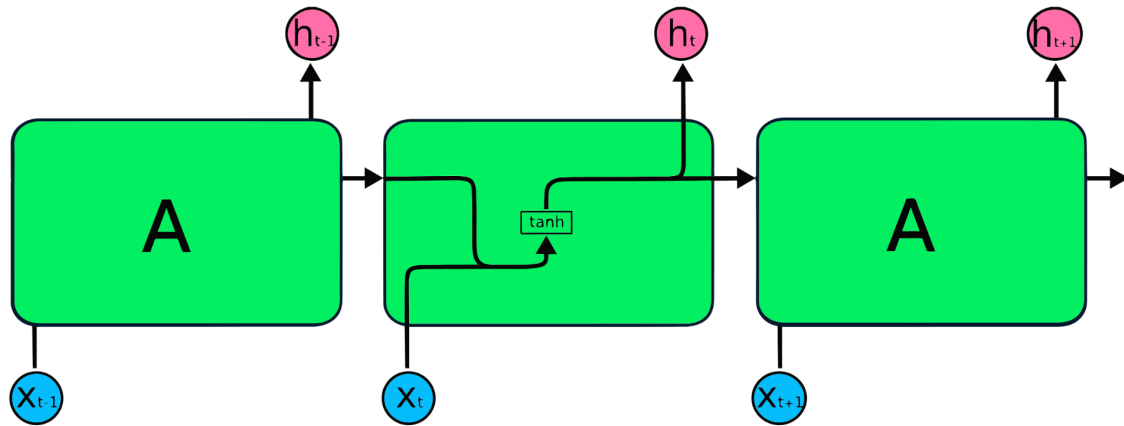


Figure 2.7: Basic Recurrent Neural Network. [7]

The most effective RNNs are Gated Recurrent Units (GRU) and Long Short Term Memory (LSTM). GRUs are RNNs that use two gates to keep historical information, the update and the forget gate. The update gate regulates what information will be passed from the previous time-step data to the next, while the forget gate decides how much of the information should be forgotten. The input and the output of the GRU consist of a data vector and a history vector. GRUs are widely used in works with less frequent data sets than other RNNs. LSTMs function

similarly to GRU, having one additional forget gate. They are computationally more heavyweight than GRU, while seemingly having very close performance to it.

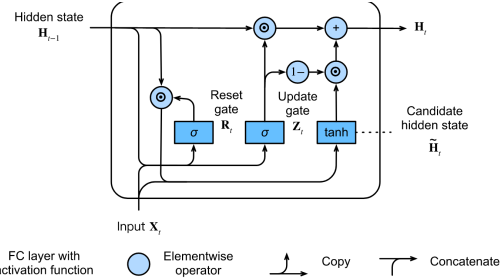
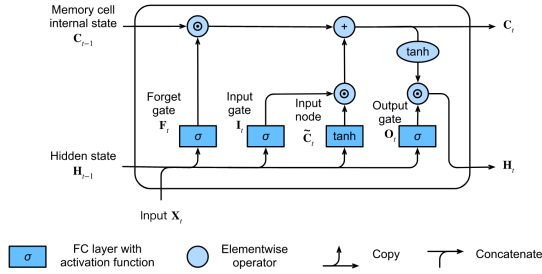


Figure 2.8: LSTM architecture. [8] Figure 2.9: GRU architecture. [9]

2.2.4 Autoencoders

Autoencoders [37] are a family of neural networks, whose goal is to compress the input by lowering its dimensionality and then reconstruct it as close as possible. To achieve that, the autoencoder has multiple layers and is split in two parts, the encoder and the decoder. The first layers, belonging to the encoder shrink the input, reaching a minimum size for the neural network. The information they contain is a compressed, distorted representation of the original, named the code. The code still contains most of the original data attributes and thus can be very useful, especially when dimensionality reduction is needed in a dataset. The code is then fed to the decoder, which consists of layers that increase in size. A basic autoencoder is shown in Figure 2.10 In a basic autoencoder, the role of the decoder would be to use the code to recreate the original dataset as close as possible. After training an autoencoder, I can get an output that is very close to the input, but also get a better code, more representative of the input set. There are various ways to take advantage of such a network. The input can be corrupted with noise and then train an autoencoder on recreating the original, clean data, essentially teaching it to remove noise. The code can also be used as the new 'features' of a clean set to make it more lightweight to train with in other applications. A disadvantage of autoencoders is, that they encounter a

bias-variance trade-off, creating problems when trying to minimize the loss of such a model.

Depending on the application, there are different autoencoder types. Some of the more common ones are denoising , sparse and contractive.

- Sparse autoencoders: They deal with the bias-variance trade-off by enforcing sparsity on the hidden activation, using sparsity regularization. This regularization is applied to the activation instead of the weights. Sparse autoencoders have been widely used for dimensionality reduction like in the work of [38].
- Denoising autoencoders: This category of robust autoencoders are used for error correction. In this configuration the model is trained with corrupted input and is expected to reconstruct the original input [39].
- Contractive autoencoders : They work on the opposing principle to denoising ones. Instead of trying to resist perturbations of the input, the emphasis is to be less sensitive to them. Contractive autoencoders are usually employed as just one of several other autoencoder nodes, activating only when other encoding schemes fail to label a data point

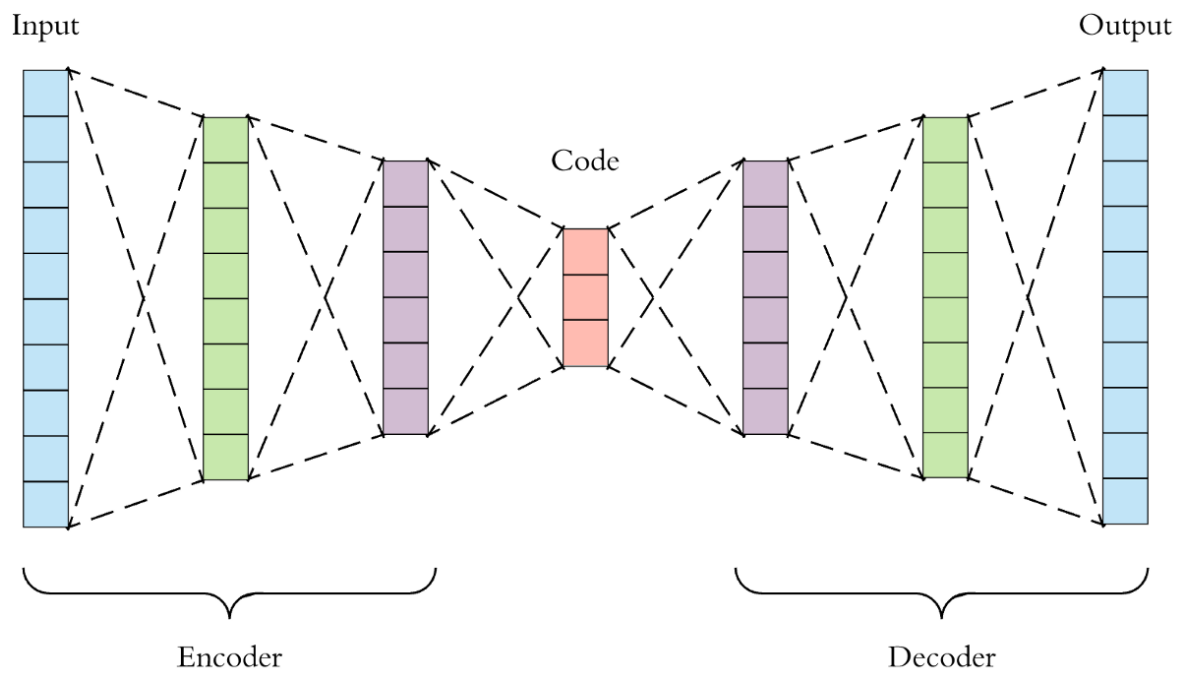


Figure 2.10: Basic Autoencoder architecture. [10]

3 State-of-the-art

Cellular performance forecasting has been a popular research topic in the recent years. In this section I will present some of the most important works that influenced my design, as well as discuss their limitations that I am addressing.

3.1 Existing solutions to general hotspot prediction

Traffic forecasting belongs to the domain of multivariate time series problems. In order to solve such problems ML methods have been found to outperform statistical methods like ARIMA, thanks to their ability to learn complex, non-linear relationships. With the added introduction of high capacity neural networks in combination with more effective training algorithms, it is also possible to learn using big data collections. Such algorithms have been used, initially for short term mobile performance prediction, like the work “Proteus” [40], which consisted of a tree based model.

In the later work “Hot or Not” [14], the target horizon was hours to days later. In said work, the authors gave a more in depth analysis of the cellular network dynamics. Serra et al. [14] casted the problem as multivariate time series forecasting and focused mainly on the temporal patterns, using a Random Forest (RF) model. This model outperformed the tested baselines in short term horizons, but its advantages vanish in longer horizons, while also not accounting for the spatial patterns of the data.

A more recent work by Zhou et al. [15] captures the temporal dependencies of the network using an LSTM-based model. The model was trained with the data of a network in a Chinese city and produced ad-

mirable results in longer horizons. This solution is the closest one to my goal. I plan to improve on the prediction by using better performing RNN algorithms than LSTMs, while also incorporating spatial dependencies to my predictor.

4 Model architecture

In this section I describe the proposed models design process and analyze the principles that lead to my approach. The main analysis is split into four parts, the data preparation, the creation of the graph used on the graph based version of the network, the basic neural network, and the optimization of it. My initial goal is to run the model on the full graph while maintaining high accuracy, high recall with a relatively good precision. A later goal is to investigate if the precision can be increased without losing the optimal recall.

4.1 Data preparation

As previously discussed, an important part of training a neural network correctly is the data preparation. In this case, the dataset consists of hourly observations from the key performance indicators from 10700 antennas in the location of London, from the month May until August, approximating 2.15 billion tuples of data. Since this is a case of real world data, it is natural to have a number of blank spots on the set. There are also a number of indicator values, or features that are not useful in the analysis. The goal is to find a way to deal with those problems while minimizing the loss of useful information. Before beginning I transform all categorical KPIs to numerical values using one hot encoding, with the exception with some cell identification related features that have to remain unchanged.

4.1.1 Missing value imputation

The first part of the data preparation is filling the gaps of information in the dataset. This task is referred to as data imputation, as shown in Figure 4.1. In this setting it was found that from the whole set, 15

percent of the tuples had at least one missing value from the 77 features at a random location. There are two common approaches to dealing with missing or corrupted data, either imputation as mentioned, or dropping the problematic tuples all together. The reason that I prefer imputation is, that removing certain values will ruin the continuity of the multivariate time series per antenna, and thus negatively affect the prediction. So my choice is imputation.

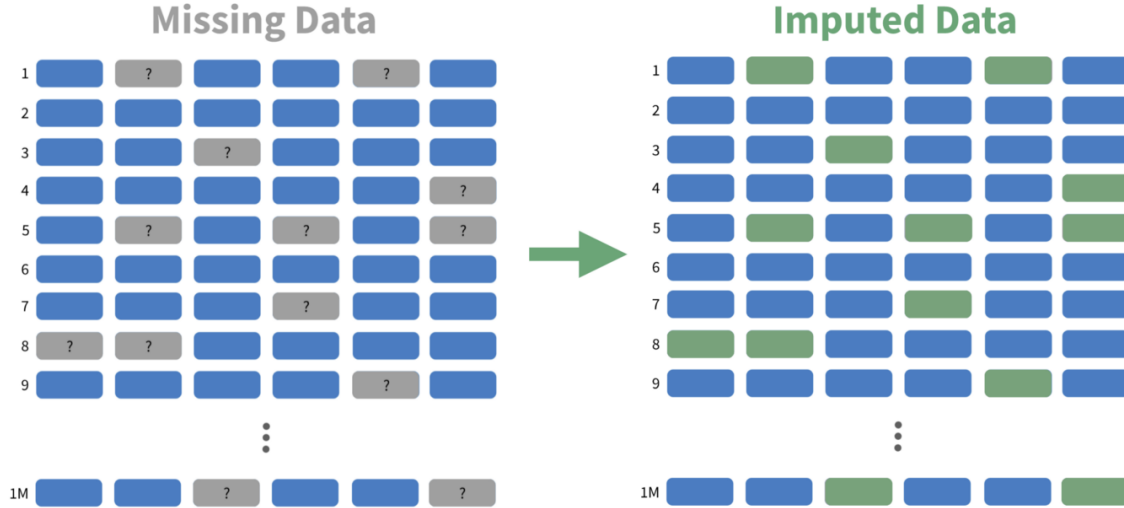


Figure 4.1: Simple data imputation [11]

Some of the more famous techniques to work around this problem are zero filling, mean imputation and imputing using a k-nearest neighbours learning method. In order to test which approach would work the best, all three algorithms were applied in a test subset of the original data named 'A'. Subset 'A' consists of tuples cut from the original set, which do not have any missing values. I artificially corrupt some of A's values at random and then test all three cases. In the case of filling every gap with zeros, the algorithm got the worst accuracy comparing to the original set, as it was originally expected. Filling with the mean values between the gaps had a more acceptable accuracy, while also having a relatively low run-time. Finally I train a k nearest neighbours algorithm that groups the tuples and then imputes based on what values the neighbours of the tuple contains. While more accurate by all 3, the knn algorithm fails in run-time, especially when considering that it has to be applied to a set that has almost 2.15 billion tuples. The final choice was mean imputation,

due to having a more balance accuracy/performance relationship.

4.1.2 Feature analysis and feature extraction using Autoencoders

After filling the gaps in the data, the next part is reducing the dimensionality of the set. My goal is to train a graph-based network with batches of one hour containing information for every cell in the network. Doing that with the full set is nearly impossible due to memory restrictions. The experiments are being conducted on a powerful server, having 256 Gb of RAM and an array of 5 modern GPUs with 12 Gb of VRAM each. Fitting the full graph with all 77 features overloads the server and thus cannot run effectively. A common practice is to apply feature analysis methods to remove features that either contain similar information to each other, or that even contain no information related to the analysis. Out of those techniques, I applied variance thresholding, that removes features that have near-zero variance. Those features mostly contained the same value across all observations, or deviated so little that they would not positively affect the models output.

Since my goal on this task is to reduce the dimensionality as much as possible, after the variance thresholding I am not going to continue using feature analysis methods. A higher reduction in dimensions is needed without the loss of useful information. For that reason I use autoencoders for a method called feature extraction. Here I train a sparse autoencoder to recreate the input set while firstly compressing it and then decompressing it. After optimizing the autoencoder I run the process on the full dataset, while keeping the output of the autoencoders final encoding layer, the aforementioned code. The code is going to replace the original input set as the new input of the proposed model, with the goal of retaining a good prediction while allocating less resources. During optimization I chose an autoencoder with four compression ratios, $2/3$, $1/3$, $1/6$ and $1/9$ of the initial size. The optimal ration was the one with a compression of $1/6$ since as shown in Table 4.1 it gives a high compression with a relatively high reconstruction accuracy, while the next highest compression has a significantly worse reconstruction accuracy. The new set consists of new compressed features that represent

the original full set, having a much smaller size while keeping most of the important information. This allows me to train the forecaster with the current resources while also giving me some extra space to try different configurations, that would otherwise not be applicable.

Compression list	
Compression ratio	Accuracy
2/3	93%
1/3	96%
1/6	95%
1/9	89%

Table 4.1: Autoencoder reconstruction accuracy using different compression ratios

4.2 Graph neighbour logic

The next step that is required to do is to defining the graph logic. In my graph, the nodes represent the cellular antennas that are being investigated . What needs to be define are the edges. An edge translates to the two connected antennas to be treated as neighbours, meaning that their values will be included in the message passing state of the networks Graph based layer. Knowing the location of each antenna I calculate the distances between the cells. Then I apply the three different neighbour criteria to create the graph's adjacency matrix. The criteria are:

- Distance-based: A distance-based criterion decides depending on how far away an antenna is from the target. That antenna is considered a neighbour only if its distance from the target is set bellow a threshold value KM.I experiment in the optimization section with different KM values. With the distance based criterion the idea is that close-by antennas have a high probability of affecting the target one. The downside of it is that it can create a big number on neighbours, which will affect both the performance and the runtime of the model negatively.
- KNN based: A KNN based criterion will simply consider a target antenna neighbour only the K nearest antennas. This method gives a threshold to the potential big number of neighbours, but may leave out some of the antennas that had important influence to the target. It may also have the opposite result of including antennas that are too far to affect the target.
- Cell site based: The final criterion tested is considering only the antennas on the same base station as neighbours. With this criterion the model is limited only to very close antennas, excluding the probability of being affected by ones that are very far away and do not provide anything useful to the prediction.

I will showcase the performance of each different neighbour logic in the results section. There, several variations of each criterion will be compared to each other for the final product.

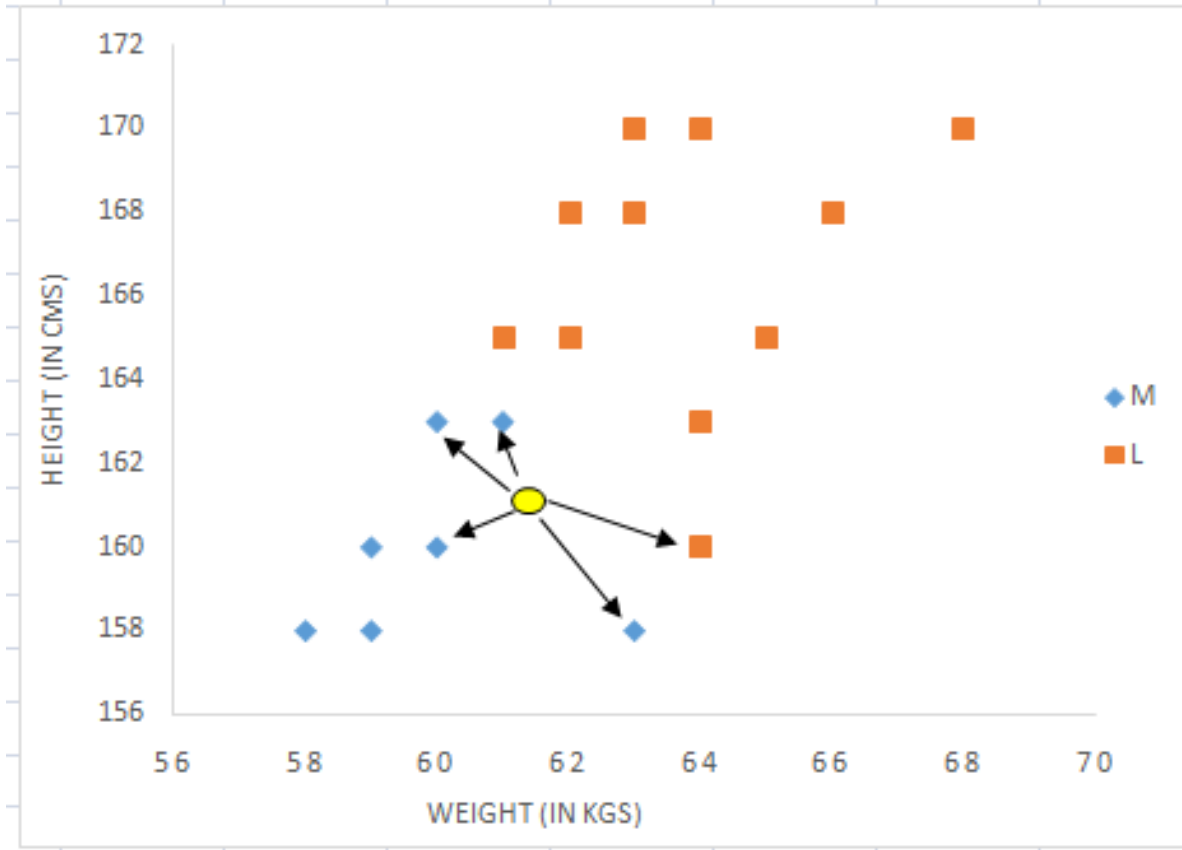


Figure 4.2: Example of a node connected by knn with $k=5$. [12]

4.3 Neural Network design principles

On this section I am going to analyze the design principle of the suggested model. The model itself consists of a Gated Recurrent Unit part that captures temporal dependencies and a Graph Convolutional part to capture spatial ones. The final output of the model is the probability of an antenna becoming a hotspot. The idea of combining those two architectures in order to take advantage of spatio-temporal patterns was influenced by the works of Zhao et al. and Bing et al. [30, 41]. In those papers RNN- type networks were combined with graph-based networks to produce the final classifier/predictor.

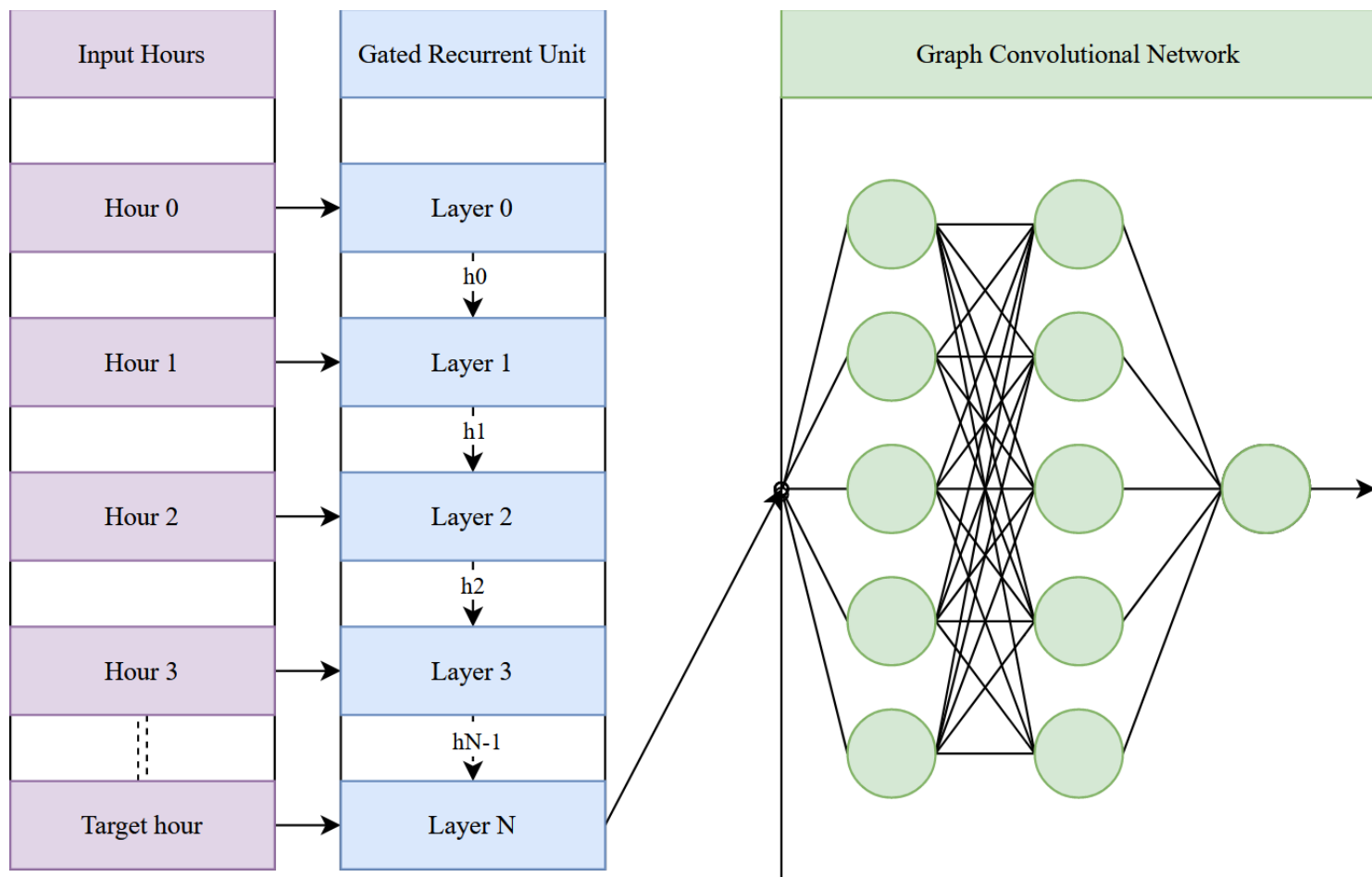


Figure 4.3: Basic diagram of the final proposed model.

4.3.1 Importance of performance metrics and class imbalance

The basic principle of the models design is the exploitation the aforementioned performance metrics. One of the challenges of the prediction goal is the high class imbalance that exists in the input data. The target variable has such a small appearance probability, that even a classifier that only predicts 0 will have an accuracy rating of over 90%, misleading observers that it performs wells in its task.

To correctly benchmark a forecasting algorithm that deals with the imbalance of my data, I need give a greater focus on precision, recall and f-score. In particular my goal is to detect as many of the real hotspots as possible, maintaining an accuracy rating over 90%. This allows the predictor to accept precision as low as 10% if the recall rating reaches values of 90% or above, similarly to cancer detection, where a false negative might cost much more than a 10 or 20 false positives. By monitoring the above performance metrics, the hyper-parameters can be optimized and finally achieve and optimal outcome.

4.3.2 Detection of temporal dependencies

The first operation executed in the model is capturing the temporal information. In doing so an RNN type network has to be deployed. As previously mentioned in the theoretical background section, there are two well performing RNNs the LSTM and the GRU. Out of the two I picked the Gated Recurrent Unit as the models module. The reason being that GRU architecture is more lightweight than the LSTM memory wise, while also being slightly faster and similarly performing [42]. Although LSTMs are known to perform well when with a large historical input, in my case the time complexity to include longer historical buffers is extremely big. In addition, as shown in the later results section, the LSTM baseline under-performs against the contemporary GRU baseline using this data-set.

The temporal part of the model consists of N GRU layers, where N is the number of historical hours that are given as input. As it is going to be discussed in the optimization section, the optimal value of layers in

this work was 24 hours of history. The output of the GRU has the form of a single hour of observations per antenna.

4.3.3 Detection of spatial patterns

The next step is to capture the spatial dependencies and produce the output. To achieve this, a graph based neural network is needed. The prime candidates are Graph Convolutional Networks and Graph Attention Networks. The main difference between GAT and GCN is the fact that GAT have learnable weights for the edges that allow them to capture more complex relations between the neighbouring nodes. This makes them more effective than GCN in certain use cases, with the disadvantage of being more resource heavy. This fact, with the addition of effective examples of GCN deployments in traffic prediction [30], make the Graph Convolutions more compatible with my use case.

It is important to add here that the effectiveness of the graph based networks relies upon the graph connections. This is also the reason that I need to try multiple neighbour logic definitions to receive an optimal outcome. GCNs take advantage of spatial information using a step called message passing. In this step a node retrieves the information vector of all neighbours and averages them before continuing with the convolution. Having too many neighbours can affect negatively the outcome of message passing. On the other hand, not including key neighbouring antennas will devoid the network of spatial information. On Figure 4.4 an example of message passing is shown.

4.3.4 Activation function choices

As it is natural in any neural network, the application of non-linearity has to be included by adding activation functions between the hidden layers. One of the more common and effective activation function is RELU. This function only allows positive values to flow from layer to layer, while nullifying negative ones. Usually it is the default choice for most works.

This work though is a special case. I observed an increased number of negative values in all of the models layers. This creates the danger of

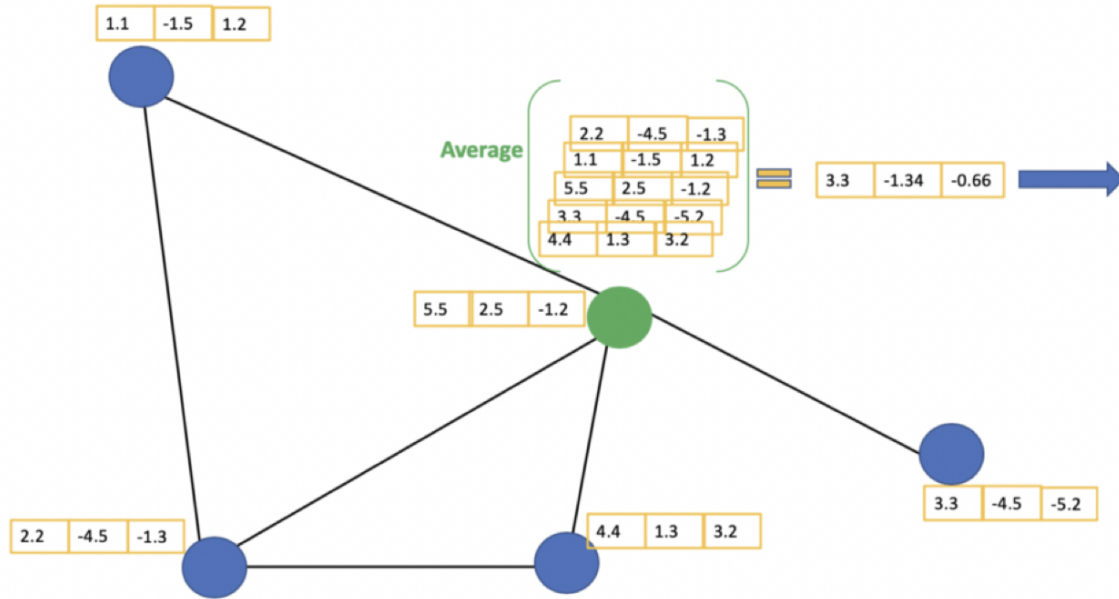


Figure 4.4: Simple example of message passing on a target node. [13]

vanishing gradients when nullifying those outputs. Thus I decide to use a version of RELU called LeakyRELU. What this function does is allow a down-scaled negative output to enter the next layer. Using it I avoid the problem of vanishing gradients, while observing a jump in performance in said model.

Finally for the output has to take the form of the probability of an antenna becoming a hotspot. To achieve this outcome a sigmoid activation function is being placed before the final output of the model. This function transform all outputs, giving them a value range between zero and one. This provides the model the probability of each antenna becoming a hotspot. A copy of the final output decides non hotspot for values lower than the decision boundary of 0.5, otherwise classifies an antenna as hotspot. Table 4.2 contains some information about the used activation functions.

4.3.5 Loss function choices

Finally I need to choose a loss function compatible to this problem. Since i am dealing with a binary prediction, I consider using binary cross

Name	Function	Derivative
Sigmoid	$\phi(x) = \frac{1}{1 + e^{-x}}$	$\phi'(x) = \phi(x)(1 - \phi(x))$
ReLU	$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$
Leaky ReLU	$\phi(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} \alpha & x \leq 0 \\ 1 & x > 0 \end{cases}$

Table 4.2: Activation functions used and their derivatives.

entropy. Binary cross entropy , or log loss, calculates how much a binary classification deviates from the real values using the probability that my model produces as output.

A variation of this function called binary cross entropy with logits allows me to add a class weight and a re-scaling factor to the loss calculation. The class weight helps in dealing with the crippling class imbalance that exists in the data , while the re-scaling factor corrects the loss so that the optimizer will not make drastic changes to the models weights between epochs. An additional benefit of the class weight is that they allow the predictor to increase the prediction recall , with a minor trade-off on precision.

4.4 Optimization and hyper-parameter tuning

Optimizing the model is the final essential task in designing a functional machine learning algorithm. Here it is required to test and find the optimal parameters that increase the overall performance. Those parameters are the total number of layers, neurons, how many epochs to train, which graph logic to use, what are the optimal class weights etc.

4.4.1 Optimizer and learning rate

In order for the model to train it is essential to include some kind of back propagation algorithms for optimizing the networks weights. To do that I am going to use the well known Adam optimizer implementation of the library Pytorch.

Adam is an extension of stochastic gradient descent. It has been used in most recent deep learning (DL) applications thanks to its high computational efficiency and low memory usage. The learning rate in which the optimizer will default is 0.001. The strategy behind that number being the initialization of the weight optimization process with a small deviation, in order to avoid losing local optima that may exist close to the initial value of the weights. Thanks to the functions of Adam, the starting learning rate doesn't have to be manually adjusted, since the algorithm itself will fine-tune it later on, depending on the output gradients and the loss it receives on the backwards call of the model.

4.4.2 Loss weights

Lastly it is important to configure the loss weights in order to counter the class imbalance. Here, the minority class that the model tries to forecast has a ratio close to 1 to 10.000, making it extremely hard to predict. If the model is left without a loss weight, then it will almost always predict the majority class, making the classifier practically useless. On the other hand a very high class weight will have the model classify every outcome to the minority class. In order to refine it I used a number of combinations of class weights. Class weights with lower value than one do not contribute into finding the minority class, and thus are excluded. Weights above one give more attention to antennas that are hotspots, but have a trade-off between recall and precision. Increasing the weight provides better recall, but at the same time sacrifices precision, especially when the weight was far above the value of 2. There the drop in precision translated to significant drop in accuracy as well. The optimal value turned up to be two. Using this weight the model achieves high recall with precision over 10%, maintaining high accuracy as well. It is also important to use a re-scaling weight of 0.9 in order to let the optimizer make smaller steps during the weight optimization. This helps avoiding

the loss of some local optima that may appear.

4.4.3 GCN and GRU configuration

The first thing that is going to be optimized is the GRU. In this part it is vital to figure out what number of N historical values will provide the best result with a relatively fast run-time. As it is natural, increasing the value of N translates to a linear increase in time requirements. I tested a variety of different historical values ranging from 1 to 48 hours. I observed that before the value of 24, increasing the historical value also gave better performance. After the value of 24, as shown in Table 4.3, GRU returns similar results between experiments. Since the improvement was marginal with historical values greater than 24 hours, I keep 24 GRU layers, having the optimal performance to run-time trade-off for this work.

GRU performance				
Historical buffer size	Accuracy	Precision	Recall	Run-time
1	97%	12%	52%	15min
12	97%	15%	78%	90min
24	98%	20%	79%	180min
36	98%	18%	80%	270min
48	98%	19%	79%	360min

Table 4.3: GRU performance using prediction horizon of 12 hours ahead and class weights with the target of reaching high recall.

For the GCN part, the number of layers allows the network to get information from nodes that are further away. Each layer act similar to collecting information from 1 step distances on further nodes. Going too far away will provide the network with too much information, not only making it more resource hungry, but also diluting the useful information so much, to a point that the model wont be able to make any predictions. On the other hand using a too shallow GCN will lose useful information from nearby nodes. Upon testing I found that the optimal number

node hops is three, as it appears in Table 4.4. Other than the number of layers, the GCN part has also a variable number of neurons. Although faster than the GRU, the GCN uses a much bigger memory space in the server, closely related to the number of neurons that the architecture uses. Since the project utilizes the full graph, more neurons allow it to learn more complex patterns and thus provide better predictions. But since there are no infinite resources, I had to limit the neurons to 256, which provided the best results, while leaving enough space in memory module to add the GRU module to the system. As exhibited in Table 4.4 the best combination for the GCN in this case is 3 layers with 256 neurons.

GCN performance					
Layers	Neurons	Accuracy	Precision	Recall	VRAM
1	256	98%	12%	38%	2 GB
2	256	96%	10%	45%	3 GB
3	256	97%	10%	50%	4 GB
4	256	96%	8%	40%	6 GB
3	64	99%	0.1%	1%	1 GB
3	128	98%	10%	30%	2 GB
3	512	96%	0.7%	25%	8 GB

Table 4.4: GCN performance using different layers and neurons. The neighbour logic used on the test was distance based with maximum neighbour distance 0.1 kilometers.

4.4.4 Epochs

Since I am dealing with a model with a large number of parameters, that accepts a very large input vector, it is important to have a way to find an optimal amount of epochs in which the model will train. In my case each epoch takes at best four hours to train. Also due to the vast amount of input data, I observe vast changes in the model loss and performance even one step ahead. To deal with this I implemented a standard early stopping method to stop the training time. Here the

model will stop training if it didn't deviate more than 5% loss-wise in the final five epochs.

4.4.5 Graph logic optimization and comparison

As part of the optimization step I need to figure out which of the three neighbour logic matches best to the performance goal. I train the model with different values in all three neighbour methods. Starting with the application of distance thresholds, I observe that increasing the distance parameter, not only increases the models complexity, but also decreases its performance. This goes to the fact that having too many neighbours in the message passing part of the GCN algorithm, in accordance to the class imbalance, dilates the useful information and leads to having inaccurate predictions. This method performed best when using very small distances while also using much less memory than the longer distance counterparts.

Next I try considering only neighbours on the same base station. This method vastly outperforms the distance based one, since it limits the graph to smaller neighbourhoods. The dilation problem still persists, since the typical number of antennas in each base station is more than 10.

Finally I apply a k-nearest neighbours algorithm as the graph logic. In this method I test the model with different k number of neighbours, limiting the dilation issues while still providing important information from the neighbouring cells. The best combination turned out to be using KNN with 5 neighbours, giving the best results compared to the other graphs that I constructed. The results of each method using GRU with 24 hours of history and three layers of GCN are expressed in Table 4.5.

Neighbours performance				
Criterion	Limit	Accuracy	Precision	Recall
Distance	0.05 km	96%	16%	70%
Distance	0.1 km	96%	15%	68%
Distance	0.2 km	98%	10%	45%
KNN	1	99%	22%	85%
KNN	5	99%	20%	95%
KNN	10	99%	15%	85%
Cell Site	-	98%	15%	75%

Table 4.5: GRU-GCN model performance using different neighbour logic.
The main parameters of the model are 24 hours of history, 3 GCN layers with 256 neurons and class weights optimized for recall.

5 Results

Reaching the end of the architecture design phase I have a final, highly optimized predictor. It is now turn to compare this models performance to a number of baseline ones that are being used in similar works, while also showcasing the performance of certain variations of it. I also investigate the performance of said designs when trying to achieve higher precision while trying to maintain the target recall.

5.1 Final model tests and comparison to baselines

The proposed GRU-GCN design consists of a Gated Recurrent Unit with 24 hour history buffer, 3 Graph Convolutional layers using 256 neurons and a neighbourhood derived from the 5 nearest neighbours of each node and is exhibited in Figure 4.3. The loss function used is binary cross entropy with logit loss, using a class weight of 2 and a scaling weight of 0.9. This configuration reached the highest recall while maintaining high accuracy and a good level of precision. A number of variations of the model were tested in order to provide a comparison to known methods. Those variations include using 1 less graph convolution in the process ,referred to as Variation 1 in the experiment tables, changing the final layer to a fully connected one as shown in the work AT-GCN [30] ,named Variation 2, and using a Long Short Term Memory unit instead of the Gated Recurrent one, named Variation 3. As baselines I also used an optimized 3 layer Graph Convolutional Network with similar configuration with the one that was embedded in the model, a Gated Recurrent Unit and a Long Short Term Memory based on the work of Zhou [15]. All configurations were tested for the prediction horizons of 12 and 24 hours ahead.

Models				
Model	Prediction Horizon	Accuracy	Precision	Recall
GRU-GCN	12 hours	99%	20%	95%
Variation 1	12 hours	99%	33%	88%
Variation 2	12 hours	99%	33%	88%
Variation 3	12 hours	99%	18%	73%
GCN	12 hours	97%	10%	50%
GRU	12 hours	98%	20%	79%
LSTM	12 hours	98%	22%	80%
GRU-GCN	24 hours	99%	15%	45%
Variation 1	24 hours	99%	11%	41%
Variation 2	24 hours	99%	16%	42%
Variation 3	24 hours	99%	18%	35%
GCN	24 hours	98%	13%	37%
GRU	24 hours	98%	20%	40%
LSTM	24 hours	98%	25%	41%

Table 5.1: Model and baseline performance with the target of high recall.

Models				
Model	Memory	Accuracy	Precision	Recall
Original set	4 GB	99%	20%	95%
A.E. set	0.8 GB	99%	18%	89%

Table 5.2: Original data versus Autoencoder output

As indicated in Table 5.1 the proposed model has a significantly better recall than all other models in both 12 and 24 hour predictions horizons. In the higher horizon of 24 hours I observe a performance drop in all cases, caused by the complex nature and size of the input data. I suspect that a partitioning of the graph into smaller ones and optimizing different training sessions with the said graphs may improve both short and long term predictions. Nevertheless I succeeded in reaching the goal of high recall using the full set of antennas.

A final experiment for this work would be training the model using the data produced by the autoencoder. The goal of doing this is lowering the resource requirements of the model as much as possible. It is expected to have a small loss in performance, as the compressed data contain noise, that is naturally created in the process of the autoencoder. As shown in table 5.2 training the model using this set of data has a small reduction in recall, but with a high reduction in memory usage. This gives the option of training even bigger graphs in cases where the graphs are so big that small drop of performance can be tolerated.

5.2 Class weights and precision/recall trade-off

A new, secondary goal is now switching the models target to increased precision while trying to maintain recall. To do this I need to re-optimize the class weights, in order to calibrate the value of the true positive compared to the true negative. I noticed in my tests with the class weights, that any weight higher than 1 returns a high recall in exchange to precision, while with lower values the predictor always returns 0 due to the high class imbalance. Since I now have an already trained model

Models with prediction horizon 12 hours				
Model	Weight	Accuracy	Precision	Recall
GRU-GCN	A	99%	40%	55%
Variation 1	A	99%	43%	45%
Variation 2	A	99%	42%	46%
Variation 3	A	99%	30%	49%
GCN	A	98%	5%	15%
GRU	A	99%	22%	40%
LSTM	A	99%	21%	40%
GRU-GCN	B	99%	65%	22%
Variation 1	B	99%	51%	25%
Variation 2	B	99%	55%	24%
Variation 3	B	99%	57%	19%
GCN	B	97%	9%	1%
GRU	B	98%	31%	12%
LSTM	B	98%	30%	14%

Table 5.3: Model and baseline performance with the target of higher precision at horizon of 12 hours ahead.

capable of finding true positives and having a high recall, I experiment continuing training it with a class weights smaller than two. I observed that after a small number of epochs the model reaches higher numbers of precision, as shown in Table 5.3. The highest precision is reached is 65%. The problem that exists is, that even with weights closer to the optimal value of two, the recall drops significantly. This is problematic, since the main goal was maintaining this high recall. On the other hand the maximum precision gained is not optimal enough to consider it a good result in case the network engineers requirements change to that.

6 Conclusions

After a significant amount of testing, optimizing and comparing the models I made a number of important observations. In this chapter I will showcase those observations while also proposing ideas for further improving the proposed models performance and architecture.

6.1 General observations

I am going to begin with the observations made during the architecture design with the goal of high recall. There, from the results of the baseline tests using only a recurrent unit, both LSTM and GRU have a fairly close yet worse performance to the proposal, meaning that it is actually benefiting from spatial information using an embedded GCN module. The memory buffer in the said recurrent unit does not offer a significant difference in performance when using higher history lengths than 24 hours, and as such 24 is a preferred value. The two factors which impacted the model the most were the neighbour logic used in the GCN part and the class weights of the loss function.

Regarding the neighbour logic, it is obvious using a high number of neighbours inserts too much noise in the computation, resulting in very low performance. Thus limiting the neighbours and only taking into consideration the closest ones provides just enough information to have a gain in performance, while also requiring lower resources since less matrices are being aggregated on the message passing of the GCN.

Class weights affected how well the model fitted to the research goal. They dictated how important the true positives were compared to false positives. In the case of recall, a weight with the value of two was high enough to let the model find most true hotspot cases, while reaching an acceptable number of false hotspots. Changing the weight into a value closer to 1 lets the model focus on finding mostly true positives, but

due to the imbalance only classifies everything as non hotspot. Here it was found that retraining an already trained model with a different class weight helped tackling the imbalance and achieving higher precision, but decreased the wanted recall.

An important observation that also made during testing is , that most of the model's run-time and performance problems derive from the size of the graph. I suspect that the inclusion of so many different antennas into the same training and prediction cycle makes the problem too complex. The behavioural patterns are just too many. This also contributes to the high dilation of the GCN aggregation when a using different neighbour logic to KNN, and to the long run-time and high resource usage.

Finally the introduction of feature extraction using an Autoencoder reduces the resource usage of the model, with a minor performance penalty. This allows running the predictor using bigger graphs, although as concluded this is not optimal due to a minor drop in performance.

6.2 Thesis conclusion

In this thesis I proposed a machine learning model capable of predicting long term performance drops in cellular antennas. The model takes advantage of both spatial and temporal information using an ensemble of GCN and GRU networks and achieves forecasting with high accuracy and recall. It is also capable of handling sizable graphs and outperforming state of the art designs in such cases.

In the goal of higher precision the GRU-GCN model achieves acceptable results on 65% , maintaining a recall of 22% and high accuracy, significantly dropping recall. In the final segment of the conclusion I provide ideas on gaining better results regarding that goal.

6.3 Alternative implementations/Future plans

Following the conclusion there are some interesting ideas that can further improve the proposed model. As it was observed the origin of most problems is the size of the graph. Including a large number of antennas ,that vary in behaviour, into the same predictions makes the problem

too complex and heavyweight. I suspect that partitioning the graph into sub-graphs will allow the model to capture both spatial and temporal dependencies faster. Partitioning could perhaps assist addressing the issue of the low positive class ratios. This will give the forecaster better performance and possibly allow prediction in greater prediction horizons.

The addition of the sub-graphs will also make the model much lighter in resource requirements. This will allow running it on lightweight computing units such as Field Programmable Gate Arrays (FPGAs). FPGAs have the advantage of being low cost, needing low energy consumption, while also potentially providing performance increases in cases similar to ours. They will also allow mobile service providers to predict performance drops in the edge in each individual antenna.

Bibliography

- [1] Joseph Yacim and Douw Boshoff. Impact of artificial neural networks training algorithms on accurate prediction of property values. *Journal of Real Estate Research*, 40:375–418, 11 2018.
- [2] Hirokijyoti Kalita, Adithi Krishnaprasad, Nitin Choudhary, and Sonali Chen, Das. Artificial neuron using vertical mos2/graphene threshold switching memristors. *Scientific Reports*, 2019.
- [3] Linear activation function. <https://iq.opengenus.org/linear-activation-function/>. Accessed: 2022:12:01.
- [4] Fan Cheng, Chen Meiling, and Wang Xinghua. A review on data preprocessing techniques toward efficient and reliable knowledge discovery from building operational data. https://www.frontiersin.org/articles/10.3389/fenrg.2021.652801/full?fbclid=IwAR0wGACYLg_vvy9FX3wPqximu9gvh3NVU49yJWeaKpKWGtj0F7y2DFf6jsU. Accessed: 2022:12:01.
- [5] Convolutional neural networks. <https://paperswithcode.com/methods/category/convolutional-neural-networks>. Accessed: 2022:12:01.
- [6] Graph convolutional networks. <https://tkipf.github.io/graph-convolutional-networks/>. Accessed: 2022:12:01.
- [7] Recurrent neural network tutorial. <https://www.datacamp.com/tutorial/tutorial-for-recurrent-neural-network>. Accessed: 2022:12:01.
- [8] Long short-term memory (lstm). https://d2l.ai/chapter_recurrent-modern/lstm.html. Accessed: 2022:12:01.

- [9] Gated recurrent units (gru). https://d2l.ai/chapter_recurrent-modern/gru.html. Accessed: 2022:12:01.
- [10] Applied deep learning - part 3: Autoencoders. <https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798>. Accessed: 2022:12:01.
- [11] How to handle missing data. <https://towardsdatascience.com/how-to-handle-missing-data-b557c9e82fa0>. Accessed: 2022:12:01.
- [12] K nearest neighbor : Step by step tutorial. <https://www.listendata.com/2017/12/k-nearest-neighbor-step-by-step-tutorial.html>. Accessed: 2022:12:01.
- [13] Recent advances in graph convolutional network (gcn). <https://towardsdatascience.com/recent-advances-in-graph-convolutional-network-gcn-9166b27969e5>. Accessed: 2022:12:01.
- [14] Joan Serrà, Ilias Leontiadis, Alexandros Karatzoglou, and Konstantina Papagiannaki. Hot or not? forecasting cellular network hot spots using sector performance indicators. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 259–270, 2017.
- [15] Lixia Zhou, Xia Chen, Runsha Dong, and Shan Yang. Hotspots prediction based on lstm neural network for cellular networks. *Journal of Physics: Conference Series*, 1624:052016, 10 2020.
- [16] Wei-Lun Chao. Machine learning tutorial. 2012.
- [17] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, 1996.
- [18] Massimo Buscema. Back propagation neural networks. *Substance use and misuse*, 33:233–70, 02 1998.

- [19] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.
- [20] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. A comprehensive survey and performance analysis of activation functions in deep learning. *CoRR*, abs/2109.14545, 2021.
- [21] Cheng Fan, Chen Wang, Jiayuan Wang, and Bufu Huang. A review on data preprocessing techniques toward efficient and reliable knowledge discovery from building operational data.
- [22] David M. W. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *CoRR*, abs/2010.16061, 2020.
- [23] Brian Liu and Madeleine Udell. Impact of accuracy on model interpretations. *CoRR*, abs/2011.09903, 2020.
- [24] Keiron O’Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.
- [25] Zhen Zhao, Ze Li, Fuxin Li, and Yang Liu. Cnn-lstm based traffic prediction using spatial-temporal features. *Journal of Physics: Conference Series*, 2037(1):012065, sep 2021.
- [26] Weiqing Zhuang and Yongbo Cao. Short-term traffic flow prediction based on cnn-bilstm with multicomponent information. *Applied Sciences*, 12(17), 2022.
- [27] Di YANG, Songjiang LI, Zhou PENG, Peng WANG, Junhui WANG, and Huamin YANG. Mf-cnn: Traffic flow prediction using convolutional neural network and multi-features fusion. *IEICE Transactions on Information and Systems*, E102.D(8):1526–1536, 2019.
- [28] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. 12 2013.

- [29] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015.
- [30] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-GCN: A temporal graph convolutional network for traffic prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9):3848–3858, sep 2020.
- [31] Zhiyong Cui, Kristian Henrickson, Ruimin Ke, Ziyuan Pu, and Yin-hai Wang. Traffic graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting, 2018.
- [32] Xiaolei Ma, Zhimin Tao, Yinhai Wang, Haiyang Yu, and Yunpeng Wang. Long short-term memory neural network for traffic speed prediction using remote microwave sensor data. *Transportation Research Part C: Emerging Technologies*, 54:187–197, 2015.
- [33] Yanjie Duan, Yisheng L.V., and Fei-Yue Wang. Travel time prediction with lstm neural network. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1053–1058, 2016.
- [34] Zheng Zhao, Weihai Chen, Xingming Wu, Peter C. Y. Chen, and Jingmeng Liu. Lstm network: a deep learning approach for short-term traffic forecast. *IET Intelligent Transport Systems*, 11(2):68–75, 2017.
- [35] Yisheng Lv, Yanjie Duan, Wenwen Kang, Zhengxi Li, and Fei-Yue Wang. Traffic flow prediction with big data: A deep learning approach. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):865–873, 2015.
- [36] Rui Fu, Zuo Zhang, and Li Li. Using lstm and gru neural network methods for traffic flow prediction. In *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 324–328, 2016.

- [37] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *CoRR*, abs/2003.05991, 2020.
- [38] Binghao Yan and Guodong Han. Effective feature extraction via stacked sparse autoencoder to improve intrusion detection system. *IEEE Access*, 6:1–1, 07 2018.
- [39] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, page 1096–1103, New York, NY, USA, 2008. Association for Computing Machinery.
- [40] Qiang Xu, Sanjeev Mehrotra, Z. Morley Mao, and Jin Li. Proteus: Network performance forecast for real-time, interactive mobile applications. In *MobiSys 2013- ACM International Conference in Mobile Systems, Applications, and Services (MobiSys)*, June 2013.
- [41] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional neural network: A deep learning framework for traffic forecasting. *CoRR*, abs/1709.04875, 2017.
- [42] Pawan Kumar Sarika. *Comparing LSTM and GRU for Multiclass Sentiment Analysis of Movie Reviews*. PhD thesis, 2020.