uc3m | Universidad Carlos III de Madrid

University Degree in Sound and Image Engineering

Academic Year 2019-2020

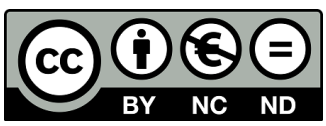
Bachelor Thesis

"Augmenting SNR Datasets through Advanced Learning Techniques"

Mario Rodríguez Ibáñez

Marco Gramaglia

Leganés, Madrid. 6th July 2020



ABSTRACT

Advanced radio scheduling techniques for highly softwarized network paradigms, such as C-RAN, will make use of Artificial Intelligence techniques to make short time scale predictions of UE SNR levels. However, this may scale badly: keeping tracks of individual user behaviors in a macrocell with hundreds of users and to individual prediction is a complex task. This work analyzes how to enrich the SNR datasets through advanced learning techniques to allow an optimal analysis of the SNR properties even when the available samples are much lower than the real ones, eventually leading to more scalable algorithms.

Keywords: GAN, mobile networks datasets, time series generation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

To satisfy the demands of 5G systems, there is a tendency to Network Functions Virtualization (NFV) and Software Defined Networks (SDN) [1]. The integration of Artificial Intelligence (AI) in this paradigms, such as C-RAN [2], will make use of Artificial Intelligence techniques to make short time scale predictions of UE SNR levels, among other channel state parameters. However, it can be challenging to properly train the AI models without sufficiently large and representative datasets. This work analyzes how to enlarge said datasets through advanced learning techniques, allowing an optimal analysis of the SNR properties even when the available samples are much lower than the real ones, eventually leading to more scalable algorithms.

1.2. Objective

The aim of this work is the augmentation of the actual empirical mobile networks datasets, by generating samples through advanced learning algorithms, in such a way that the new data is as representative as the original. This shall be achieved by the utilization of Generative Adversarial Networks, that shapes random noise to approximate the real data.

1.3. Summary

As many other Machine Learning related work, the first step is the collection of data. This is crucial as the whole experiment is therefore biased after this initial data. In this case, the data collection consists in capturing the parameters of real traffic in a testbed using srsLTE, configured as different scenarios. These are the combination of three different MGEN traffic patterns, three bitrate configurations, three levels of SNR, and three kinds of evolution of SNR levels.

The next step is the validation of the collected data. This consists in the visual repre-

sentation of the different parameters in every scenario to observe the possible correlation between them. Additionally, outliers are identified in this step through the analysis of the results.

After validating the data the next procedure is the generation of synthetic data. For that purpose a Wasserstein Generative Adversarial Network (WGAN) is used. This network is composed by two modules: generator and critic, that (in this case) can either be Recurrent Neural Networks (RNNs) or causal Convolutional Neural Networks (CNN). Best performance is achieved by the combination of a CNN as generator and a CNN as discriminator.

So that the WGAN does not converge on sub-optimal solutions, the algorithm is modified: whenever the difference between synthetic data and real data is below a certain boundary the gradient clip parameter is incremented. The logic under this modification is that if the critic cannot distinguish between one an other, it should learn faster but not so much that the generator cannot learn.

In successful configurations the obtained synthetic data presents similar root mean square and peak to average ratio to the real data. Therefore it is possible to say that, with some limitations, the synthetic data could be useful.

1.4. Socioeconomic context

Artificial intelligence[1] potentially can have a major impact on economy [4]. The McKinsey Global Institute predicts that, by 2030, 70% of companies would have implemented some artificial intelligence technology, implying a growth in the global GDP of the 16%. They also claim that the potential impact of these technologies is comparable to other "general-purpose technologies", as the steam engine or electricity. However, the publication [5], highlights the scepticism about that possible large scale impact of this "new" technologies.

In [3], the value of deep learning applications in use cases is sized, saying that in telecommunications, there is a potential incremental value of artificial intelligence over

[1]Note: The considerations as artificial intelligence in this work consist of: Deep learning neural networks (e.g., feed forward neural networks, CNNs, RNNs, GANs), Transfer learning, and Reinforcement learning techniques. As it is considered in [3]

other analytics techniques of the 44%. Mobile networking companies, such as Nokia, are already offering cloud computing services to provide artificial intelligence solutions [6]. Although it has not been until recent years that artificial intelligence techniques have started to be applied to telecommunications, there is a tendency of Communication Service Providers (CSPs) adopting artificial intelligence into their networks [7]. This can be due to the fact that there is a prediction of a beneficial return of investment for early adopters of this technology. Other motivation for CSPs is that in order to accomplish the requisites of 5G systems (capacity, low latency, consistency...), the network traffic has to be managed effectively [7].

As addressed in [7], one of the challenges of the implementation of artificial intelligence is the "collection, structuring and analysis of data" to maximize the potential of this technology. CSPs are concerned about having "an excess of data from too many sources [...] and lack of single ownership or oversight of the data". Thus, the motivation of this work is solving the second concern by augmenting the datasets of individual users.

The equipment used for this work belongs to Departamento de Ingeniería Telemática of the Universidad Carlos III de Madrid, therefore it was not bought specifically for the project. In the case of not having said equipment, in the Table 1.1 an approximated budget is described. All the software used is free of charge, and consequently not included in the budget.

Item	Quantity	Unit price	Total
USRP B210 SDR Kit	2	1,321.00 €	2,642.00 €
Intel NUC NUC7I5BNH Intel Core i5-7260U	2	403.34 €	806.68 €
Total budget:			3,448.68 €

Table 1.1. EQUIPMENT PRICE LIST

1.5. Regulatory framework

Most of the software used in this work (srsLTE, MGEN, iPerf, Python, and PyTorch among others) are free and Open Source licensed [8]. "Free" meaning that "users have the freedom to run, copy, distribute, study, change and improve" said software, as de-

fined in [9]. It is specially beneficial because in some cases, in order to solve difficulties, modifications are required. For example, srsLTE does not provide the feature of exporting metrics to a ".csv" file, therefore, for this study, a modified version of the software (created for [10]) was used. In the case of the software used in this work, they are also free of charge, which is ideal for the limited budget of the project.

On the other hand, some disadvantages related to the use of this kind of software can emerge during the development of the work [11]. This include the compatibility issues, and the misguided documentation due to asynchronous communication. For example, the last release of PyTorch does not support the use of cuda devices in built-in LSTM models, and, although in the issues list of the project seems to be plenty of solutions, they are all deprecated.

Other software used in this study are not open source nor "free software", although they are free to charge. To run all the Python code related with the Machine Learning algorithms, the hosted Jupyter notebook service Google Colab was used. This option is chosen over the open source one (local Jupyter notebook) because that proprietary software offers the possibility of using cloud resources (GPUs, TPUs), consequently, accelerating the training process.

To create the tcpdump binary file needed for the MGEN clone pattern, a video-call using the application Skype was monitored. According to the point (a) section 6.3 of [12], it is prohibited to monitor any communication not intended to the user, but since both user subscriptions are in ownership, it is allowed. Likewise, Google Hangouts does not forbid such practice.

2. DATA COLLECTION

Retrieving data from a LTE base station is a rather trivial process in a real scenario, but having access and permits to them is not that much. And even if they are accessible, the data will correspond to multiple users and terminals, so keeping track of a particular one is a costly procedure [7].

Moreover, datasets of controlled metrics of individual users are scarce . Thus, for the purpose of this project, original datasets are generated. The metrics of interest in this datasets are the signal to noise ratio (SNR), the modulation coding scheme (MCS), the decoding time, and the buffer state report (BSR). Works such as [13] or [14] address the relevance of this parameters in softwarized RAN scheduling.

The SNR in this study is actually the uplink signal to interference plus noise ratio (SINR), according to the metrics reported by srsLTE. It is an indicator of the channel quality, and, therefore, has implications in scheduling algorithms [15].

Although this work does not aim to generate synthetic MCS values, it is contemplated as a relevant parameter given its relation with the other metrics. The reason why it is not generated is that it chosen with respect to the channel quality indicators (CQI), which depends on the SNR. Nonetheless, it is used in the data validation section for visualization.

The decoding time is the interval that the eNodeB needs to decode the information sent by the UE. Hence, it is one of the most time consuming tasks, it is crucial in RAN scheduling decision making [14].

The BSR indicates how much data is in the UE buffer ready to be sent to the network. This information is used by the eNodeB to make memory allocation decisions and assign resources to the UE in C-RAN systems [16], which makes the metric convenient to be reproduced.

2.1. Devices and configuration

The available testbed for this study is based on the open source LTE library srsLTE, which provides a fully working Software-Defined Radio User Equipment (srsUE), an eNodeB (srsENB), and an Evolved Packet Core (srsEPC) [17]. Although Software-Defined Radio is not an optimal solution in terms of performance [18] it is still a good approach as the bitrate is adapted to stay below the hardware limitations, and delay is not a critic parameter in this scenario.

The testbed consists in two different Ubuntu 16.04.6 LTS devices, which, both have srsLTE installed, and their network cards are connected by a RF cable. One of the devices runs srsUE to work as a user's terminal, and the other device runs both srsENB and srsEPC to work as the rest of the LTE network. In this work the data is collected in the second device as the scope of the study only covers the uplink parameters.

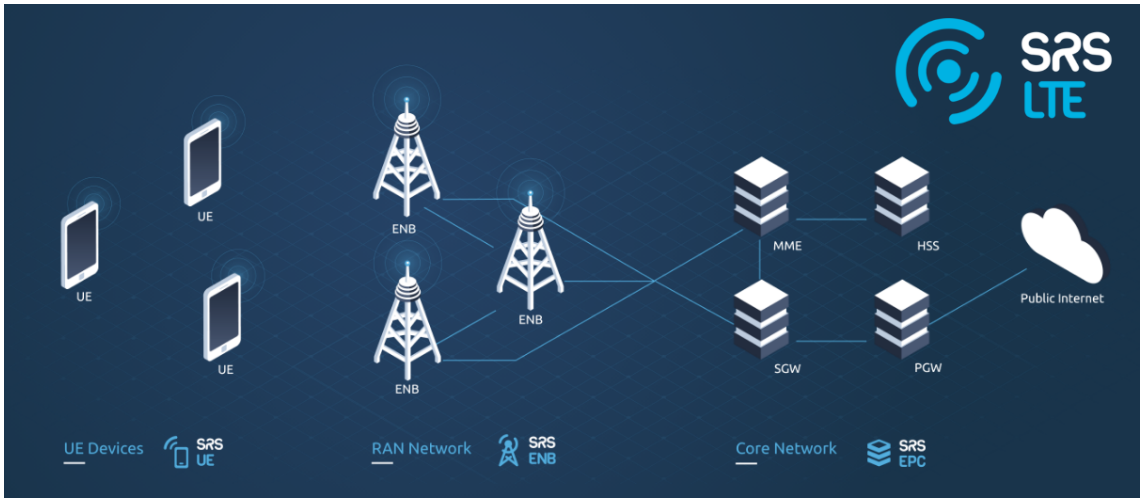


Fig. 2.1. srsLTE architecture diagram [19]

The srsUE is configured to the default values excluding the downlink E-UTRA Absolute Radio Frequency Channel Number (EARFCN) that is set to 3600 (Hz) so it is in the same channel than the eNB. The IP address, among many other parameters, is automatically assigned when connecting to the srsENB.

Analogously to the UE, the srsENB is configured to the default values, only changing the downlink EARFCN to 3600 (Hz).

The only modification to the srsEPC default configuration is to the SGi TUN interface

IP address, which by default is equal to the laboratory subnet, and therefore, can crash the connectivity of the device once the srsEPC is executed.

The version of srsLTE used in this work is the same as the one used for the paper [10]. That is due to the fact that for that study some modifications were made to the library that solve some difficulties applicable to this case. One of them is the exportation of data to comma separated variables (.csv) file, which is only possible in the vanilla version of srsLTE for the log information but not for the metrics. In the vrAIn version of srsLTE, it is possible to retrieve the information from the socket and copy it into the file by executing the Python code monitoring.py in appendix 3. That information is formed by the metrics of the link sampled every second. Consequently, the resolution of the data is then limited to one second. Additionally, this version allows to change the transmission gain dynamically, meaning, that in opposition to what the official srsLTE latest version documentation [19] says -the transmission gain is set before running the UE at the receiver- it is possible to change that value while the UE is running. The request of the transmission gain modification is send by executing the python code set_txgain.py in appendix 1. This request produces some anomalies in the metrics when there is not actual traffic being send at the moment. This implies that after the data collection phase, the outliers must identified for their later purge.

The open source software MGEN is used to generate traffic. It creates real time traffic patterns that can be configured as desired, as later is broadly discussed.

Another tool used is iPerf, which utility is to measure the maximum bandwidth and packet loss, among other parameters. It is therefore very useful to analyse the functioning of the testbed.

2.2. Testbed characterization

There is no detailed information about the behaviour of the testbed prior to this study. Accordingly, in order to work with this setup, it is necessary to understand its response when different kinds of traffic are transmitted.

The crucial characteristics to understand the testbed behaviour for this work are the maximum bandwidth and the performance of the signal to noise ratio at the receiver

with respect to the transmission gain set at the emitter.

Hence, this study only considers an uplink channel (and another downlink channel), and a given distance between the UE and eNB antennas, it is possible that the measured characteristics differ when this conditions change.

2.2.1. Maximum bandwidth

In order to know which is the maximum available bandwidth, the tool iPerf is used. The procedure to achieve that goal consists in executing the srsue command in the *computer 1*, and the srsepc and srsenb commands (in two different consoles) in the *computer 2*. After the UE is successfully attached and the connection is established, an iPerf UDP server is ran in another console in the *computer 2* and an iPerf UDP client is executed in the *computer 1*. In the mentioned iPerf client, the bitrate is sequentially set to increasing values every try. As the client bitrate increments, the received bitrate at the server reaches a boundary of approximately 15 Mbps. This implies that when the client bitrate is set to values above that boundary, the packet loss increases proportionally.

From that experience it is safe to say that the maximum achievable bandwidth is around 15 Mbps, and that the packet loss due to bandwidth overflow is negligible if the bitrate is set to values below 12 Mbps.

2.2.2. Mean SNR with respect to the transmission gain

Both computers of the testbed (with their respective peripherals) are static and always in similar conditions. Thus, for the signal to noise ratio (SNR) at the receiver to change, the transmission gain at the srsUE device has to be modified.

In this case the transmission gain can be set in the srsUE configuration file as it remains constant during the capturing periods, but by convenience it is set using the Python code fixed_txgain.py (appendix 2) before capturing.

As the srsLTE documentation [19] mentions, the range of acceptable values for the transmission gain is from 24 dB to 84 dB. In this test, for every unit value within that range, ten sequential fluxes that are 20 seconds long are sent. These fluxes are generated using iPerf (in the same configuration as described in section 2.2.1) and the client bitrate

is set to 12 Mbps. The traffic metrics are captured at the receiver (*computer 2*). After capturing the SNR at the receiver for every value of transmission gain and flux, the mean and variance for every value is computed and then plotted in the figure 2.2.

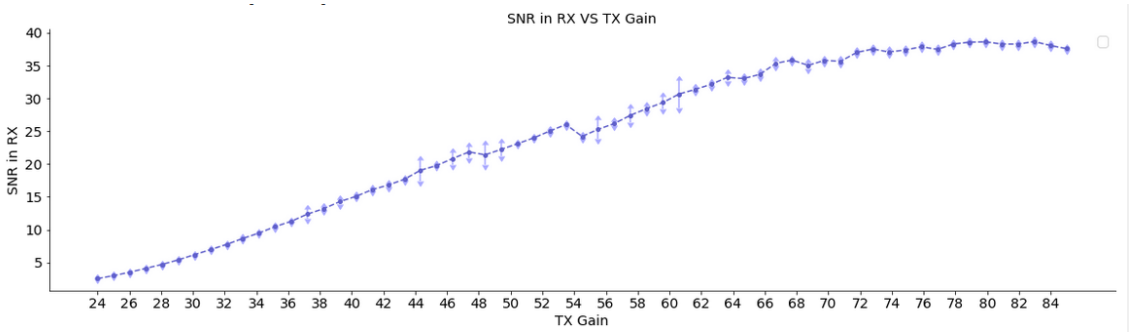


Fig. 2.2. Mean SNR at the receiver versus TX gain at the emitter

Following the same guidelines as in [10], it is possible to distinguish three different levels of SNR for srsLTE:

- Low: $< 14dB$
- Medium: $> 14dB$ and $< 23dB$
- High: $> 23dB$

Using the figure 2.2 as reference it is possible to conclude that each SNR level then corresponds to the transmission gains:

- Low: $24dB$ to $38dB$
- Medium: $39dB$ to $50dB$
- High: $51dB$ to $84dB$

2.3. Capturing scenarios

The ground truth data has to be diverse enough so that it is possible to check that the synthetic data can be generally valid. For that purpose, the real dataset is composed by a combination of eight traffic patterns and nine transmission gain patterns.

2.3.1. MGEN patterns

In order to resemble different traffic scenarios the iPerf tool is not sufficient as its options do not allow much personalizing. The alternative used for this purpose is MGEN traffic generator, which is an open source software and it is well documented [20].

This tool enables the creation and orchestration of different traffic patterns. In this work only a single flux is send every time, and the patterns used are the Poisson, Burst, and Clone patterns.

Poisson pattern

This pattern generates messages of a fixed sized and send them in a given interval that varies following a Poisson distribution. In this work the packet size is set to the maximum possible for UDP traffic: 8192 Bytes, and the sent packets per second is set to 31, 77, or 153, corresponding to 2 Mbps, 5 Mbps, or 10 Mbps, of uplink bitrate.

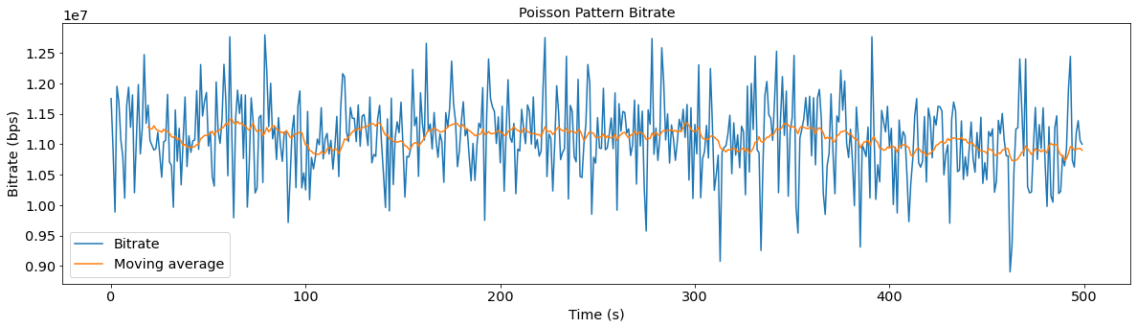


Fig. 2.3. Uplink bitrate versus time using Poisson pattern in 10 Mbps configuration and high fixed SNR

As it can be observed, in this configuration (Poisson pattern, with 10 Mbps, and high fixed SNR), the bitrate fluctuates around the 11 Mbps and does exceed the maximum bandwidth specified in section 2.2.1. The variance is not significant.

Burst pattern

As the name of this pattern announces, it generates bursts of messages of another kind of MGEN pattern at a given average time interval. The distribution of this time interval

can also be chosen as exponential or uniform. The bursts last a time interval that can also be chosen fixed or random. In this study the configuration used is: bursts appear following a exponential distribution with mean 5 seconds, the pattern of the bursts is a Poisson pattern with the same parameters explained in the previous point, and the duration of the bursts follows a exponential distribution with average on 3 seconds.

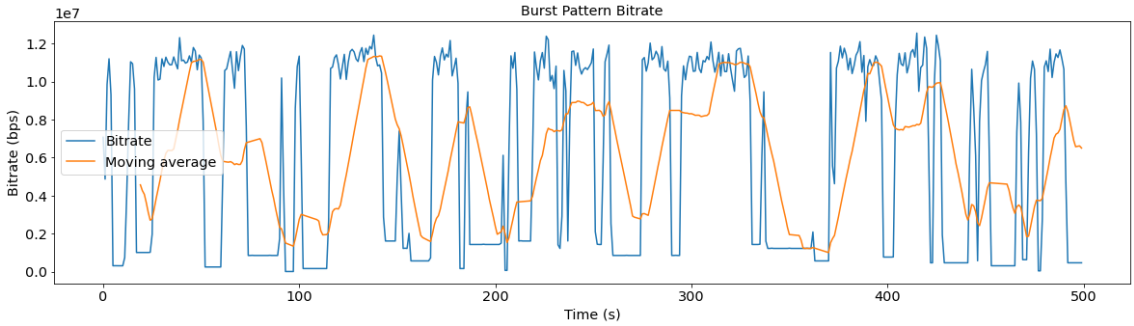


Fig. 2.4. Uplink bitrate versus time using Burst pattern in 10 Mbps configuration and high fixed SNR

The bitrate clearly peaks every few seconds when the bursts happen and afterwards it decays to zero. The variance is very notable.

Clone pattern

This pattern uses a tcpdump binary file and "clones" it, meaning that takes the packet sizes and its correspondent timestamps from the tcpdump file and generates a traffic with the same characteristics. In this work the tcpdump files are created using WinDump, which is the equivalent of tcpdump for windows, capturing the traffic of video-calls using two different applications: Skype and Hangouts.

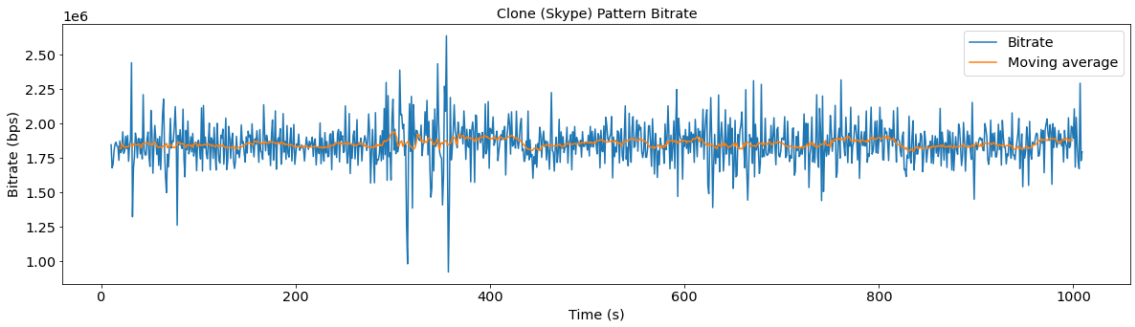


Fig. 2.5. Uplink bitrate versus time using Clone pattern with Skype and high fixed SNR

The bitrate behaves in a similar way to the Poisson pattern: it is generally around the same value (1.8 Mbps) with a not significant variance. But at some points it also presents some peaks of variation.

2.3.2. Transmission gain patterns

In a real scenario signal to noise levels may vary over time as the user terminal approximates to the base station or as they get farther away, interference may also not be stationary, or even the climatic conditions can effect. Hence, in this case the physical conditions of the testbed cannot be changed to simulate these real scenarios, the transmission gain is modified for that purpose.

The Python code used to modify the transmission gain following these patterns is in the file set_txgain.py in appendix 1.

Triangular waveform

This pattern tries to simulate the approaching and distancing of the UE and eNB. It does so by following the form of a stepped triangular wave. The transmission gain is initialized within the range of gain levels, and every 10 seconds the gain increases by $1dB$, until it reaches the maximum level of the range, in which case, starts to decrease by $1dB$ every 10 seconds, until it reaches the minimum value of the range. Then the process repeats until it is manually stopped.

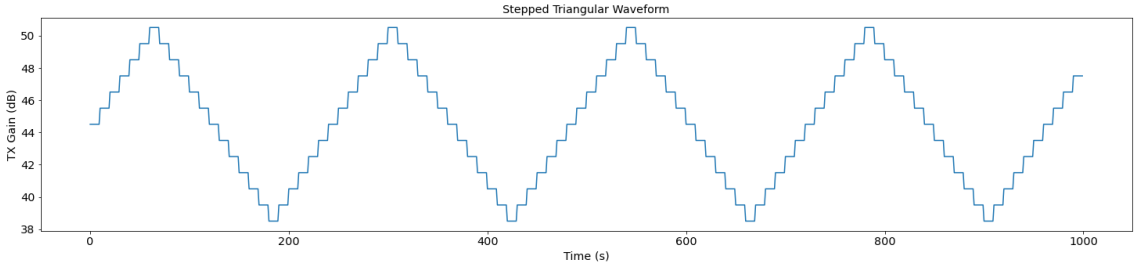


Fig. 2.6. Stepped triangular Waveform

Random steps waveform

This pattern emulates the sudden appearance (or disappearance) of interferences. It consists in, every 10 seconds, increasing the transmission gain by a integer value from $-5dB$ to $5dB$ that is generated randomly following a uniform distribution. If the new transmission gain exceeds the upper or lower boundary of the range it is set to that boundary.

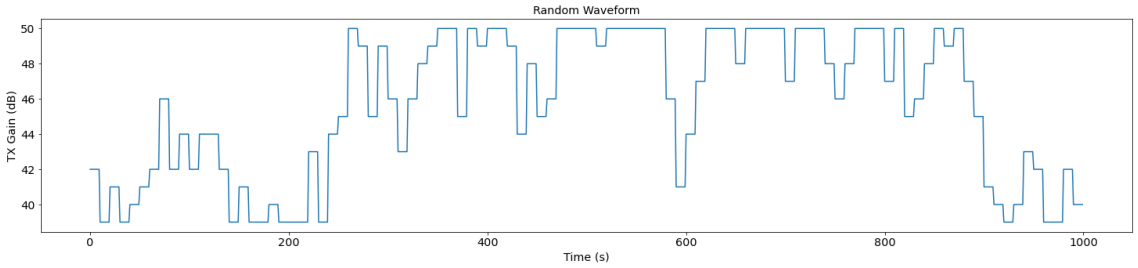


Fig. 2.7. Random Steps Waveform

Fixed gain

This simulates a more stationary scenario where the previously discussed conditions do not have a mayor impact on the SNR, meaning that the transmission gain is set before starting capturing, and remains at the same value until the end of it.

2.4. Capture procedure

The capture procedure consists of 4 different steps:

1. Connect srsLTE
2. Set transmission gain
3. Send MGEN traffic
4. Monitoring

To explain this process in a simpler manner the computers of the testbed are referred as *computer 1* and *computer 2*, where *computer 1* plays the role of UE and *computer 2* of the rest of the network (eNB and EPC).

2.4.1. Connect srsLTE

According to the guidelines in [19], the first step is running the srsEPC in *computer 2*: accessing the directory srsLTE/build/srsepc/src and executing the command:

```
        sudo ./srsepc epc_base.conf
```

This is to ensure that the version which the terminal is going to run is the "modified" one with the none default configuration file.

In another terminal in the *computer 2*, while srsEPC is running, srsENB is executed analogously to the srsEPC: accessing the directory srsLTE/build/srsenb/src and executing:

```
    $ sudo ./srsenb enb_base.conf
```

Then in *computer 1* the same steps are followed to run srsUE: access srsLte/build/srsue/src and then executing:

```
    $ sudo ./srsue ue.conf
```

It is then properly set if it shows in *computer 1*:

```
Waiting PHY to initialize...
...
Attaching UE...
Searching cell in DL EARFCN=3600, f_dl=940.0 MHz, f_ul=895.0 MHz
.
Found Cell:  Mode=FDD, PCI=1, PRB=50, Ports=1, CFO=-0.1 KHz
[INFO] [B200] Asking for clock rate 11.520000 MHz...
[INFO] [B200] Actually got clock rate 11.520000 MHz.
Found PLMN:  Id=00101, TAC=7
Random Access Transmission: seq=18, ra-rnti=0x2
RRC Connected
Random Access Complete.     c-rnti=0x46, ta=1
Network attach successful. IP: 172.10.0.19
Software Radio Systems LTE (srsLTE)
```

2.4.2. Set transmission gain

As mentioned before, to set the transmission gain the set_txgain.py (appendix 1) Python code is used in *computer 1*. The command which runs it must be contain two arguments: the SNR level mode, and transmission gain pattern. The SNR level mode argument can be either "l", "m", or "h", corresponding to low, medium, or high levels, and the pattern argument can be "trian" or "rand", which correspond to stepped triangular waveform or random waveform patterns.

For simplicity, the fixed pattern is set using another script: fixed_txgain.py in appendix 2, which needs of the same "l", "m", or "h", arguments that are equivalent to the aforementioned SNR levels.

2.4.3. Send MGEN traffic

The traffic is generated through the MGEN scripts poisson.mgn, burst.mgn, and clone.mgn, which are used in *computer 1*. These scripts correspond to the Poisson, Burst, and Clone patterns respectively. Following the MGEN documentation [20] the scripts are ran with:

```
$ sudo ./mgen input /[scrpit name].mgn
```

For the Poisson and Burst pattern cases, the values of packet sent per second of the scripts the must be changed depending of the desired approximate bitrate. 153 packets of 8192 Bytes per second will be equivalent to approximately 10 Mbps, 77 packets to 5 Mbps, and 31 packets to 2 Mbps.

For the Clone pattern it is necessary to write the route to the tcpdump binary file corresponding to Skype or Hangouts.

2.4.4. Monitoring

The command to save the metrics to a comma separated variable file can be executed last since the data that the socket retrieves belongs to the last 50 seconds. Therefore, if all commands are executed right away there is not any problem. Said commands are executed using the script monitoring.py (appendix 3) in *computer 2* and needs to contain

the route to the generated file. For example:

```
$ python monitoring.py>metrics_file.csv
```

The reason of only monitoring in *computer 2* is that the dataset of interest for this study only includes the uplink information. This is due to the fact that obtaining real downlink datasets of individual users is a trivial process: the metrics that a UE is receiving are the corresponding to that device. Thus, there is not actual use in augmenting downlink datasets, whilst there is for uplink datasets, as tracking data of single users in the eNodeB is much more difficult [7].

2.5. Capture results

The combination of all traffic patterns with transmission gain patterns results in 72 different measurements, and a total of 113836 samples, distributed as shown in the Table 2.1.

Number of samples of each combination										
		Triangular			Random			Fixed		
		Low	Med	High	Low	Med	High	Low	Med	High
Poisson	2 Mbps	339	337	339	340	328	324	3550	3550	3550
	5 Mbps	336	338	334	323	336	334	3550	3550	3550
	10 Mbps	331	322	331	334	320	323	3550	3550	3550
Burst	2 Mbps	323	373	333	335	330	326	3550	3550	3550
	5 Mbps	332	334	323	332	377	336	3550	3550	3550
	10 Mbps	340	382	323	336	327	327	3550	3550	3550
Clone	Skype	1845	1865	1846	1856	1856	1860	1863	1845	1840
	Hangouts	2361	2353	2345	2365	2364	2340	2375	2366	2343

Table 2.1. NUMBER OF SAMPLES OF EVERY PATTERN COMBINATION

The captures which use the pattern combinations with static levels of SNR (fixed SNR) have many more samples in comparison to the other captures. This may bias the

dataset to a rather simpler scenario, but it should be advantageous as the neural network may then find a solution faster.

3. DATA VALIDATION

Before working with the dataset to generate synthetic data, it is crucial to determine the possible correlation between the different parameters of the collected data as it can facilitate the task by generating parameters together, or in the absence of this correlation it can difficult the process.

In this case, the parameters of interest are mainly the signal to noise ratio (SNR), the buffer status report (BSR), the modulation and coding scheme (MCS), and the decoding time. These parameters are referred to the uplink, as are worthier of been synthetically generated than been captured, in contrast with the downlink case.

3.1. Signal to noise ratio analysis

The signal to noise ratio (SNR), and modulation and coding scheme are highly correlated by definition. The logic behind this is that the error corrector can assume a given error rate which is proportional to the MCS and inversely proportional to the SNR. Therefore, if the SNR is high then a MCS less robust (higher) can be used, as the error rate remains the same and the throughput increases.

However, as it can be observed in the figures 3.1, 3.2, and 3.3, the MCS values are quite similar for the medium and high SNR levels. This may happen because the MCS is in its maximum range of values when the SNR is at its medium values, so that when the SNR increases the MCS cannot stays the same.

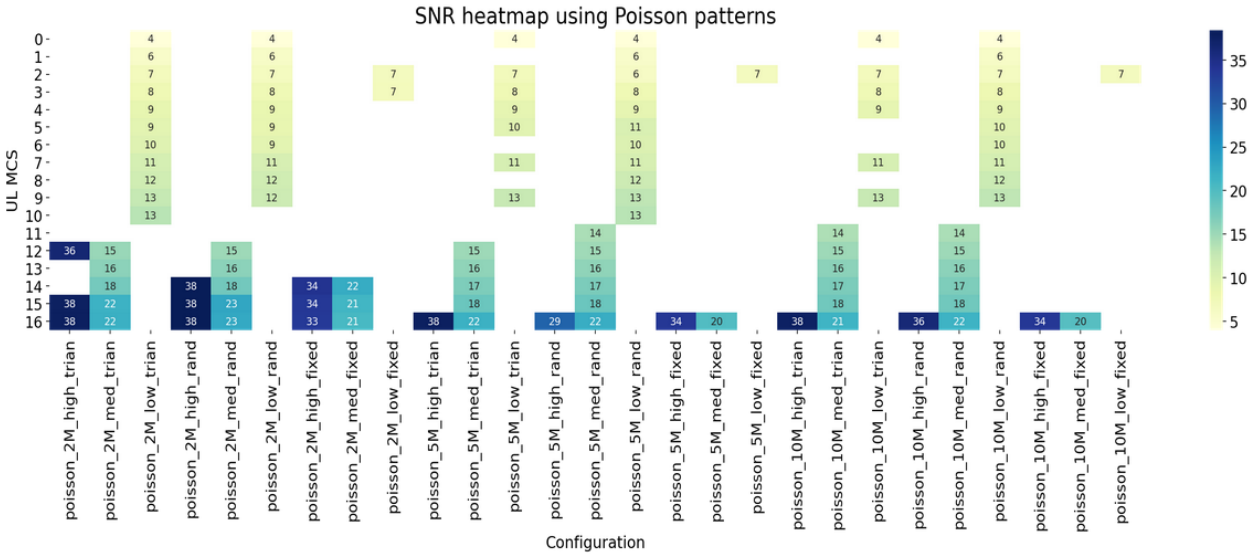


Fig. 3.1. SNR for Poisson patterns

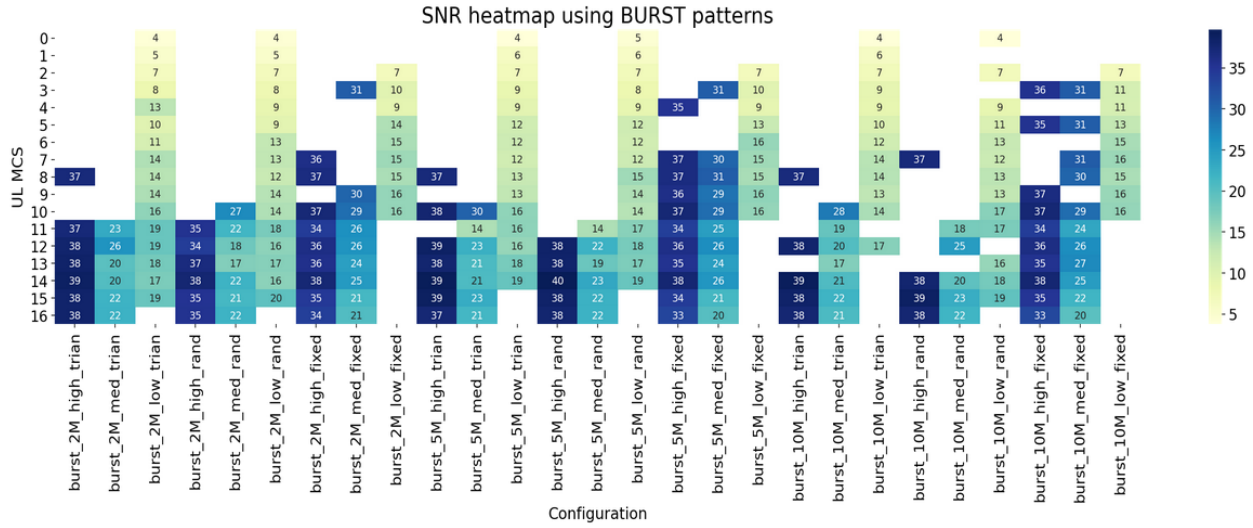


Fig. 3.2. SNR for Burst patterns

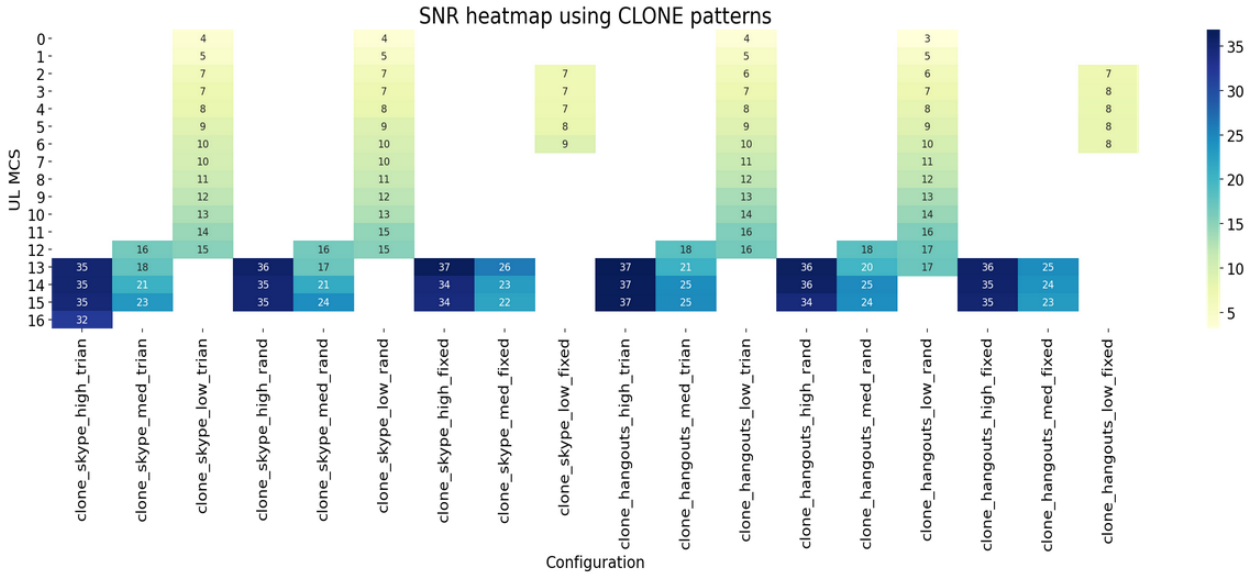


Fig. 3.3. SNR for Clone patterns

By seeing the heatmaps in figures 3.1, 3.2, and 3.3, it is possible to conclude that it can be easier to generate the SNR time series grouping them by SNR levels rather than by traffic patterns.

3.2. Buffer status report analysis

Observing the Figures 3.4, 3.5, and 3.6, it is possible to say that the buffer status report is higher for the patterns with 10 Mbps bitrate, as predictable. The more traffic sent, the more data pending to be sent will be in the UE buffers. Additionally, it can be seen that for low SNR levels the greater values of BSR are for Poisson patterns with low MCS, whilst for the medium and high SNR levels the greater values of BSR are for Burst and Poisson patterns with 10 Mbps of bitrate.

That relation may be a difficulty in order for the neural networks to learn, as it changes depending on the pattern, and would find a sub-optimal solution. Consequently, it may be useful to group the data by traffic pattern for the training.

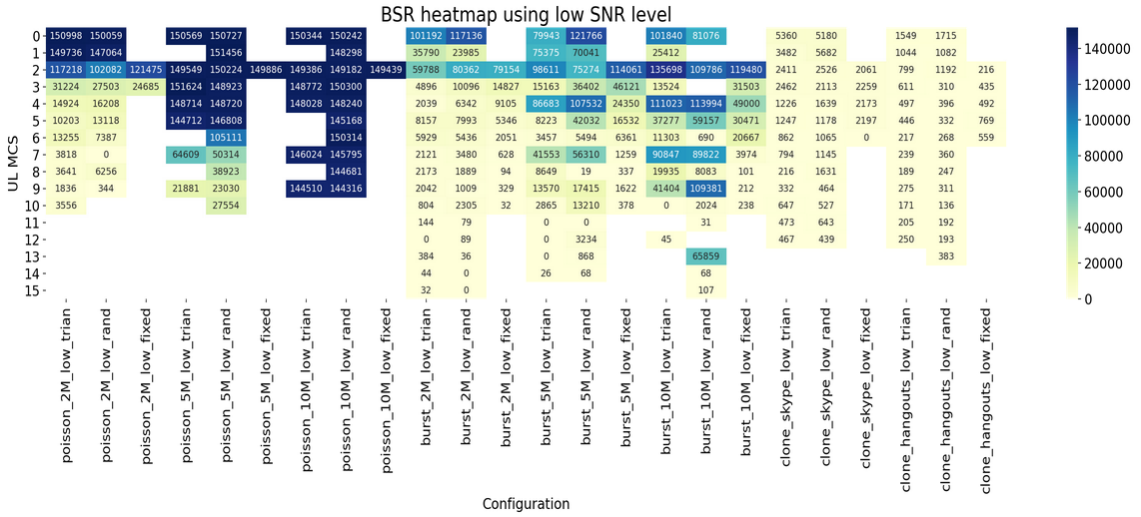


Fig. 3.4. BSR for low level SNR patterns

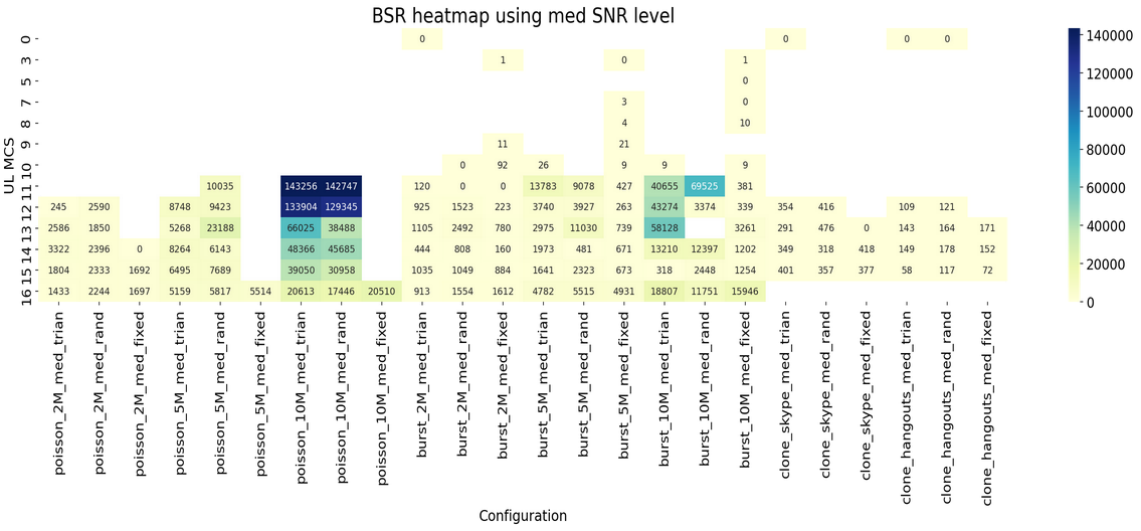


Fig. 3.5. BSR for medium level SNR patterns

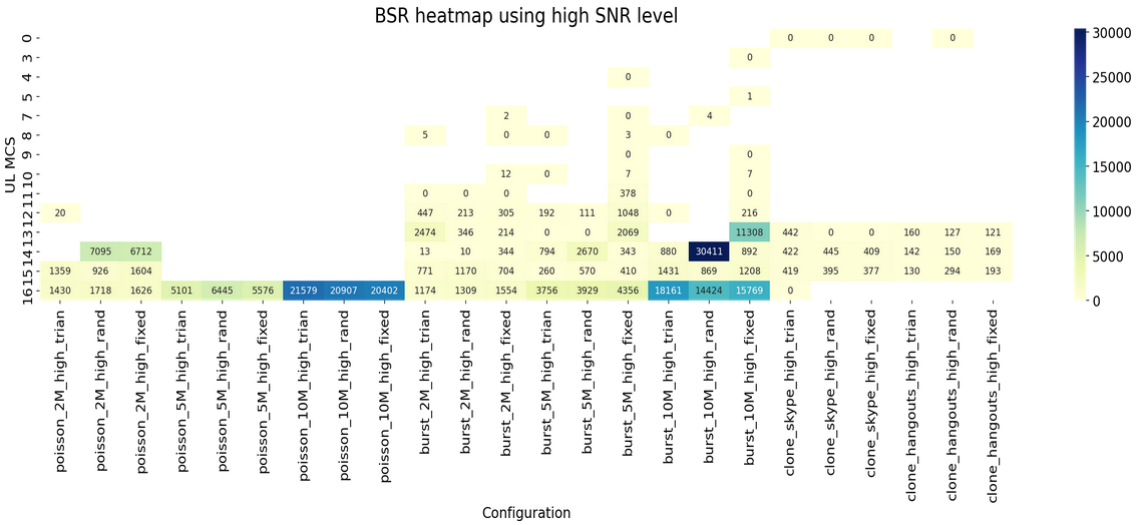


Fig. 3.6. BSR for high level SNR patterns

3.3. Decoding time analysis

Decoding the information from the UE (in the uplink direction) is one of the most time consuming task, as addressed in [14], therefore, the decoding time parameter has a considerable relevance in softwarized RAN scheduling decisions.

In figures 3.7, 3.8, and 3.9, it is observable that the behaviour of the UL decoding time parameter is different depending on the traffic pattern. Additionally, the SNR level seems to alter said behaviour: for high and medium SNR levels the decoding times are very similar, whilst for low levels it is much different. This might imply a certain difficulty in generating this parameter, because the training dataset cannot be grouped neither by traffic pattern nor by SNR level.

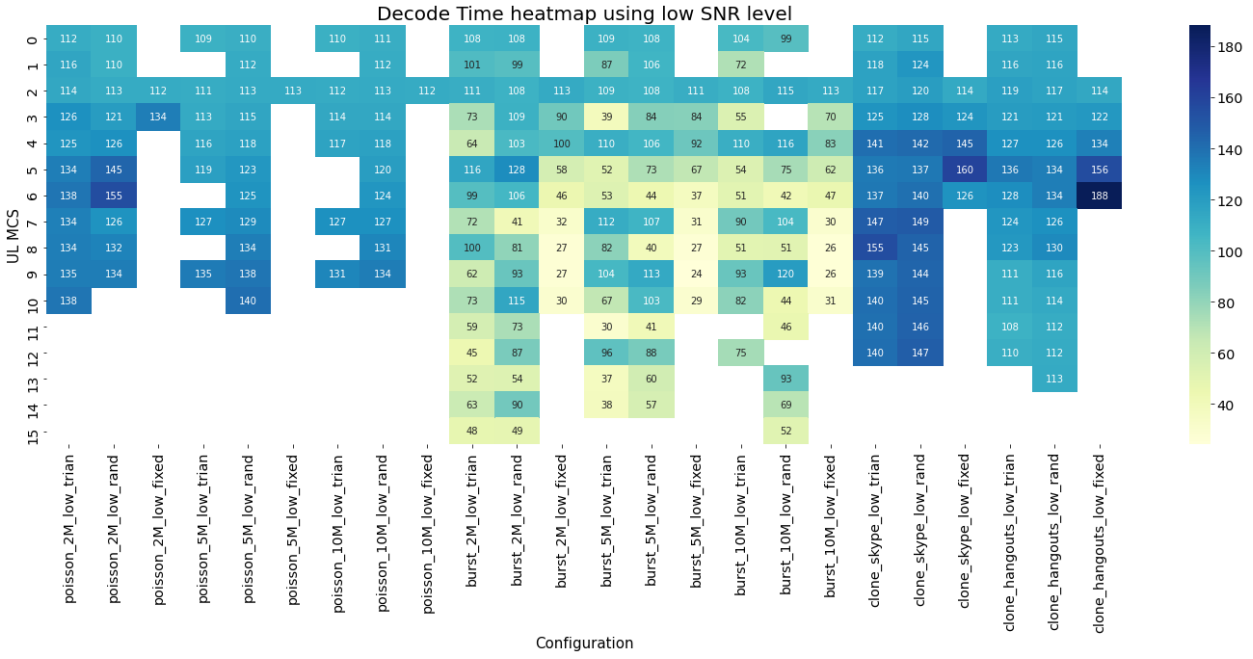


Fig. 3.7. Decoding time for low level SNR patterns

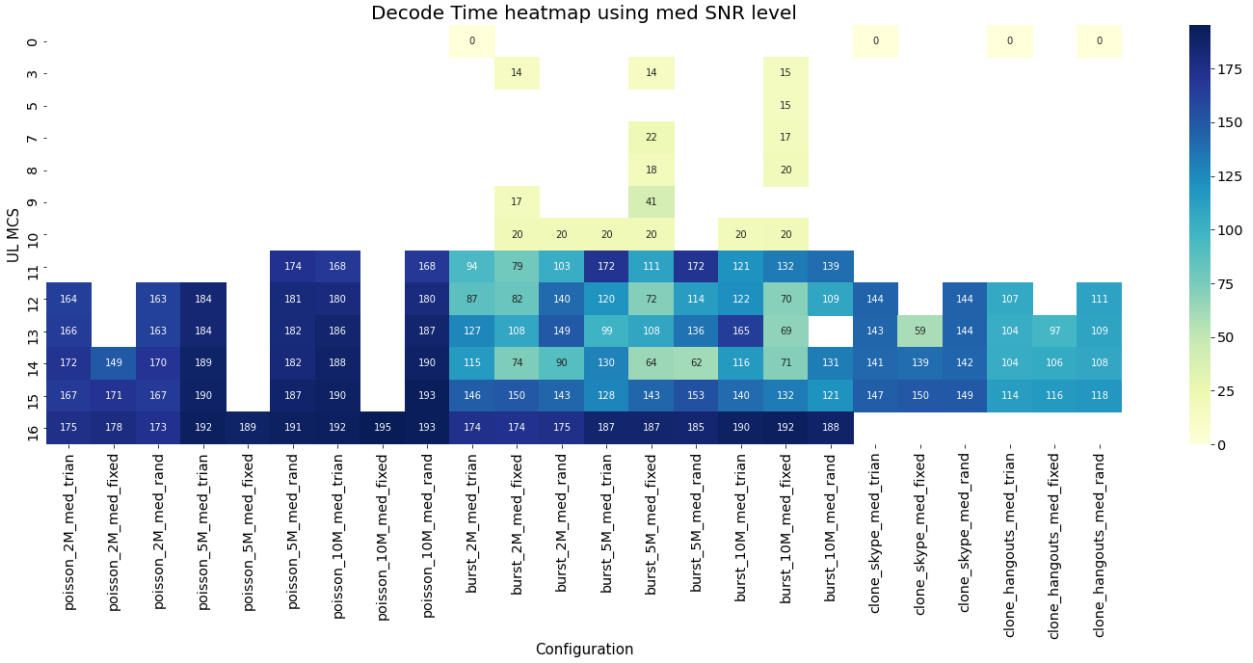


Fig. 3.8. Decoding time for medium level SNR patterns

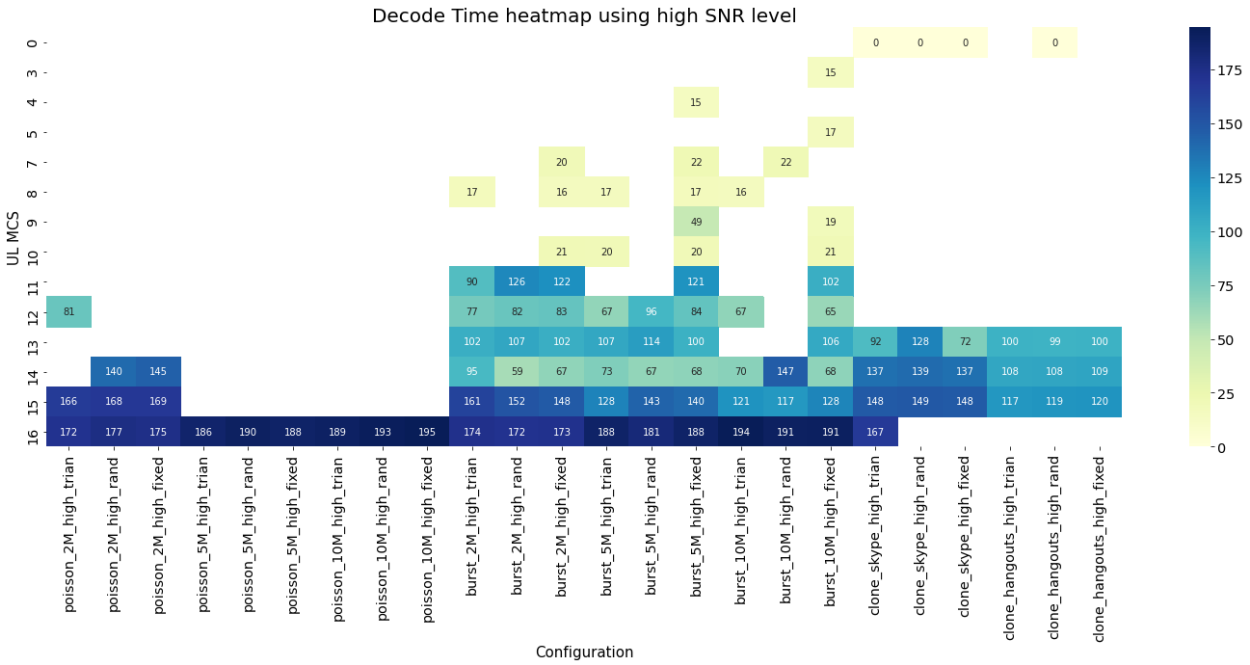


Fig. 3.9. Decoding time for high level SNR patterns

3.4. R^2 correlation

The R^2 correlation coefficient indicates to which degree the variance of a dependant variable can be explained by the variance of an independent variable. In this case the MCS, BSR, and decoding Time are considered the dependant variables and are compared to the SNR. For simplicity, it is calculated as the Pearson correlation squared, given that there is a built in function in the Python library Pandas that computes it for the whole dataset.

The Pearson correlation coefficient indicates how and in which direction are two continuous variables linearly correlated. It is calculated by the equation 3.1:

$$\gamma_{XY} = \frac{(\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$

Equation 3.1. Pearson Correlation Coefficient Formula

From figure 3.10 it is possible to confirm the supposition of section 3.2. The R^2 correlation is high between low SNR levels and MCS. However, for medium and high levels of SNR the correlation with the MCS is much lower.

Only for some cases the MCS is strongly correlated to the SNR, despite knowing that one is adjusted after the other. Consequently it is possible to assume that the neural network can struggle finding a general solution for all cases.

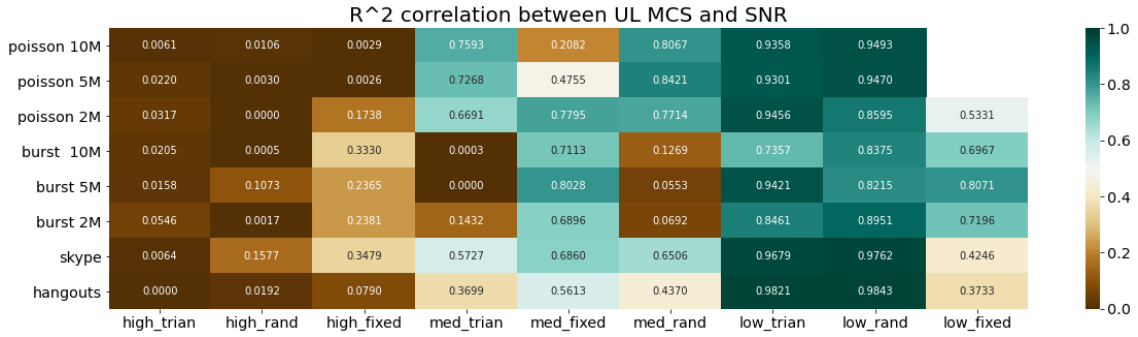


Fig. 3.10. R^2 correlation between SNR and MCS for the different patterns

As it can be observed in Figure 3.11, the R^2 correlation between the BSR and the SNR is generally very close to zero. Therefore, generating this two parameters together may not be beneficial.

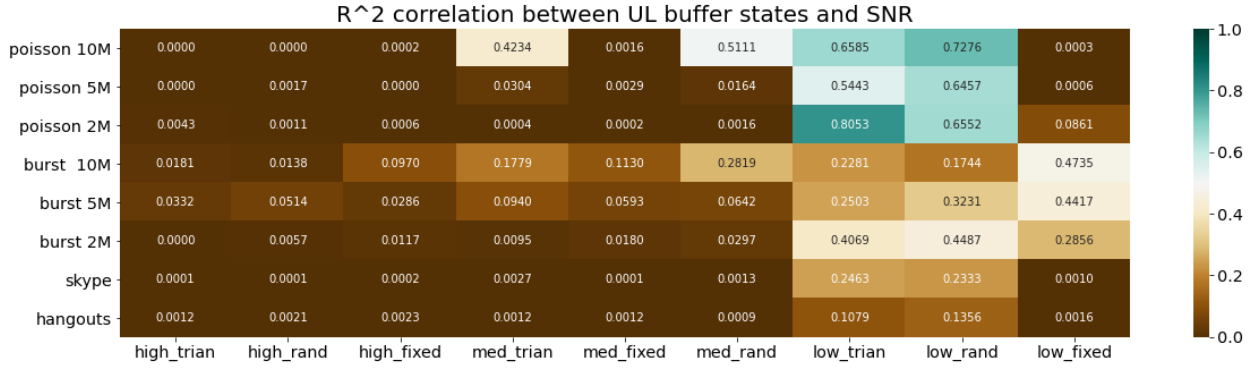


Fig. 3.11. R^2 correlation between SNR and BSR for the different patterns

Hence, the R^2 correlation between the decoding time and the SNR is for some cases very close to zero whilst in others high, as it shows Figure 3.12, generating this two parameters in parallel may delay the training process and possibly keeping it from convergence.

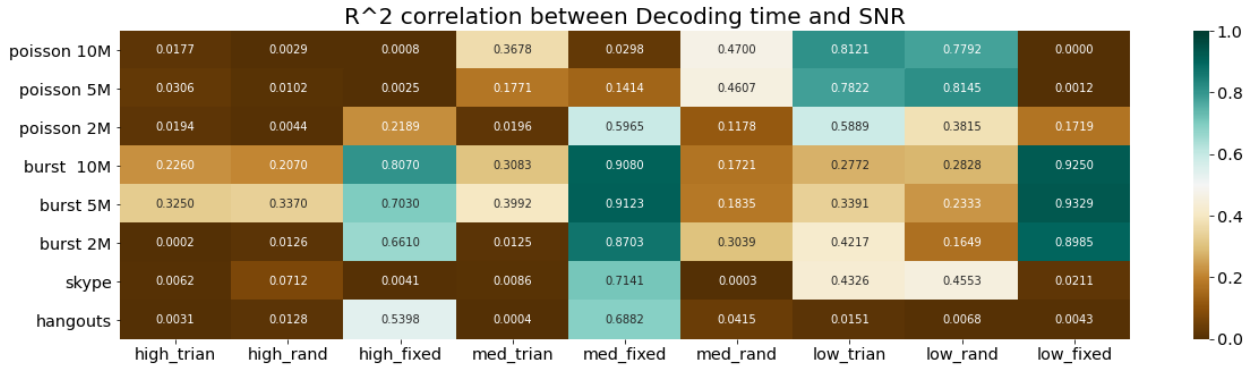


Fig. 3.12. R^2 correlation between SNR and Decoding time for the different patterns

4. DATA GENERATION

Time series forecasting is a research area that is very active. Algorithms like autoregressive integrated moving average integrated moving average (ARIMA), or Markov chain models, have shown their success after some modifications (e.g [21] or [22]). This models can also be adapted and used for time series generation ([23] and [24]). However, in the last years there is a tendency to use artificial neural network models like long short term memory (LSTM) [25] or generative adversarial networks (GAN) [26] for that purpose.

The interest of this study is not only designing a model capable of making mobile network's parameters prediction, but of enlarging the original dataset created following the steps described in section 2. There may be more adequate algorithms for advanced RAN schedulers to forecast parameters, but, in order to train said algorithms, a considerably large dataset is needed.

In this work, a Wasserstein generative adversarial network is used. It is formed by a CNN generator and a CNN critic, due to the fact that these neural networks take into consideration the time dependency between samples and the relationship between the different input parameters. RNNs were also tested, as they also fulfill these requisites, but their performance was poor due to the impossibility of using GPUs to run them in the last PyTorch release.

The distance equation used is the Wasserstein-1 distance, later discussed in section 4.2.1. It is chosen because of the stability benefits that the WGAN algorithm offers over the regular GAN [27].

4.1. Generative adversarial network

Since the presentation of generative adversarial networks (GANs) [28] in 2014 the field of synthetic image generation has seen huge advancements, but the application of GANs are not only limited to that field. In later years, a plethora of works (e.g.[29] or [30]) have studied the time series generation using GANs and variants of it.

A GAN is composed by two different models: a generator and a discriminator.

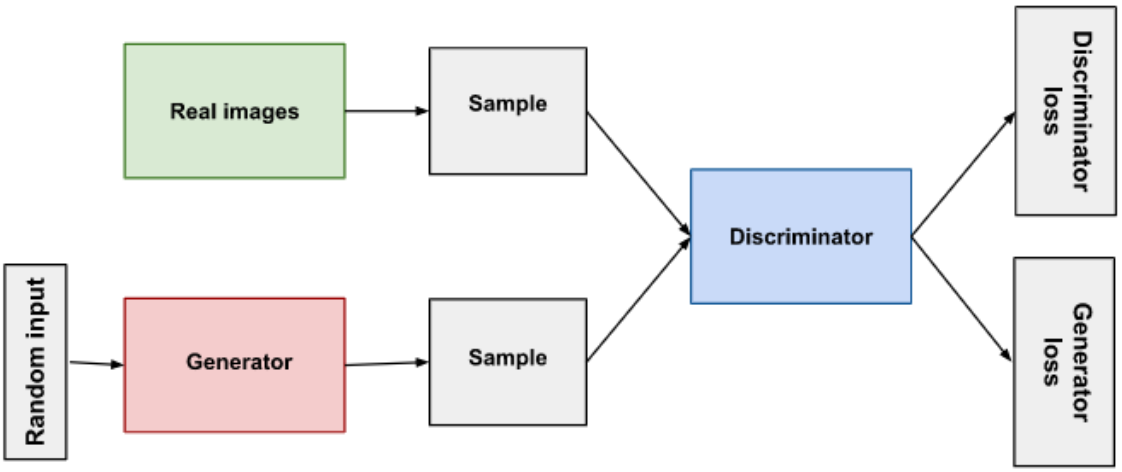


Fig. 4.1. GAN structure diagram [31]

4.1.1. Discriminator

The discriminator model aims to distinguish between the real data and the synthetic data created by the generator. Accordingly, it is a classifier, so any artificial neural network model that classifies can be used for this purpose. The discriminator loss depends on both real and synthetic data and it is back propagated, meaning that the discriminator tries to minimize the error made when missclassifies a sample of data.

As the data generated are time series, there exists a temporal dependency between samples. Consequently, it is necessary to use a discriminator that has that relationship into account, like Causal Convolutional Neural Networks (C-CNN) or Recurrent Neural Networks (RNN). In this work, the models follow the implementation in [32] and are available in appendix 4.

4.1.2. Generator

The generator takes random noise as input and transform it to resemble real data. During its training the generator sends its output to the discriminator and tries to "fool" it into classifying the output as real data. The generator loss is received as feedback from the discriminator, meaning that the generator tries to maximize the error made by the discriminator when it classifies a synthetic sample as a real one.

4.1.3. Limitations

As discussed in [33], training a GAN has some difficulties. The two models are trained alternately, but once the generator is sufficiently good to generate data that the discriminator always classifies as real, the discriminator performs worse -decides arbitrarily-, and therefore the generator loss is random. Consequently, the generator starts creating less quality samples over time. This implies that the convergence of the GAN is compromised, as when it starts converging it gets worse, creating a loop of improving and deteriorating.

Another limitation of GANs is the mode collapse. It happens when the generator learns to create a sample that is similar to real one, and, consequently, it starts to always generate that kind of sample. Then, the discriminator starts to classify all real and synthetic samples that are similar to the one fixed by the generator, as it is the best approach to minimize its loss function. In such a way, the discriminator is avoiding to be fooled by the generator at the expense of rejecting all "realistic" samples of that kind, instead of trying to differentiate between real and fake data.

In [34] other instability sources are addressed as vanishing gradients on the generator or the volatility of generator gradient updates.

4.2. Wasserstein generative adversarial network

The stability of traditional GANs is overcome by Wassertein generative adversarial networks (WGAN) [27], thus this algorithm is chosen over the traditional one for the study.

The WGAN algorithm is well explained in the article [27] and its main difference with the traditional GAN algorithm is that there is no longer a discriminator model but a critic model. This model has to be trained till optimality so that it avoids modes collapse.

For this work the implementation of WGAN is made following the model in [35]. The architecture of the model is the same as the one depicted in 4.1, the only change is that the "discriminator" module is in this case named "critic". The full configuration of the generative model is in appendix 4.

4.2.1. Critic

For traditional GANs, the discriminator tried to minimize the error of missclassifying samples. In WGAN, the critic tries to maximize the distance between the distribution of the real samples and the distribution of the synthetic samples. That means maximizing the distance $W(\mathbb{P}_r, \mathbb{P}_\theta)$ equation 4.1:

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

Equation 4.1 Earth-Mover (EM) distance or Wasserstein-1 after applying Kantorovich-Rubinstein duality [27].

Where:

$\mathbb{E}_{x \sim \mathbb{P}_r}[f(x)]$ is the expectation of the critic's output for a sample of the real distribution input.

$\mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$ is the expectation of the critic's output for a sample of the synthetic distribution.

4.2.2. Generator

In the WGAN model the generator stays the same as in the traditional model. The only change is in its loss function. In the WGAN tries to "move" the distribution of synthetic samples closer to the distribution of the real ones by maximizing $\mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$.

4.2.3. Limitations

Tuning the parameters of the WGAN can be a difficult task. As discussed in [27] , if the weight clipping parameter is not sufficiently high it can lead to vanishing gradients, and if it is large, then the critic takes a long time until reaching optimality. This difficulty is solved in [36] by the introduction of gradient penalty.

4.2.4. Weight clipping modification

In this work a simpler (that may not optimal) solution is implemented. So that the weight clipping parameter does not induce vanishing gradients, this parameter is slightly incremented every ten consecutive critic loss values greater than a boundary, in such a way

that the training is pushed forward, instead of stuck in a sub-optimal solution. So that the clipping parameter still accomplishes its purpose of making Lipschitz constraint be fulfilled, every new epoch the parameter is set to its initial value.

5. RESULTS

Every combination of recurrent neural networks and convolutional neural networks, as generator or critic, is tried. The best results are achieved by the utilization of CNN as generator and critic, and are the ones presented in this section.

This chapter considers a visual comparison between the synthetic data (generated after at least 4 hours of training) and real data. Additionally, it is compared the mean root mean square (RMS) and mean peak to average ratio (PAR) of 100 generated time series with 100 slices of the real time series.

The RMS ratio in the tables 5.2, 5.3, and 5.4, is calculated as the RMS of the real data divided by the RMS of the synthetic data, in such a way that if said ratio is close to 1, the RMS are similar values. The PAR ratio is analogously computed.

5.1. Synthetic trace with multiple parameters

Generating all the parameters simultaneously is ideal, as the created time series would be correlated, and, therefore, be closer to reality. Unfortunately, the relations between said parameters are not simple, and the GAN algorithm is not capable to fully reproduce them. That can be observed by comparing figures 5.1 with 5.2, 5.3 with 5.4, and 5.5 with 5.6. The generated data is nothing alike the real data: the root mean square of the series differ in a ratio of 0.023 in the case of SNR for Poisson pattern, and in a ratio of 2.287 in the case of BSR for Clone pattern, in such a way that not even the difference of RMS is consistent. The PAR does not provide better results a difference greater than 1:4 exists in the SNR for Poisson pattern case. (See Table 5.1)

The simultaneous generation of these parameters is far from acceptable results using the configurations of this work. Better results can be achieved training the model with individual variables.

Metrics of Synthetic and Real Data (Simultaneous Variables)							
		Root Mean Square			Peak to Average Ratio		
		SNR	Decoding time	BSR	SNR	Decoding time	BSR
Poisson	Real data	22.14	168.56	46019.84	1.057	1.089	3.735
	Synthetic data	960.43	6047.53	163161.50	4.743	4.6213	6.784
Burst	Real data	22.39	137.86	42321.42	1.447	1.361	4.076
	Synthetic data	37.47	47.92	7288.86	1.084	3.754	6.447
Clone	Real data	21.92	126.97	1413.68	1.242	1.234	6.215
	Synthetic data	20.21	78.97	618.24	3.331	3.602	7.020

Table 5.1. METRICS OF SYNTHETIC AND REAL DATA WHEN TRAINED WITH SEVERAL VARIABLES SIMULTANEOUSLY

5.1.1. Poisson pattern

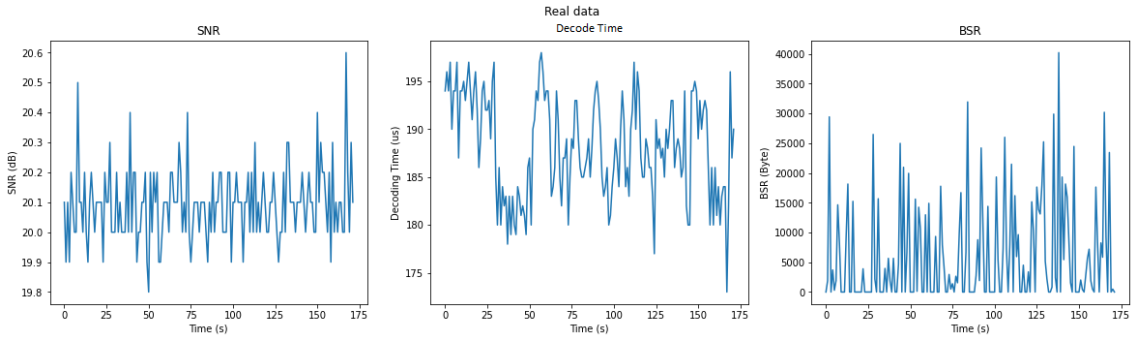


Fig. 5.1. Example of SNR, decoding time, and BSR values from real dataset using Poisson patterns

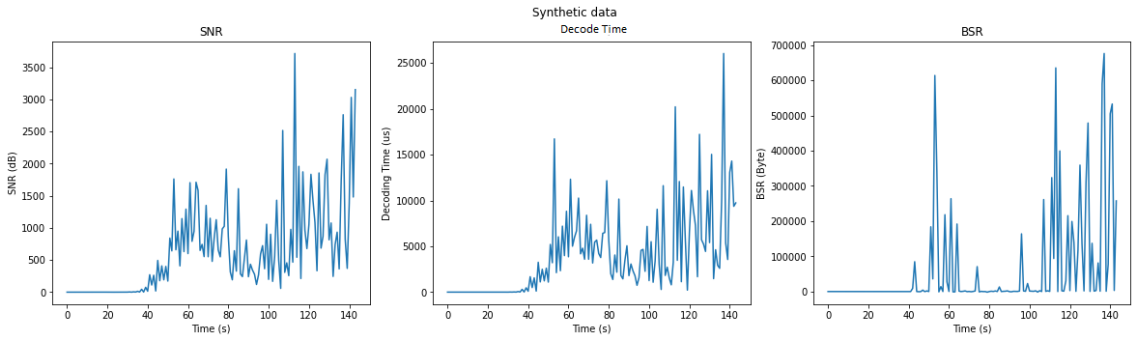


Fig. 5.2. Synthetic generation of SNR, decoding time, and BSR values trained with the Poisson patterns

5.1.2. Burst pattern

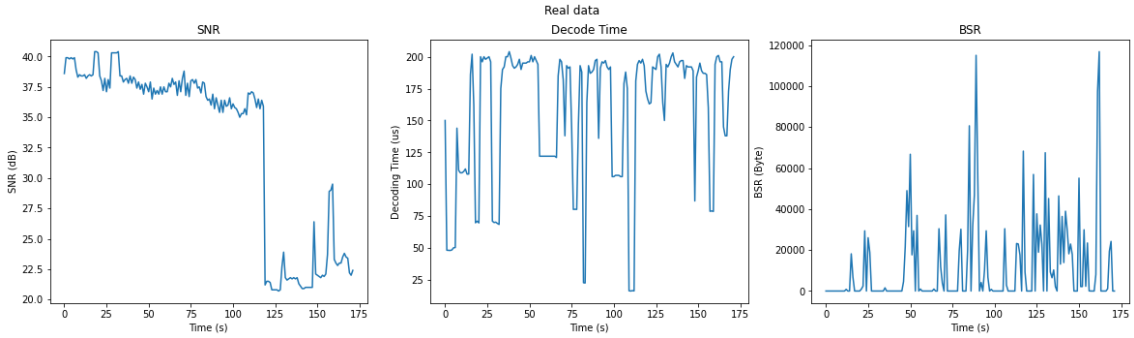


Fig. 5.3. Example of SNR, decoding time, and BSR values from real dataset using Burst patterns

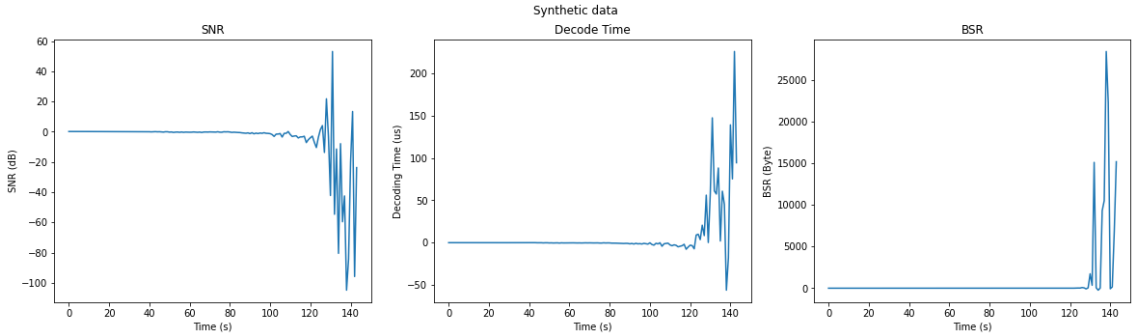


Fig. 5.4. Synthetic generation of SNR, decoding time, and BSR values trained with the Burst patterns

5.1.3. Clone pattern

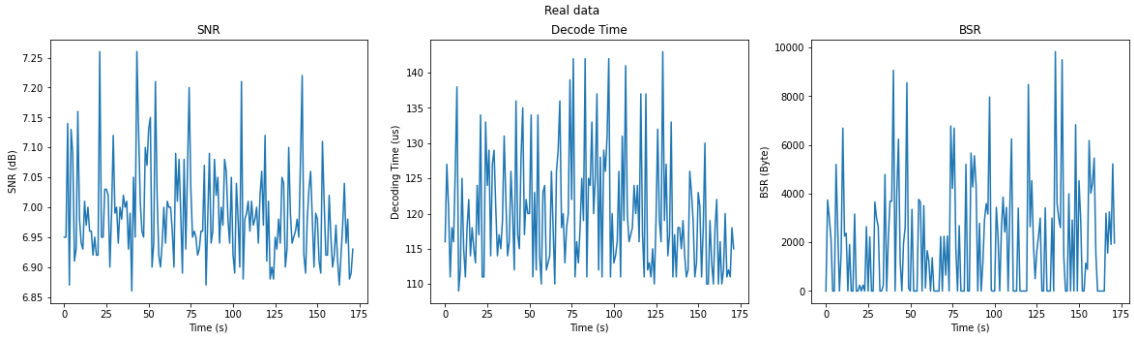


Fig. 5.5. Example of SNR, decoding time, and BSR values from real dataset using Clone patterns

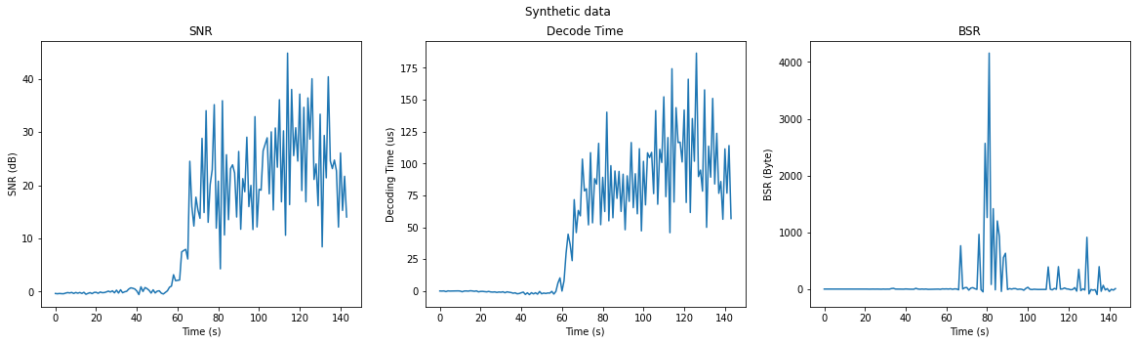


Fig. 5.6. Synthetic generation of SNR, decoding time, and BSR values trained with the Clone patterns

5.2. Synthetic SNR

The synthetic SNR values are generated with the WGAN trained on fixed SNR level patterns. This is due to the impossibility of the WGAN to adapt to the sudden changes of the stepped triangular waveform and the random waveform. In order to create more realistic scenarios of SNR for this purpose, smoother SNR patterns should be applied.

The RMS values in Table 5.2 are promising: their ratios of difference are very close to 1. On the other hand, the peak to average ratios are less favorable, as their differences are higher. This can be due to the fact that the variance of SNR for the fixed pattern is very limited, and since the synthetic data is generated from random noise, the model struggles to "flatten" said input. A possible improvement, to get more realistic results, might be increasing the values of the weight clipping parameter, but then much longer trainings would be required.

Metrics of Synthetic and Real Data (SNR)					
		RMS	RMS ratio	PAR	PAR ratio
Low SNR	Real data Synthetic data	6.83 6.81	1.003	1.010 1.177	0.904
Medium SNR	Real data Synthetic data	20.15 20.33	0.991	1.026 1.075	0.954
High SNR	Real data Synthetic data	33.69 33.85	0.995	1.033 1.178	0.877

Table 5.2. METRICS OF SYNTHETIC AND REAL DATA WHEN TRAINED WITH SNR VALUES

5.2.1. Low fixed SNR

As it can be observed in figure 5.7 the real SNR takes values between 6.800 dB and 6.840 dB, whereas in figure 5.8 the synthetic SNR takes values from approximately 5.1 dB to 8.5 dB. Thus, the PAR is different between real and synthetic series, even if the RMS is almost the same.

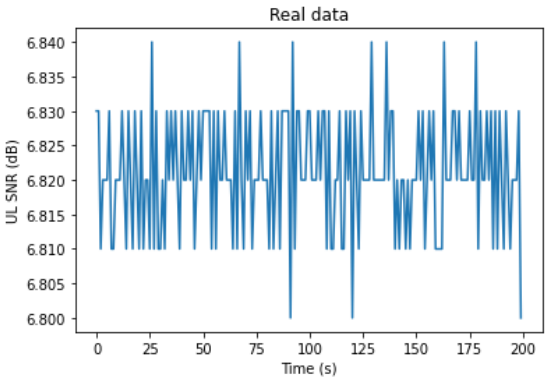


Fig. 5.7. Example of SNR values from real dataset using low fixed SNR levels patterns

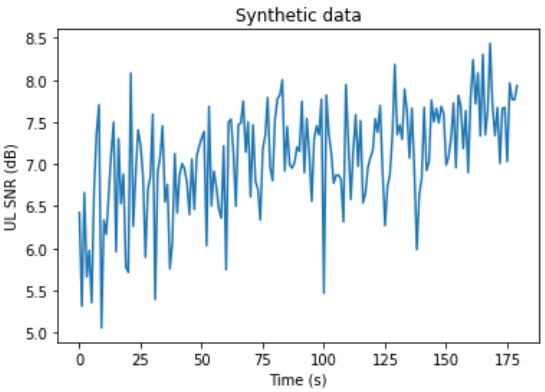


Fig. 5.8. Synthetic generation of SNR values trained with the low fixed SNR levels patterns

5.2.2. Medium fixed SNR

In the figures 5.9 and 5.10 it is observable that the behaviour for medium SNR levels is similar to the low SNR levels: the average values of the time series are almost the same, but the synthetic time series varies in a wider range than the real one. Additionally, if the moving average of both time series was calculated, it would be very stable for the real data, whereas it would oscillate in the synthetic data.

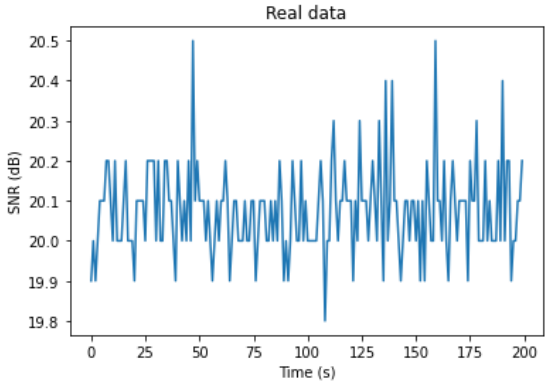


Fig. 5.9. Example of SNR values from real dataset using medium fixed SNR levels patterns

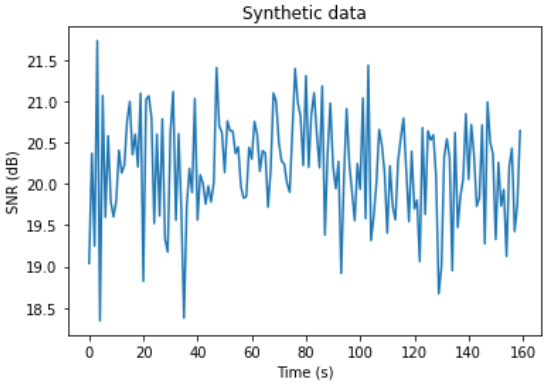


Fig. 5.10. Synthetic generation of SNR values trained with the medium fixed SNR levels patterns

5.2.3. High fixed SNR

It is observable, comparing figures 5.11 and 5.12, that the average value of both time series are similar, but the synthetic data presents a much higher variance: it takes values from 24.0 dB to almost 40 dB, whilst the real is between 33.0 dB and 34.2 dB. This behaviour explains why the worse PAR ratio calculated for SNR generation is for the high SNR level. It could be solved by a longer training with higher value of the weight clipping parameter.

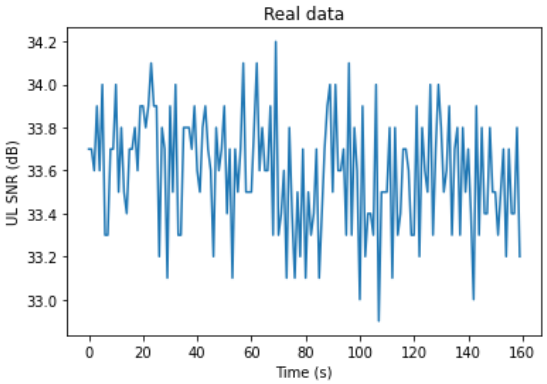


Fig. 5.11. Example of SNR values from real dataset using high fixed SNR levels patterns

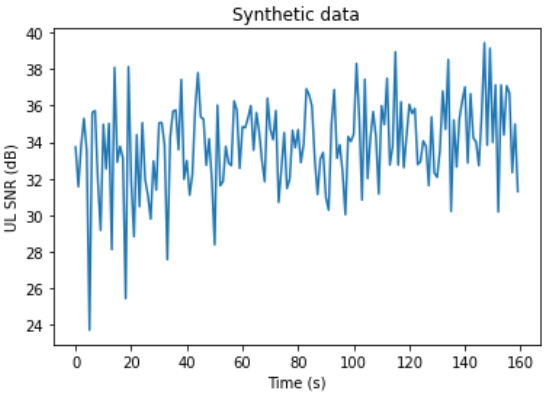


Fig. 5.12. Synthetic generation of SNR values trained with the high fixed SNR levels patterns

5.3. Synthetic BSR

The real values BSR are characterized by their sudden changes from low to high values and the synthetically generated values, as observed in 5.14, 5.16, and 5.18, accomplish to resemble that attribute. However, the synthetic values are not always in the same ranges than their real equivalents, therefore, even when the PAR are very similar, the RMS are very different.

Metrics of Synthetic and Real Data (BSR)					
		RMS	RMS ratio	PAR	PAR ratio
Poisson	Real data	46640.56	2.258	3.591	0.898
	Synthetic data	20657.88		3.998	
Burst	Real data	34387.23	2.519	4.468	0.699
	Synthetic data	13649.65		6.390	
Clone	Real data	1442.11	0.739	6.200	0.991
	Synthetic data	1950.71		6.259	

Table 5.3. METRICS OF SYNTHETIC AND REAL DATA WHEN TRAINED WITH BSR VALUES

5.3.1. Poisson pattern

It is discernible in figures 5.13 and 5.14 that the values of the real BSR are around 150 KBytes, whereas in the case of the synthetic the values do not overcome the 70 KBytes. This is reflected in the table 5.3, where for the Poisson pattern there is a ratio between real and synthetic RMS of 2.258, meaning that the real is more than the double of the synthetic. Despite that, behaviour of having sudden changes is rather good emulated, as the PAR are similar.

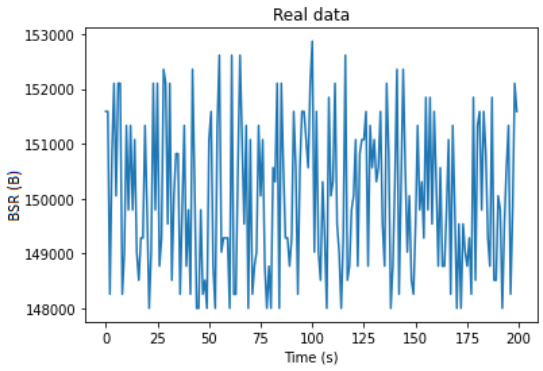


Fig. 5.13. Example of BSR values from real dataset using Poisson patterns

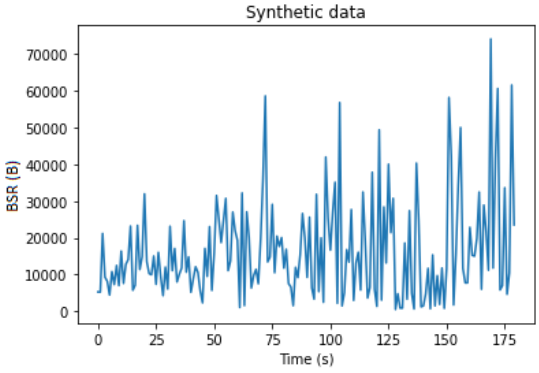


Fig. 5.14. Synthetic generation of BSR values trained with the Poisson patterns

5.3.2. Burst pattern

Even given the not promising RMS and PAR metrics for this scenario, by comparing figures 5.15 and 5.16, it is possible to say that the synthetic time series is similar to the real ones. The dissonance between the metrics and the graphs may be due to the intervals in the synthetic series where the values are very close to zero. For example, in figure 5.16 there are two periods from approximately 85 s to 105 s, and from 115 s to 130 s, where the values are very low, whilst this rarely happen in real data.

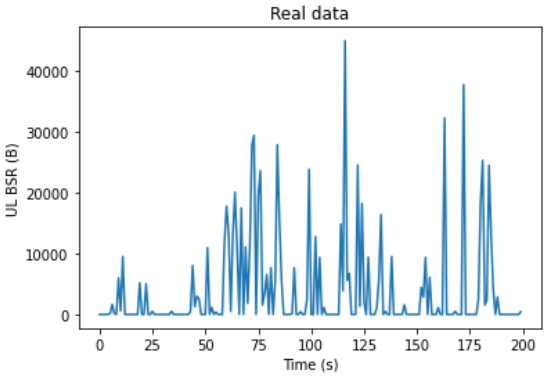


Fig. 5.15. Example of BSR values from real dataset using Burst patterns

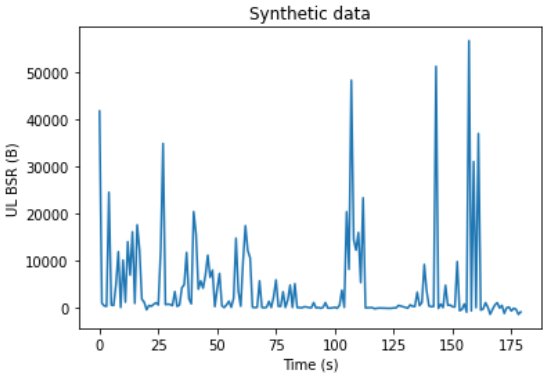


Fig. 5.16. Synthetic generation of BSR values trained with the Burst patterns

5.3.3. Clone pattern

In this scenario, the real and synthetic time series are very similar as it can be observed in figures 5.17 and 5.18. Their PARs are also close, as the ratio between these values is almost equal to 1. However, the RMS values are not that similar. This could be because there are some times in the synthetic data that are "noisy", as in the first 50 s plotted in 5.18, that is mostly never close to zero.

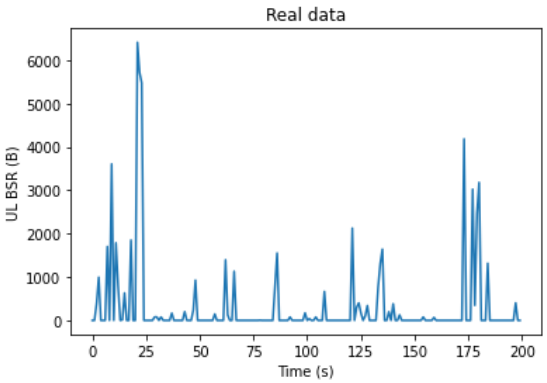


Fig. 5.17. Example of BSR values from real dataset using Clone patterns

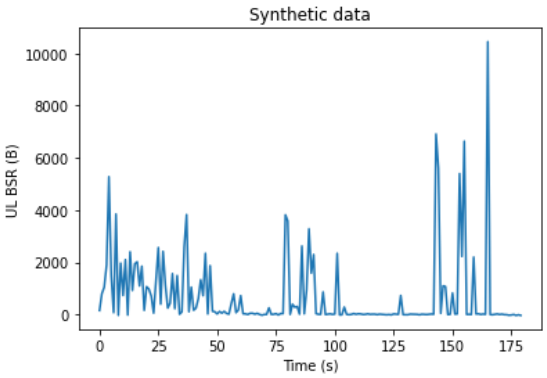


Fig. 5.18. Synthetic generation of BSR values trained with the Clone patterns

5.4. Synthetic Decoding Time

The generated decoding time data is rather similar to the original data, as it can be checked by the metrics in Table 5.4. However, by observing the figures 5.19, 5.20, 5.21,

and 5.22, it is possible to differentiate between real and synthetic data, by the shape that the time series take.

Metrics of Synthetic and Real Data (Decoding Time)					
		RMS	RMS ratio	PAR	PAR ratio
Poisson	Real data	163.26	0.942	1.134	0.942
	Synthetic data	173.35		1.204	
Burst	Real data	140.95	1.041	1.379	0.810
	Synthetic data	135.43		1.690	
Clone	Real data	121.36	1.023	1.226	0.956
	Synthetic data	118.57		1.283	

Table 5.4. METRICS OF SYNTHETIC AND REAL DATA WHEN TRAINED WITH DECODING TIME VALUES

5.4.1. Poisson pattern

In the Poisson pattern case, it is possible, by comparing figures 5.19 and 5.20, to deduct that although both time series have very close average values and variance, their shapes are different. The real data seems to oscillate around a given value with certain peak, whereas the synthetic data seems to have an incrementing average.

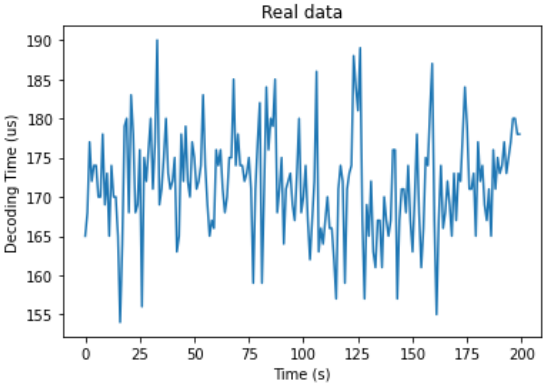


Fig. 5.19. Example of decoding time values from real dataset using Poisson patterns

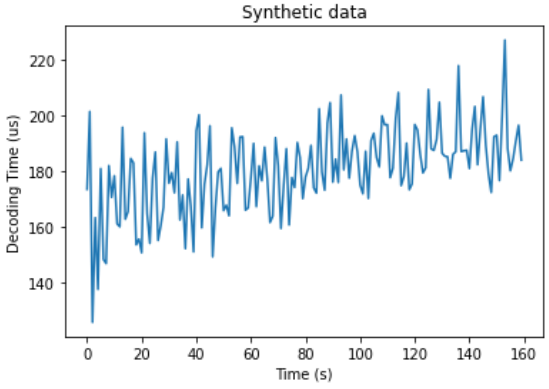


Fig. 5.20. Synthetic generation of decoding time values trained with the Poisson patterns

5.4.2. Burst pattern

It can be seen in the figures 5.21 and 5.22 that both time series are very similar but for one aspect: it seems that the synthetic data is incapable of reproducing stable periods of the real data. It looks as if the synthetic time series was a version of the real one plus an additive noise. This problem may be solved by reducing the kernel size of the CNNs, but then the other characteristics, as the sudden steps, may be compromised.

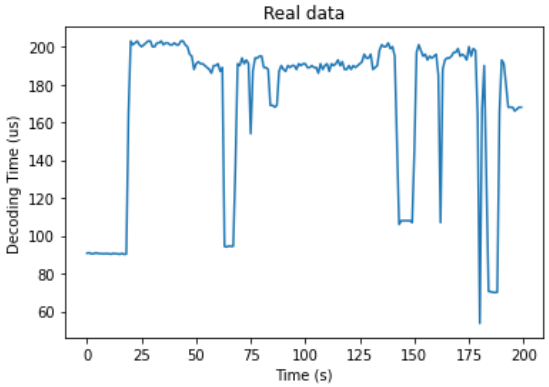


Fig. 5.21. Example of decoding time values from real dataset using Burst patterns

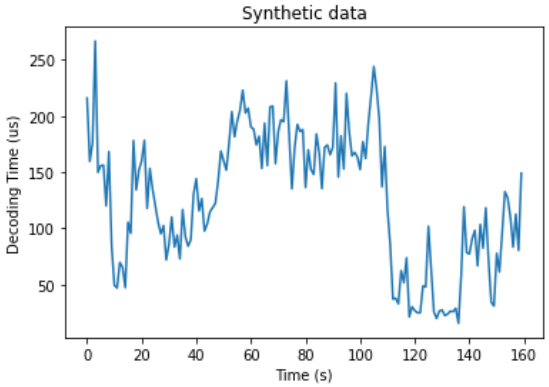


Fig. 5.22. Synthetic generation of decoding time values trained with the Burst patterns

5.4.3. Clone pattern

The figures 5.23 and 5.24 are considerably similar: their average values are in the same range, and they vary in the same manner: having sudden positive and negative peaks. This could be due to the real data of this scenario being stable, that the model is capable of generating a good resemblance. This would then reinforce the theory that the model's capacity of emulating time series of low variance is limited.

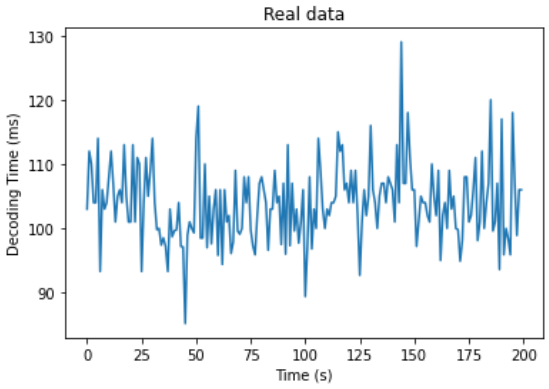


Fig. 5.23. Example of decoding time values from real dataset using Clone patterns

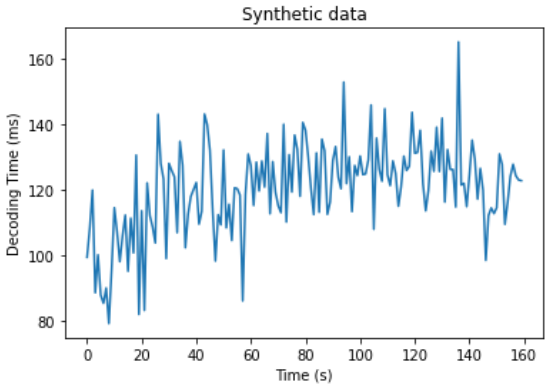


Fig. 5.24. Synthetic generation of decoding time values trained with the Clone patterns

6. CONCLUSIONS

This work purposes a WGAN model to generate different mobile network parameters as SNR, BSR, or decoding time. Although promising results are obtained for said parameters, some limitations are addressed as the model struggles with emulating all metrics simultaneously and low variance time series.

It is possible to say that the combination of CNNs as generator and critic performs better than any combination with RNNs, as they lead to a less efficient (and therefore slower) trainings.

Generally, parameters generated using datasets of Clone patterns outperform the other cases in being similar to real data. This is a positive aspect given that Clone patterns are the nearest to real use cases: they replicate actual traffic scenarios.

The SNR patterns have not shown to be very useful as the sudden changes of the SNR values cannot be simulated with the purposed generative model. However, by using them, more diverse scenarios have been created for the other parameters. Smoother evolving SNR patterns might be more appropriate, but the repercussion of high frequency requests of transmission gain modification should then be studied in detail.

It would be helpful to have a higher time resolution original dataset, in order to analyse profoundly the time evolution of the metrics. Other further steps may include the utilization of peak to average ratio metric in the training process, the implementation of a WGAN with gradient penalty, or the utilization of automated hyperparameters tuning algorithm.

BIBLIOGRAPHY

[1] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges", *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[2] J. Wu, Z. Zhang, Y. Hong, and Y. Wen, "Cloud radio access network (c-ran): A primer", *IEEE Network*, vol. 29, no. 1, pp. 35–41, 2015.

[3] M. Chui *et al.*, "Notes from the ai frontier: Insights from hundreds of use cases", *Notes from the AI frontier: Applications and value of deep learning*, Apr. 2018. [Online]. Available: https://www.mckinsey.com/~/media/mckinsey/featured%20insights/artificial%20intelligence/notes%20from%20the%20ai%20frontier%20applications%20and%20value%20of%20deep%20learning/notes-from-the-ai-frontier-insights-from-hundreds-of-use-cases-discussion-paper.pdf.

[4] J. Bughin, J. Seong, J. Manyika, M. Chui, and R. Joshi, *Notes from the AI frontier: Modeling the impact of AI on the world economy*, Sep. 2018. [Online]. Available: https://www.mckinsey.com/~/media/McKinsey/Featured%20Insights/Artificial%20Intelligence/Notes%20from%20the%20frontier%20Modeling%20the%20impact%20of%20AI%20on%20the%20world%20economy/MGI-Notes-from-the-AI-frontier-Modeling-the-impact-of-AI-on-the-world-economy-September-2018.pdf.

[5] E. Brynjolfsson and T. Mitchell, "What can machine learning do? workforce implications", *Science*, vol. 358, no. 6370, pp. 1530–1534, 2017.

[6] [Online]. Available: https://www.nokia.com/networks/solutions/nokia-ava-telco-ai-ecosystem/.

[7] *Employing AI techniques to enhance returns on 5G network investments*, 2019. [Online]. Available: https://www.ericsson.com/49b63f/assets/

local/networks/offerings/machine-learning-and-ai-aw-screen.pdf.

[8] *The open source definition (annotated)*, Mar. 2007. [Online]. Available: https://opensource.org/docs/definition.php.

[9] [Online]. Available: http://www.gnu.org/philosophy/free-sw.html.

[10] J. Ayala Romero *et al.*, "Vrain a deep learning approach tailoring computing and radio resources in virtualized rans", May 2019, pp. 1–16. DOI: 10.11453300061.3345431.

[11] J. Đurković, V. Vuković, and L. Raković, "Open source approach in software development—advantages and disadvantages", *Manag Inforation Syst*, vol. 3, pp. 029–33, 2008.

[12] Apr. 2013. [Online]. Available: https://www.skype.com/en/legal/ios/tos/.

[13] S. Bhaumik *et al.*, "Cloudiq: A framework for processing base stations in a data center", in *Proceedings of the 18th annual international conference on Mobile computing and networking*, 2012, pp. 125–136.

[14] K. C. Garikipati, K. Fawaz, and K. G. Shin, "Rt-opex: Flexible scheduling for cloud-ran processing", in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 267–280.

[15] S. A. Ahson and M. Ilyas, *WiMAX: technologies, performance analysis, and QoS*. CRC press, 2018, pp. 227–232.

[16] A. M. Mikaeil, W. Hu, and L. Li, "Joint allocation of radio and fronthaul resources in multi-wavelength-enabled c-ran based on reinforcement learning", 2019.

[17] I. Gomez-Miguelez *et al.*, "Srslte: An open-source platform for lte evolution and experimentation", in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, ser. WiNTECH '16, New York City, New York: Association for Computing Machinery, 2016, pp. 25–32. DOI: 10.1145/2980159.2980163. [Online]. Available: https://doi.org/10.1145/2980159.2980163.

[18] T. Schmid, O. Sekkat, and M. B. Srivastava, "An experimental study of network performance impact of increased latency in software defined radios", in *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, ser. WinTECH '07, Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 59–66. DOI: 10.1145/1287767.1287779. [Online]. Available: https://doi.org/10.1145/1287767.1287779.

[19] 2019. [Online]. Available: https://docs.srslte.com/en/latest/.

[20] J. Weston, *Mgen documentation*, Sep. 2019. [Online]. Available: https://github.com/USNavalResearchLaboratory/mgen/blob/master/doc/mgen.pdf.

[21] G. P. Zhang, "Time series forecasting using a hybrid arima and neural network model", *Neurocomputing*, vol. 50, pp. 159–175, 2003.

[22] Y. Zhang, "Prediction of financial time series with hidden markov models", PhD thesis, Applied Sciences: School of Computing Science, 2004.

[23] P. Chen, T. Pedersen, B. Bak-Jensen, and Z. Chen, "Arima-based time series model of stochastic wind power generation", *IEEE transactions on power systems*, vol. 25, no. 2, pp. 667–676, 2009.

[24] A. Shamshad, M. Bawadi, W. W. Hussin, T. Majid, and S. Sanusi, "First and second order markov chain models for synthetic generation of wind speed time series", *Energy*, vol. 30, no. 5, pp. 693–708, 2005.

[25] Y. Hua *et al.*, "Deep learning with long short-term memory for time series prediction", *IEEE Communications Magazine*, vol. 57, no. 6, pp. 114–119, 2019.

[26] C. Zhang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Generative adversarial network for synthetic time series data generation in smart grids", in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, IEEE, 2018, pp. 1–6.

[27] M. Arjovsky, S. Chintala, and L. Bottou, *Wasserstein gan*, 2017. arXiv: 1701.07875 [stat.ML].

[28] I. Goodfellow *et al.*, "Generative adversarial nets", in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[29] C. Esteban, S. L. Hyland, and G. Ratsch, "Real-valued (medical) time series generation with recurrent conditional gans", *arXiv preprint arXiv1706.02633*, 2017.

[30] Y. Luo, X. Cai, Y. Zhang, J. Xu, *et al.*, "Multivariate time series imputation with generative adversarial networks", in *Advances in Neural Information Processing Systems*, 2018, pp. 1596–1607.

[31] *GAN Structure Diagram*. Google Developers, May 2019. [Online]. Available: https://developers.google.com/machine-learning/gan/gan_structure.

[32] P. D'Oro, *Proceduralia/pytorch-gan-timeseries*, Sep. 2019. [Online]. Available: https://github.com/proceduralia/pytorch-GAN-timeseries.

[33] *GAN Training*. Google Developers, May 2019. [Online]. Available: https://developers.google.com/machine-learning/gan/training.

[34] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks", *arXiv preprint arXiv:1701.04862*, 2017.

[35] A. Kristiadi, *Wiseodd/generative-models/gan/wasserstein_gan*, Mar. 2017. [Online]. Available: https://github.com/wiseodd/generative-models/blob/master/GAN/wasserstein_gan/wgan_pytorch.py.

[36] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans", in *Advances in neural information processing systems*, 2017, pp. 5767–5777.

APPENDIX 1

CÓDIGO 6.1. set_txgain.py

```
import zmq
import sys, getopt
import time
import random

# Global variables
prevGain = 0
counter = 0
topGain = 84
bottomGain = 24
waveSlope = 1

# Gain update functions
def squareWaveUpdateGain():
    global counter
    if counter<10000000:
        counter += 1
        return 80
    else :
        if counter > 20000000:
            counter = 0
        counter += 1
        return 40

def triangularWaveUpdateGain(prevGain):
   global topGain
   global bottomGain
   global waveSlope

   if prevGain == 0:
        prevGain = (topGain + bottomGain) / 2
   else:
        time.sleep(10)
```

```
34
35          if prevGain >= topGain:
36              waveSlope = -1
37          if prevGain <= bottomGain:
38              waveSlope = 1
39
40          prevGain += waveSlope
41
42      return prevGain
43
44  def randomWaveUpdateGain(prevGain):
45      global topGain
46      global bottomGain
47      global waveSlope
48
49      if prevGain == 0:
50          prevGain = random.randint(bottomGain, topGain)
51      else:
52          time.sleep(10)
53                  waveSlope = random.randint(-5,5)
54                  prevGain += waveSlope
55          if prevGain > topGain:
56              prevGain = topGain
57          if prevGain < bottomGain:
58              prevGain = bottomGain
59
60      return prevGain
61
62
63  # Sets the tx gain limits depending on the mode (low = 'l', medium =
        'm', or high = 'h')
64  def setGainLimits(mode):
65      modes = ['l', 'm', 'h']
66      if mode not in modes:
67                  print ("Unvalid mode selected")
68                  return
69      if mode == 'l':
70                  topGain = 38
71                  bottomGain = 24
```

```
72      if mode == 'm':
73                  topGain = 50
74          bottomGain = 39
75      if mode == 'h':
76          topGain = 84
77          bottomGain = 51
78      return topGain, bottomGain
79
80  # Main
81  def main(argv):
82      global prevGain
83      if len(argv) < 2:
84          print("Not arguments enough. Need tx_gain mode (h, m, or l)
                and wave mode (trian or rand) ")
85          return
86      global topGain
87      global bottomGain
88      topGain, bottomGain = setGainLimits(argv[0])
89
90      if topGain == 84 and bottomGain == 24:
91          print("Error setting tx_gain mode")
92          return 0
93      if argv[1] == 'trian':
94          gainUpdated = triangularWaveUpdateGain(prevGain)
95      elif argv[1] == 'rand':
96          gainUpdated = randomWaveUpdateGain(prevGain)
97
98      if (prevGain <> gainUpdated):
99          try:
100             opts, args = getopt.getopt(argv,"hg:o:")
101         except getopt.GetoptError:
102             print('set_txgain.py -g <gain> ')
103             sys.exit(2)
104         for opt, arg in opts:
105             if opt == '-h':
106                 print('set_txgain.py -p <gain>')
107                 sys.exit()
108             elif opt in ("-g", "--gain"):
109                 gainUpdated = int(arg)
```

```
110             print('Requesting tx gain = ', gainUpdated)
111
112             context = zmq.Context()
113
114             #  Socket to talk to server
115             socket = context.socket(zmq.REQ)
116             socket.connect("tcp://localhost:5557")
117
118             # CONVERTING GAIN (BASE IS 74)
119             gain = gainUpdated-74
120
121             print("Sending request ...")
122             socket.send_string(str(gain))
123
124             #  Get the reply.
125             message = socket.recv()
126             print("Received reply [ %s ]" % (message))
127         #else :
128             #print("No update")
129         prevGain = gainUpdated
130         return 1
131
132 if __name__ == "__main__":
133     running = 1
134     while running==1:
135         running = main(sys.argv[1:])
```

APPENDIX 2

CÓDIGO 6.2. fixed_txgain.py

```
import zmq
import sys, getopt

def main(argv):
    if argv[0] == 'h':
        gain = 63
    elif argv[0] == 'm':
        gain = 45
    elif argv[0] == 'l':
        gain = 31
    else:
        gain = 85
    try:
        opts, args = getopt.getopt(argv,"hg:o:")
    except getopt.GetoptError:
        print('set_txgain.py -g <gain> ')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print('set_txgain.py -p <gain>')
            sys.exit()
        elif opt in ("-g", "--gain"):
            gain = int(arg)
    print('Requesting tx gain = ', gain)

    context = zmq.Context()

    #  Socket to talk to server
    socket = context.socket(zmq.REQ)
    socket.connect("tcp://localhost:5557")


    # CONVERTING GAIN (BASE IS 74)
```

```
34      gain = gain-74
35
36      print("Sending request ...")
37      socket.send_string(str(gain))
38
39      #  Get the reply.
40      message = socket.recv()
41      print("Received reply [ %s ]" % (message))
42
43  if __name__ == "__main__":
44     main(sys.argv[1:])
```

APPENDIX 3

CÓDIGO 6.3. monitoring.py

```
import zmq
import time

context = zmq.Context()

#  Socket to talk to server
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5556")

while 1:
        action=[1,2]
        socket.send_string("REQ")
        message = socket.recv()
        print(message)
        time.sleep(1)
```

APPENDIX 4

CÓDIGO 6.4. pytorch_wgan.py

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from matplotlib import pyplot as plt
import tensorflow as tf
import numpy as np

import torch
import torch.nn
import torch.nn.utils
import torch.nn.functional as nn
import torch.autograd as autograd
import torch.optim as optim
from torch.autograd import Variable
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset

"""# Data import"""

labels =["sample","period","timestamp","UEs count","subsamples count
    ","Error count 1","Error count 2","Error count 3","DL MCS",
          "UL MCS", "DL rate", "UL rate","UL goodput","DL buffer
              states","UL buffer states","UL mean SINR", "UL var
              SINR UEs", "UL mean RSSI",
          "UL var RSSI UEs","Turbodecoder iterations", "Decoding
              time", "DL BLER", "UL BLER", ""]

common_path = '/content/drive/My Drive/TFG/mgen/data/mgen_data/
    filtered_data/'
patterns = ['poisson', 'burst', 'clone']
ul_rates = ['2M', '5M', '10M']
video_apps = ['skype', 'hangouts']
```

```
gain_modes =['high_trian', 'med_trian', 'low_trian', 'high_rand', '
    med_rand', 'low_rand', 'high_fixed', 'med_fixed', 'low_fixed']

from google.colab import drive
drive.mount('/content/drive')

import pandas as pd
src_paths = []

src_paths_poisson = []
src_paths_burst = []
src_paths_clone = []

[src_paths_poisson.append(common_path+patterns[0]+'_'+ul_rate+'_'+
    gain_mode+'.csv')
     for ul_rate in ul_rates
     for gain_mode in gain_modes]

[src_paths_burst.append(common_path+patterns[1]+'_'+ul_rate+'_'+
    gain_mode+'.csv')
     for ul_rate in ul_rates
     for gain_mode in gain_modes]

[src_paths_clone.append(common_path+patterns[2]+'_'+video_app+'_'+
    gain_mode+'.csv')
     for video_app in video_apps
     for gain_mode in gain_modes]

src_paths.append(src_paths_poisson)
src_paths.append(src_paths_burst)
src_paths.append(src_paths_clone)

data_frames = [[],[],[]]

poisson_df = []
for i in range(len(data_frames)):
  [poisson_df.append(pd.read_csv(path, sep=",", encoding="utf-8",
      header = 0)) for path in src_paths_poisson]

```

```
burst_df = []
for i in range(len(data_frames)):
  [burst_df.append(pd.read_csv(path, sep=",", encoding="utf-8",
     header = 0)) for path in src_paths_burst]

clone_df = []
for i in range(len(data_frames)):
  [clone_df.append(pd.read_csv(path, sep=",", encoding="utf-8",
     header = 0)) for path in src_paths_clone]

huge_df =  pd.concat(clone_df)

huge_df =huge_df[["UL mean SINR", "Decoding time", "UL buffer states
    "]]

"""#Models

## CNN models
"""

class Chomp1d(torch.nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class TemporalBlock(torch.nn.Module):
    def __init__(self, n_inputs, n_outputs, kernel_size, stride,
       dilation, padding, dropout=0.2):
        super(TemporalBlock, self).__init__()
        self.conv1 = torch.nn.utils.weight_norm(torch.nn.Conv1d(
           n_inputs, n_outputs, kernel_size,
                                           stride=stride, padding=
                                              padding, dilation=
                                              dilation))
        self.chomp1 = Chomp1d(padding)
```

```
95          self.relu1 = torch.nn.ReLU()
96          self.dropout1 = torch.nn.Dropout(dropout)
97
98          self.conv2 = torch.nn.utils.weight_norm(torch.nn.Conv1d(
                n_outputs, n_outputs, kernel_size,
99                                                  stride=stride, padding=
                                                        padding, dilation=
                                                        dilation))
100         self.chomp2 = Chomp1d(padding)
101         self.relu2 = torch.nn.ReLU()
102         self.dropout2 = torch.nn.Dropout(dropout)
103
104         self.net = torch.nn.Sequential(self.conv1, self.chomp1, self
                .relu1, self.dropout1,
105                                        self.conv2, self.chomp2, self.relu2
                                               , self.dropout2)
106         self.downsample = torch.nn.Conv1d(n_inputs, n_outputs, 1) if
                 n_inputs != n_outputs else None
107         self.relu = torch.nn.ReLU()
108         self.init_weights()
109
110     def init_weights(self):
111         self.conv1.weight.data.normal_(0, 0.01)
112         self.conv2.weight.data.normal_(0, 0.01)
113         if self.downsample is not None:
114             self.downsample.weight.data.normal_(0, 0.01)
115
116     def forward(self, x):
117         out = self.net(x)
118         res = x if self.downsample is None else self.downsample(x)
119         return self.relu(out + res)
120
121
122 class TemporalConvNet(torch.nn.Module):
123     def __init__(self, num_inputs, num_channels, kernel_size=2,
            dropout=0.2):
124         super(TemporalConvNet, self).__init__()
125         layers = []
126         num_levels = len(num_channels)
```

```
127         for i in range(num_levels):
128             dilation_size = 2 ** i
129             in_channels = num_inputs if i == 0 else num_channels[i
                    -1]
130             out_channels = num_channels[i]
131             layers += [TemporalBlock(in_channels, out_channels,
                        kernel_size, stride=1, dilation=dilation_size,
132                                      padding=(kernel_size-1) *
                                             dilation_size, dropout=
                                             dropout)]
133
134         self.network = torch.nn.Sequential(*layers)
135
136     def  forward(self, x):
137         return self.network(x)
138
139
140 class TCN(torch.nn.Module):
141     def __init__(self, input_size, output_size, num_channels,
            kernel_size, dropout):
142         super(TCN, self).__init__()
143         self.tcn = TemporalConvNet(input_size, num_channels,
                kernel_size=kernel_size, dropout=dropout)
144         self.linear = torch.nn.Linear(num_channels[-1], output_size)
145         self.init_weights()
146
147     def init_weights(self):
148           self.linear.weight.data.normal_(0, 0.01)
149
150     def forward(self, x, channel_last=True):
151         #If channel_last, the expected format is (batch_size,
                seq_len, features)
152         y1 = self.tcn(x.transpose(1, 2) if channel_last else x)
153         return self.linear(y1.transpose(1, 2))
154
155
156 class CausalConvDiscriminator(torch.nn.Module):
157     """Discriminator using casual dilated convolution, outputs a
            probability for each time step
```

```
158         Args:
159             input_size (int): dimensionality (channels) of the input
160             n_layers (int): number of hidden layers
161             n_channels (int): number of channels in the hidden layers (
                    it's always the same)
162             kernel_size (int): kernel size in all the layers
163             dropout: (float in [0-1]): dropout rate
164 
165         Input: (batch_size, seq_len, input_size)
166         Output: (batch_size, seq_len, 1)
167         """
168         def __init__(self, input_size, n_layers, n_channel, kernel_size,
                dropout=0):
169             super().__init__()
170             #Assuming same number of channels layerwise
171             num_channels = [n_channel] * n_layers
172             self.tcn = TCN(input_size, 1, num_channels, kernel_size,
                    dropout)
173 
174         def forward(self, x, channel_last=True):
175             return  torch.sigmoid(self.tcn(x, channel_last)) # self.tcn(
                    x, channel_last)#
176 
177     class CausalConvGenerator(torch.nn.Module):
178         """Generator using casual dilated convolution, expecting a noise
                vector for each timestep as input
179         Args:
180             noise_size (int): dimensionality (channels) of the input
                    noise
181             output_size (int): dimenstionality (channels) of the output
                    sequence
182             n_layers (int): number of hidden layers
183             n_channels (int): number of channels in the hidden layers (
                    it's always the same)
184             kernel_size (int): kernel size in all the layers
185             dropout: (float in [0-1]): dropout rate
186 
187         Input: (batch_size, seq_len, input_size)
188         Output: (batch_size, seq_len, outputsize)
```

```
    """
    def __init__(self, noise_size, output_size, n_layers, n_channel,
        kernel_size, dropout=0):
        super().__init__()
        num_channels = [n_channel] * n_layers
        self.tcn = TCN(noise_size, output_size, num_channels,
            kernel_size, dropout)

    def forward(self, x, channel_last=True):
        return self.tcn(x,channel_last) # torch.tanh(self.tcn(x,
            channel_last))#

"""## LSTM Models
Currently not working on GPU
"""

class LSTMGenerator(torch.nn.Module):
    """An LSTM based generator. It expects a sequence of noise
       vectors as input.
    Args:
        in_dim: Input noise dimensionality
        out_dim: Output dimensionality
        n_layers: number of lstm layers
        hidden_dim: dimensionality of the hidden layer of lstms
    Input: noise of shape (batch_size, seq_len, in_dim)
    Output: sequence of shape (batch_size, seq_len, out_dim)
    """

    def __init__(self, in_dim, out_dim, n_layers=1, hidden_dim=256):
        super().__init__()
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim
        self.out_dim = out_dim

        self.lstm = torch.nn.LSTM(in_dim, hidden_dim, n_layers,
            batch_first=True)
        self.linear = torch.nn.Sequential(torch.nn.Linear(hidden_dim
            , out_dim))#, torch.nn.Tanh())#, torch.linear(hidden_dim
            , out_dim))
```

```
221
222     def forward(self, input):
223         batch_size, seq_len = input.size(0), input.size(1)
224         h_0 = torch.zeros(self.n_layers, batch_size, self.hidden_dim
                )
225         c_0 = torch.zeros(self.n_layers, batch_size, self.hidden_dim
                )
226
227         recurrent_features, _ = self.lstm(input, (h_0, c_0))
228         outputs = self.linear(recurrent_features.contiguous().view(
                batch_size*seq_len, self.hidden_dim))
229         outputs = outputs.view(batch_size, seq_len, self.out_dim)
230         return outputs
231
232
233 class LSTMDiscriminator(torch.nn.Module):
234     """An LSTM based discriminator. It expects a sequence as input
            and outputs a probability for each element.
235     Args:
236         in_dim: Input noise dimensionality
237         n_layers: number of lstm layers
238         hidden_dim: dimensionality of the hidden layer of lstms
239     Inputs: sequence of shape (batch_size, seq_len, in_dim)
240     Output: sequence of shape (batch_size, seq_len, 1)
241     """
242
243     def __init__(self, in_dim, n_layers=1, hidden_dim=256):
244         super().__init__()
245         self.n_layers = n_layers
246         self.hidden_dim = hidden_dim
247         self.lstm = torch.nn.LSTM(in_dim, hidden_dim, n_layers,
                batch_first=True)
248         self.linear = torch.nn.Sequential(torch.nn.Linear(hidden_dim
                , 1), torch.nn.Sigmoid())
249
250     def forward(self, input):
251         batch_size, seq_len = input.size(0), input.size(1)
252         h_0 = torch.zeros(self.n_layers, batch_size, self.hidden_dim
                )
```

```
253         c_0 = torch.zeros(self.n_layers, batch_size, self.hidden_dim
                ).
254
255         recurrent_features, _ = self.lstm(input, (h_0, c_0))
256         outputs = self.linear(recurrent_features.contiguous().view(
                batch_size*seq_len, self.hidden_dim))
257         outputs = outputs.view(batch_size, seq_len, 1)
258         return outputs
259
260 """# Data load"""
261
262 class Df_to_dataset(Dataset):
263   def __init__(self, dataframe):
264
265     self.data_set = torch.FloatTensor(dataframe.values.astype('float
            '))
266
267   def __len__(self):
268     return len(self.data_set)
269
270   def __getitem__(self, index):
271
272     return self.data_set[index]
273
274 train_dataset = Df_to_dataset(huge_df)
275 print(len(train_dataset))
276 batch_size = 200
277
278 device_D = "cuda" if torch.cuda.is_available() else "cpu"
279 device_G = "cuda" if torch.cuda.is_available() else "cpu"
280 kwargs = {'num_workers': 16, 'pin_memory': True} if device_D=='cuda'
        else {}
281
282 train_loader = DataLoader(train_dataset, batch_size=batch_size,
        shuffle=False, **kwargs)
283
284 X_dim = huge_df.shape[1]
285 z_dim = huge_df.shape[1] * 2
286 h_dim = 14 # kernel size
```

```
287
288 lr = 1e-4 # Learning rate
289
290 """# Set models"""
291
292 G = CausalConvGenerator(noise_size=z_dim, output_size=X_dim,
        n_layers=8, n_channel=10, kernel_size=h_dim, dropout=0.2).to(
        device_G)
293 D = CausalConvDiscriminator(input_size=X_dim, n_layers=10, n_channel
        =10, kernel_size=h_dim, dropout=0).to(device_D)
294 '''
295 ######################################
296 # LSTM not currently working in GPU ##
297 ######################################
298
299 #device_D = 'cpu'
300 #train_loader.num_workers = 0
301 #device_G = 'cpu'
302 #G = LSTMGenerator(z_dim, X_dim,hidden_dim =256).to(device_G)
303 #D = LSTMDiscriminator(X_dim, hidden_dim =256).to(device_D)
304 '''
305 def reset_grad():
306     G.zero_grad()
307     D.zero_grad()
308
309 """Generator and Discriminator Optimizers"""
310
311 G_optim = optim.RMSprop(G.parameters(), lr=lr)
312 D_optim = optim.RMSprop(D.parameters(), lr=lr)
313
314 """# Training"""
315
316 # Wasserstein parameters
317 clip_parameter = 0.1
318 n_crit = 5
319 clip_adjustment = 0
320 epochs = 100000
321 num_iter = epochs * len(train_loader)
322 z_len = 1
```

```
323
324 titles = ["SNR", "Decode Time", "BSR"]
325
326 # Allocate memory in device
327 input = torch.FloatTensor(1, batch_size,  X_dim).to(device_D)
328 noise = torch.FloatTensor(z_len, batch_size,  z_dim).to(device_G)
329 fixed_noise = torch.FloatTensor(z_len, batch_size, z_dim).normal_(0,
        1).to(device_G)
330 fake_input = torch.FloatTensor(1, batch_size, X_dim).to(device_D)
331
332 ##########################
333 ####### TRAIN ############
334 ##########################
335 for it in range(epochs): # while GAN hasn't converged
336   data_iter = iter(train_loader)
337   for i in range(len(data_iter)):
338     data = next(data_iter)
339     for j in range(n_crit):
340     ############################
341     # (1) Update D network
342     ############################
343       for p in D.parameters():
344         p.requires_grad = True # to avoid computation
345
346       if clip_adjustment > 10:
347         clip_parameter = clip_parameter + 0.005
348
349       # Weight clipping
350       for p in D.parameters():
351         p.data.clamp_(-clip_parameter, clip_parameter)
352
353       D.zero_grad()
354       G.zero_grad()
355
356       # REAL DATA TO TRAIN DISCRIMINATOR
357       real_cpu = data
358       real_cpu = real_cpu.unsqueeze(0)
359
360       input.resize_as_(real_cpu).copy_(real_cpu)
```

```
        inputv = Variable(input)
        D_real = D(inputv)

        # SYNTH DATA
        noise.resize_(z_len,batch_size,  z_dim).normal_(0, 1)
        noisev = Variable(noise)
        G_sample = G(noisev)

        fake_input.resize_as_(G_sample.data).copy_(G_sample.data)
        fake_inputv = Variable(fake_input)

        D_fake = D(fake_inputv)

        # Dicriminator forward-loss-backward-update
        D_loss = -(torch.mean(D_real) - torch.mean(D_fake))
        if D_loss > -0.01:
          clip_adjustment = clip_adjustment + 1
        else:
          clip_adjustment = 0

        D_loss.backward()
        D_optim.step()

      for p in D.parameters():
        p.requires_grad = False # to avoid computation
      G.zero_grad()
      noise.resize_(z_len, batch_size, z_dim).normal_(0, 1)
      noisev = Variable(noise)

      G_sample = G(noisev)

      D_fake = D(G_sample)

      G_loss = -torch.mean(D_fake)

      G_loss.backward()

      G_optim.step()

```

```
400       #Print and plot every now and then
401       if i == len(data_iter)-1:
402         print('Iter-{}; D_loss: {}; G_loss: {}'
403                 .format(it, D_loss.cpu().data.numpy(), G_loss.cpu().
                        data.numpy()))
404         G_sample = G(fixed_noise)
405         fig, axs = plt.subplots(1,X_dim)
406         fig.set_size_inches(18, 4)
407         fig.suptitle("Data generated")
408         for i in range(X_dim):
409           axs[i].plot(G_sample.cpu().data.numpy()[0,:,i])
410           axs[i].set_title(titles[i])
411         plt.show()
412
413   """# Generation check"""
414
415   gen_len = batch_size
416   noise.resize_(1,gen_len,  z_dim).normal_(0, 1)
417   noisev = Variable(noise)
418
419   synth_serie = G(noisev).cpu().data.numpy()[0,h_dim:-h_dim,:]
420
421   ylabels = ["SNR (dB)", "Decoding Time (us)", "BSR (Byte)"]
422   fig, axs = plt.subplots(1,X_dim)
423   fig.set_size_inches(20, 5)
424
425   for j in range(X_dim):
426     fig.suptitle("Synthetic data")
427     axs[j].set_xlabel("Time (s)")
428     axs[j].set_ylabel(ylabels[j])
429     axs[j].set_title(titles[j])
430     axs[j].plot(synth_serie[h_dim:-h_dim,j])
431   fig.savefig('/content/drive/My Drive/TFG/res/all_clone_synth.png',
        bbox_inches='tight')
432
433   fig.show()
434
435   import random
436   import math
```

```
437
438 random_chunk = random.randint(10, len(train_dataset)-gen_len-2*h_dim
        )
439 real_serie = train_dataset.data_set.numpy()[random_chunk:
        random_chunk+gen_len, :]
440 fig, axs = plt.subplots(1,X_dim)
441 fig.set_size_inches(20, 5)
442 for j in range(X_dim):
443
444
445   fig.suptitle("Real data")
446   axs[j].set_xlabel("Time (s)")
447   axs[j].set_ylabel(ylabels[j])
448   axs[j].set_title(titles[j])
449   axs[j].plot(real_serie[h_dim:-h_dim,j])
450 fig.savefig('/content/drive/My Drive/TFG/res/all_clone_real.png',
        bbox_inches='tight')
451
452 fig.show()
453
454 """#Metrics"""
455
456 example_num = 100
457 list_synth = []
458 list_real = []
459 for i in range(example_num):
460   noise.resize_(1,gen_len,  z_dim).normal_(0, 1)
461   noisev = Variable(noise)
462   list_synth.append(G(noisev).cpu().data.numpy()[0,h_dim:-h_dim,:])
463
464   random_chunk = random.randint(10, len(train_dataset)-gen_len)
465   list_real.append(train_dataset.data_set.numpy()[random_chunk:
          random_chunk+gen_len,:])
466
467 def RMS(time_serie):
468   squares = []
469   [squares.append(value**2) for value in time_serie]
470   sum_of_squares = sum(squares)
471   rms = math.sqrt(sum_of_squares/len(time_serie))
```

```
472     return rms
473
474   mean_synth = []
475   mean_real = []
476   for i in range(3):
477     real_rms = []
478     synth_rms = []
479     [real_rms.append(RMS(item[:,i])) for item in list_real]
480     [synth_rms.append(RMS(item[:,i])) for item in list_synth]
481     mean_real.append(sum(real_rms)/len(real_rms))
482     mean_synth.append(sum(synth_rms)/len(synth_rms))
483
484   print("The mean RMS of the real time series is: ", mean_real)
485   print("The mean RMS of the synthetic time series is: ", mean_synth)
486
487   def PAR(time_serie):
488     peak = max(time_serie)
489     return abs(peak)/RMS(time_serie)
490
491   mean_synth = []
492   mean_real = []
493   for i in range(3):
494     real_par = []
495     synth_par = []
496     [real_par.append(float(PAR(item[:,i]))) for item in list_real]
497     [synth_par.append(PAR(item[:,i])) for item in list_synth]
498     mean_real.append(sum(real_par)/len(real_par))
499     mean_synth.append(sum(synth_par)/len(synth_par))
500
501
502   print("The PAR of the real time serie is: ",mean_real)
503   print("The PAR of the synthetic time serie is: ",mean_synth )
504
505   def mean_RMS(time_serie):
506     rms_list = [[],[],[]]
507     for i in range(len(time_serie)):
508       squares = []
509
510       [squares.append(value**2) for value in time_serie(i)]
```

```
511         sum_of_squares = sum(squares)
512         rms = math.sqrt(sum_of_squares/len(time_serie(i)))
513         rms_list(i).append(rms)
514       mean_list = []
515       for i in range(len(time_serie)):
516         mean = sum(rms_list(i))/len(rms_list(i))
517         mean_list.append(mean)
518
519       return mean_list
520
521   print("The mean RMS of the real time series is: ", mean_RMS(
          list_real))
522   print("The mean RMS of the synthetic time series is: ", mean_RMS(
          list_synth))
523
524   def PAR(time_serie):
525     peak = max(time_serie)
526     return abs(peak)/RMS(time_serie)
527
528   real_par = []
529   synth_par = []
530   [real_par.append(float(PAR(item))) for item in list_real]
531   [synth_par.append(PAR(item)) for item in list_synth]
532
533   print("The PAR of the real time serie is: ", sum(real_par)/len(
          real_par))
534   print("The PAR of the synthetic time serie is: ",sum(synth_par)/len(
          synth_par) )
```