MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

LEARNING DYNAMIC BRAIN CONNECTIVITY FROM EEG SIGNALS WITH A SPATIAL-TEMPORAL GRAPH CONVOLUTION APPROACH

LICENSE THESIS

Graduate: Cristian-Ioan BLAGA
Supervisor: Prof. Dr. Eng. Mihaela DÎNȘOREANU

2020

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The understanding of the brain and how it functions has baffled and intrigued humans for thousands of years, with the first known evidence dating back to the ancient greeks. Early philosophers believed that, since the body is just a mere vessel of the soul, its actual location could be in the brain. Thousands of years have passed since this philosophical argument and a lot of discoveries were made in this area but we are still far from fully understanding how the brain works.

Nowadays, this study is encapsulated in the area of Neuroscience which is an interdicsiplinary science focused in understanding the properties of not only neurons, but also of the neural circuits they form, with the goal of gaining an insight in how different mechanisms are actually performed. The final goal of this study field is actually being able to answer the question *"What is consciousness?"* but, until an answer to that question can be found, neuroscientists deal with understanding the cognitive processes that occur such as learning, perception and so on.

This field is studied by people with multiple backgrounds, resulting in a mixed view and focus of the intentions behind making certain discoveries. This thesis has been focused on the aspect of Computational Neuroscience, which is the subfield focused in how the brain functions from the perspective of a machine, with the goal of actually being able to replicate and simulate the functionality of the brain.

1.1 Project context: Visual Recognition

In this thesis, the focus is on the brain functionality involving the visual recognition process. Visual recognition is defined as the ability to assign multiple labels to an object, such as different shape properties, color attributes and so on, regardless if the object presented is rotated, has a varying size or is deformed.

From an evolutionary point of view, vision played a crucial role in the evolution of species in the history of Earth dating back to millions of years. The first known vision process that a species was able to perform was to actually sense light. From this starting

point, the vision of animals has improved drastically during the history, with the most significant moment being when vision was used not only as a locomotion and navigation tool, but also as a way to avoid predators.

Nowadays, for humans, both vision and visual recognition are processes that can be done with a minimal effort in normal conditions. At the speed of thought, a human can not only identify the object, but can also analyse the properties of that particular object, regardless of any identity preserving transformations that object has suffered. Therefore, the visual recognition process that happens in the brain is actually invariant with respect to viewing conditions. Moreover, it was also discovered that this ability is also shared with our primate cousins, with the latter being able to accurately identify the identity of an object presented in their central visual field [10].

This ability of the brain to perform the vision task regardless of the viewing conditions is what actually the Neuroscience field tries to understand. This is known as the *"invariance problem"*, which currently represents the bottleneck of performance for any computer vision recognition system [10].

In this context, the importance of understanding the visual recognition process that the brain can perform with ease actually gains practical meaning. By understanding the underlying concepts and mechanisms that conduct the recognition task, we will be able to simulate artificially the vision process and solve one of the misteries of neuroscience, which can be applied in a large number of fields, varying from self driving cars to cancer identification from X-ray images.

1.1.1 Thesis Structure

The thesis is structured as follows:

- **Chapter 2** displays the objectives of this project, alongside the motivation to why this problem was chosen.
- **Chapter 3** covers the bibliograhic research performed in order to provide a basis for the architecture of the thesis.
- **Chapter 4** highlights the conceptual architecture of the project, focusing on the the logical pipeline of processing and learning, with a detailed explanation of each component and the reasoning behind them.
- **Chapter 5** presents the detailed architecture and implementation of the solution, having as a focus both the actual flow inside the application, but also the challenges and design decisions that occurred during the development process.
- **Chapter 6** presents the obtained results and discusses their significance with respect to both other approaches and their interpretation from a Neuroscience perspective.
- **Chapter 7** provides the necessary steps for a user to be able to use the solution.

- **Chapter 8** summarizes the obtained results, focusing on the personal contribution to the field of Neuroscience and discussing further approaches that may be studied.

Chapter 2

Project Objectives and Specifications

Visual recognition is a topic that is studied from multiple perspectives. Approaches vary, with different studies using either the Local Field Potential, recorded using an electrode inserted invasivaly into the V1 region of the Visual Cortex [11], or the ElectroEncephalogram, which is a non-invasive method of recording the electric potential of the brain at the surface of the scalp. In this context, this thesis makes use of the EEG signals recorded on human subjects as a result of an experiment conducted by the Transilvanian Institute of Neuroscience [7].

2.1 Motivation

At the Technical University of Cluj-Napoca, in collaboration with the Transilvanian Institute of Neuroscience, this problem has been studied before by the Knowledge Engineering Group with a solution that is capable of extracting the Functional Network from EEG signals using a statistical approach [2] [3]. The reasoning behind extracting a Functional Network is the fact that it represents a set of dependencies between the brain regions during the image recognition process. Their premise is that, by representing and understanding the overview image of the brain correlations, observations might be extracted that could lead to a broader understanding of the visual recognition process.

Following this reasoning, this thesis tries to apply the same study principle in a different manner. Considering the brain as a machine with multiple states, a network of correlations at a certain point in time represents a snapshot of the brain's activity during that time frame with respect to the state that is in. By extracting these networks at a granularity level small enough to capture all the meaningful states, a set of Dynamic Networks can be built which represent an overview of the evolution of the brain states that occur during the image recognition task. In order to be able to extract these networks, an insight of the brain's activity must be provided. In this thesis, the insight corresponds to the EEG signal recorded during the visual recognition process and the generation of the Dynamic Networks is performed using the latent information of a Deep Learning model

able to learn the properties of multivariate correlated signals. The development of both the conceptual and implementation aspects of the current solution have been done as a joint effort between my colleague, Vlad-Cristian Buda [12] and I.

However, the problem that arises here for both the Functional Networks and the Dynamic Networks is the fact that both of these types of networks are actually hard to validate. Considering our current understanding of the brain, it is rather hard to be able to inspect these two results and make a pragmatic analysis of whether they actually represent something meaningful or not. However, in order to solve this issue, the validation problem is tackled by the Knowledge Engineering Group in two steps. First of all, the results are studied from the Neuroscience perspective to see if they contain processes that are known with respect to the visual recognition task. Afterwards, the results are to be studied with other machine learning approaches in order to see if they hold any discriminatory information that could help, for example, a classifier perform at its best.

2.2 Project Objectives and Challenges

In the context of Dynamic Networks, the objectives of this thesis can be defined as follows:

- Development of a machine learning model capable of identifying the necessary structural parameters that represent the granularity level of the extraction of Dynamic Networks.
- Creating a processing pipeline using a Deep-Learning approach able of extracting a correlation network based on the timeanalysis of multivariate signals.
- Implementing the means for visualization and manipulation of the Dynamic Networks.
- Perform a study of the extracted Dynamic Networks with respect to certain statistical properties they might posses.

With respect to these objectives, certain challenges appear. From the perspective of the Computer Science field, the challenges are represented by identifying, defining, finetuning or organizing multiple Machine Learning approaches that could prove to be meaningful in the context of this thesis. Moreover, considering the limited hardware resources available, it is important to note here that this aspect should also be taken into consideration when developing the solution, with a lot of optimization aspects needing to be covered in order to actually be able to use the models.

Furthermore, the challenges that appear with respect to the Neuroscience field are represented by understanding not only the data provided by the experiment, but also the methodology and the context in which it took place. Data analysis and preprocessing are essential steps in any Machine Learning application and can not be properly done without

a broad understanding of the field the method is applied in. Moreover, certain architectural or design aspects of the solution might be influenced both by the experiment and the field in which the solution is to be used.

Chapter 3

Bibliographic research

3.1 Functional Networks

A major challenge in Neuroscience is understanding the functional purpose of the connections in the brain consisting of billions of neurons, interconnected at different hierarchy levels, forming hubs of connections at various levels of modularity, resulting in a complex hierarchical functionality structure. An overview of the brain's hierarchy modelling can be seen in figure 3.1.

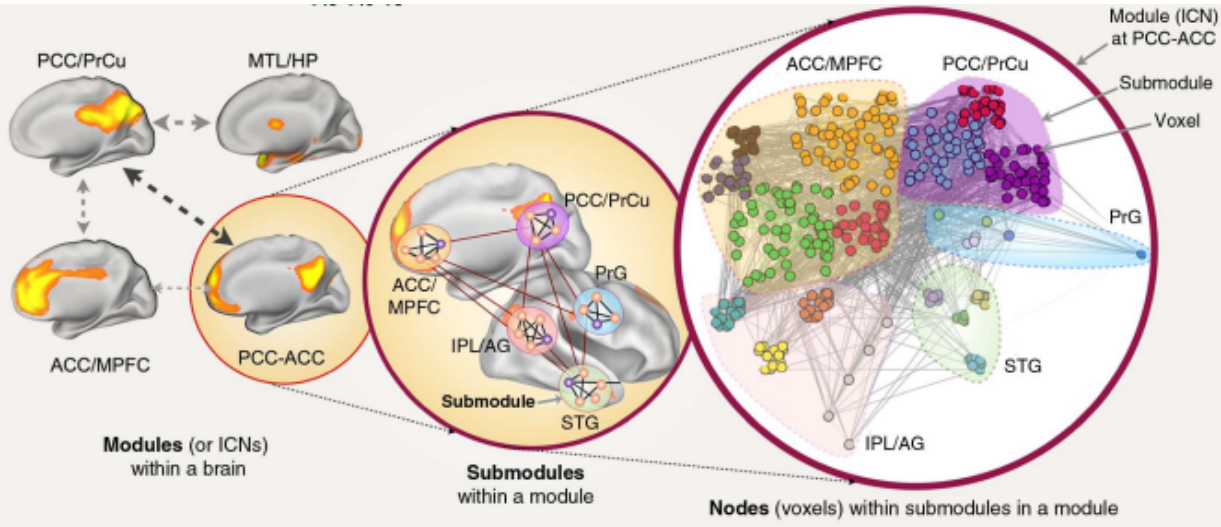


Figure 3.1: A representation of the functional hierarchy of the brain [1].

In order to be able to understand, model and replicate this structure, researchers have come up with the term of *Functional Networks*, representing an empirical representation of the brain's architecture as a neuronal network, integrating information from

multiple anatomical and physiological aspects. Having a base representation of the brain, researchers can study the phenomena present in this complex structure and extract information that might give insight into how the corresponding functionality occurs. More concretely, by answering questions such as *"What and how is information distributed over the network?"* or *"How are edge strengths distributed with respect to the brain's state?"*, researchers can broaden the understanding of the brain by performing an analysis over the functional structure [1].

In this context, in [1], multiple types of Functional Networks are presented based on the type of connectivity they capture. More concretely, they define the following connectivities:

- **Structural Connectivity**, focused on capture the significance of the atomical links in the brain.
- **Functional Connectivity**, used for defining the undirected statistical dependencies between brain regions.
- **Effective Connectivity**, representing the directed causal relationships that occurr in the brain.

In this context, as specified previously, the purpose of this thesis is the extraction of a sequence of Functional Networks from an EEG signal that represent the Effective Connectivity properties with respect to the visual recognition process at different moments during an experiment. Since the result will be a temporal sequence of networks, they are to be named **Dynamic Networks**.

3.2 Previous Approach

The generation of Functional Networks has been studied previously in the Knowledge Engineering Group by our colleagues, Dan-Alexandru Dumitru [2] and Emanuela-Bianca Ceuta [3]. In their thesis, they have studied the means of generating Functional Networks using a statistical approach at a trial level.

More concretely, in their work, they have made use of the **Scaled Correlation** function. In [13], this function is defined as the average short-term correlation over a set of data consisting of a temporal component, such as timeseries. The novelty of this function comes from the fact that is able of filtering out the slow components of a signal, meaning that, in the context of brain waves, it is capable of identifying components in the high frequency range such as beta waves.

In this context, in [2] and [3], scaled correlation was applied on EEG signals based on pairs of electrodes from the EEG headset. The result was a weighted graph with the weights represented by the scaled correlation coefficient. After a filtering step, the final network layout can be seen in figure 3.2.

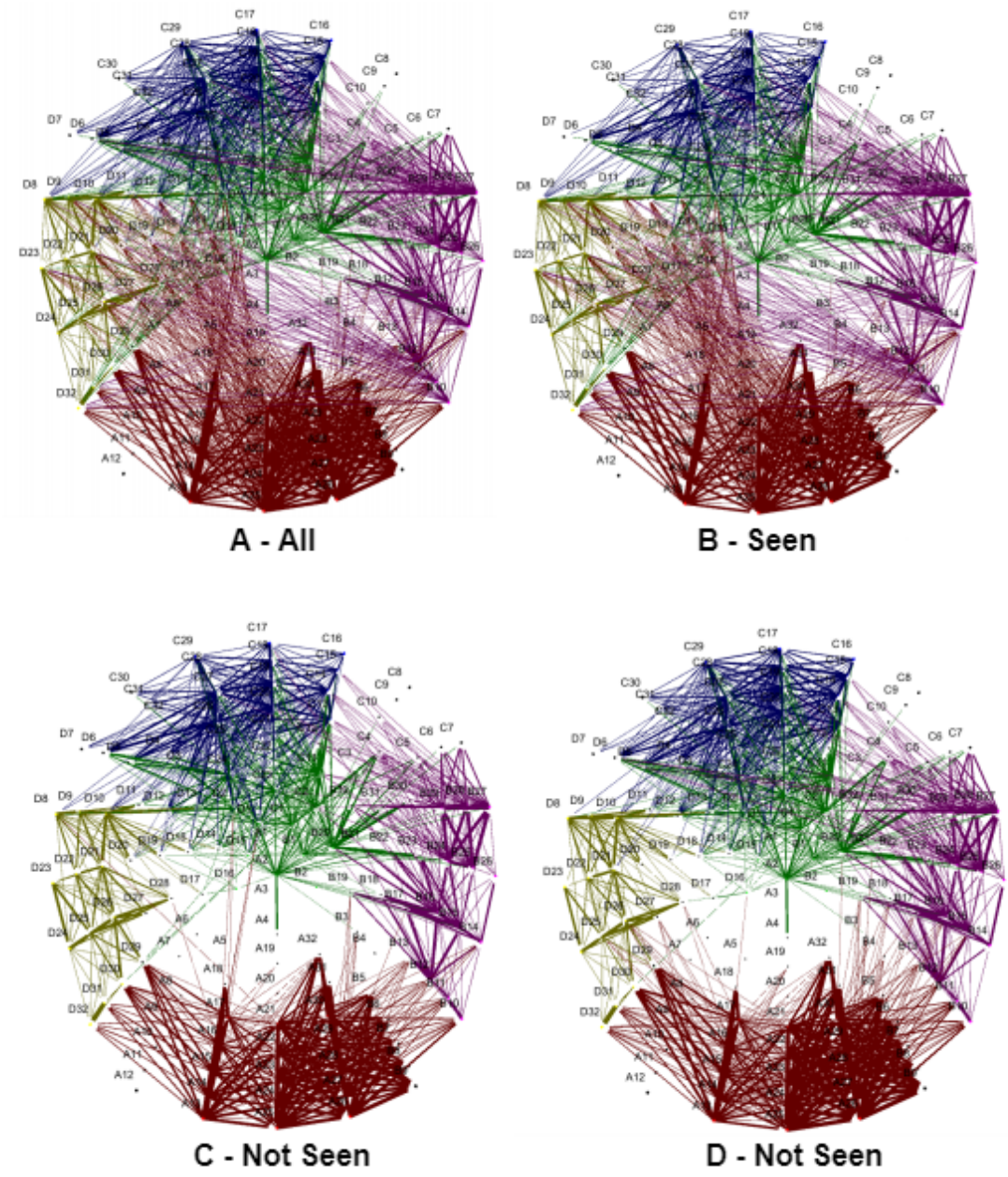


Figure 3.2: Functional Networks based on response classes [2] [3].

However, there were two problems with this approach. First of all, as it can also be seen from the figure, two response classes are quite similar even if the activity of the brain is not necessarily the same. This fact indicates that this approach was not capable

of extracting the discriminatory information that could help make a distinction between the two classes.

In this context, these results were afterwards used as the basis dataset for the binary classification model implemented by our colleague, Liana-Daniela Palcu, namely the Deep Graph Convolutional Neural Network (DGCNN) [14], which is capable of making an accurate classification on the problem she has studied during her thesis. However, for the generated Functional Networks, her model was not able to reach a performance better than random guessing, with her conclusive analysis being that the set of functional networks from two classes generated in this manner do not hold any specific information that could help the model make a distinction between the two.

The second problem with this approach was the fact that, for a trial, no matter the length, only one Functional Network was extracted. This is quite problematic because of the fact that the brain, when viewed as a state machine, passes through multiple states during a specific timeframe and, by ignoring this aspect, the necessary insight for understanding how cognitive processes are formed in the brain can not be provided.

3.3 Graph Neural Networks

Considering that the results of the experiment are a series of EEG signals, each representing the activity of a brain region from where the electrode was placed, the correlation of these signals actually creates a graph-like structure of dependencies. In this context, we have decided to actually use a Deep Learning approach that can handle graph structured data in order to solve the task of generating Dynamic Networks.

In [15], multiple models that fit into this machine learning category are presented. However, since the values analysed evolve in time, we needed to find a model that was capable of analysing both the spatial correlation dependencies of the signals and the temporal evolution component. Taking into consideration this aspect, the machine learning model we are looking for fits in the category of **Spatial Temporal Graph Convolutional Networks (STGCN)**. Based on [15], two models are suited for this task, namely:

- STGCNs for Skeleton-Based Action recognition [4].
- Graph WaveNet for Deep Spatial-Temporal Graph Modelling [9].

The **STGCN for Skeleton-Based Action Recognition** is defined in [4], where a novel approach for classifying a sequence of frames under the form of a video for identifying the action performed by a human is presented. For this, from each video frame, the human skeleton is extracted (by skeleton is meant joints) which are then processed both spatially and temporally using Graph Convolutional Networks as seen in figure 3.3.

However, the problem that appears with this approach is the fact it assumes a **local spatiality constraint** that is to be respected by the data. More concretely, it makes the assumption that the points close to eachother in a frame are actually correlated. Even

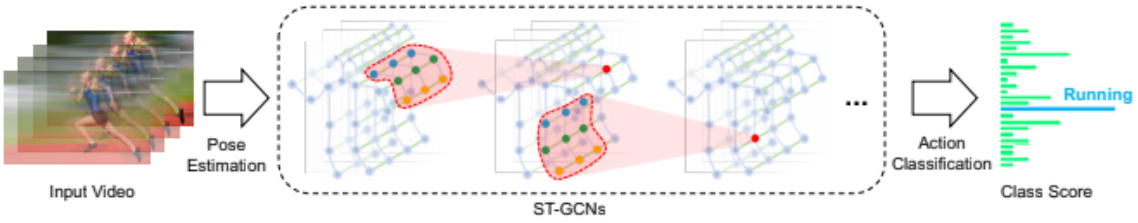


Figure 3.3: Architecture of the STGCN model for predicting the action class [4].

if this fact is true in the context of image recognition, it is not suitable for the current scenario since the correlation of the brain regions is not location based.

On the other hand, the **Graph WaveNet** model proves to be fit for the task at hand. As described in [9], Graph WaveNet is a novel approach to tackling the problem of timeseries forecasting for data that is spatio-temporal correlated, reaching state-of-the-art performance on the traffic prediction problem.

The power of the Graph WaveNet model actually comes from the fact that is the joint result of two powerful machine learning models, namely **WaveNet** and **DCRNN**. As described in [16], the WaveNet model is one of the top timeseries analysis models, being capable of extracting meaningful information from audio samples and reaching state-of-the-art performance on text-to-speech problems. Its performance comes from its unique structure, using dilated causal convolutions in order to cover long-range temporal sequences with a small number of convolutional layers.

Moreover, the **Diffusion Convolutional Recurrent Neural Network (DCRNN)**, presented in [5], used to reach state-of-the-art performance on the traffic prediction problem until Graph WaveNet took its place. The novelty of the DCRNN model comes from the way it models the **unconstraint** spatial dependency between the signals by interpreting them as a a diffusion process, modeled using different orders of neighbourhood from the learnable adjacency matrix. The architecture of the DCRNN model can be seen in 3.4.

As a result of these two approaches, **Graph WaveNet** models the temporal dependencies using the **WaveNet** approach and the spatial dependencies using the **DCRNN** approach. However, it makes slight changes to these approaches, like changing the way the learnt matrix from the DCRNN model is to be generated.

Moreover, in a recent paper [17], the performance of the **Graph WaveNet** was enhanced, reaching an even better performance. Based on multiple tests on the model, the authors changed some structural aspects regarding the architecture of the Graph WaveNet together with some runtime parameters. Some examples of the small modifications performed are the way the residual connection are computed or the value of the learning rate.

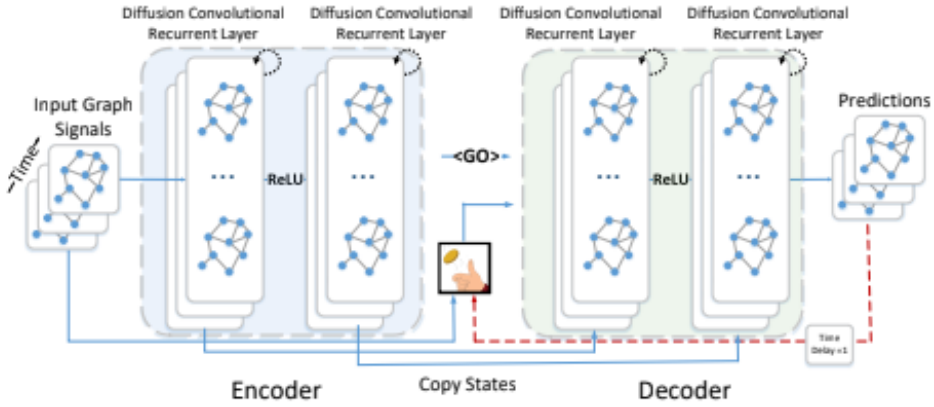


Figure 3.4: Architecture of the DCRNN model [5].

Taking all these aspects into consideration, Graph WaveNet can solve the problem of generating Dynamic Networks by being capable of modelling the spatial dependencies of multiple signals evolving in time, generating a **hidden dependency self-adaptive learnt matrix** which actually represents a Functional Network, with no spatiality constraint imposed. Therefore, by applying the Graph WaveNet model on a series of time snapshots over multiple EEG signals, it can build a Dynamic Network representing the evolution of the brain states during the image recognition process.

3.4 Timeseries validation

Considering the fact that Graph WaveNet is a timeseries predictive model, certain choices regarding how the performance of the model is validated needed to be made in order to ensure that the Functional Network learnt in a self-supervised manner actually holds meaning. For this, two problems were encountered, which are described in the following sections.

3.4.1 Timeseries validation scheme

As in any machine learning task, when training and choosing a model, one of the most important aspects that needs to be taken care of is the ability of the model to generalize what it has learnt on previously unseen data. In this context, what is usually done is taking a random subset from the dataset, called the **cross-validation** set, and use this subset only for testing purposes. However, this type of process makes the assumption that the examples that can be placed in the cross-validation set are all extracted from an

underlying distribution, which, in the case of timeseries, is not true. More specifically, in the case of timeseries analysis, any observation in the set depends on all the past observations and the rudimentary generator process of the timeseries data may evolve over time [6].

In this context, in [6], the problem of choosing the right validation scheme for a timeseries distribution is studied, not only from the perspective of choosing the correct cross-validation set, but also taking into consideration how the **K-fold cross-validation** mechanism can be performed on timeseries data.

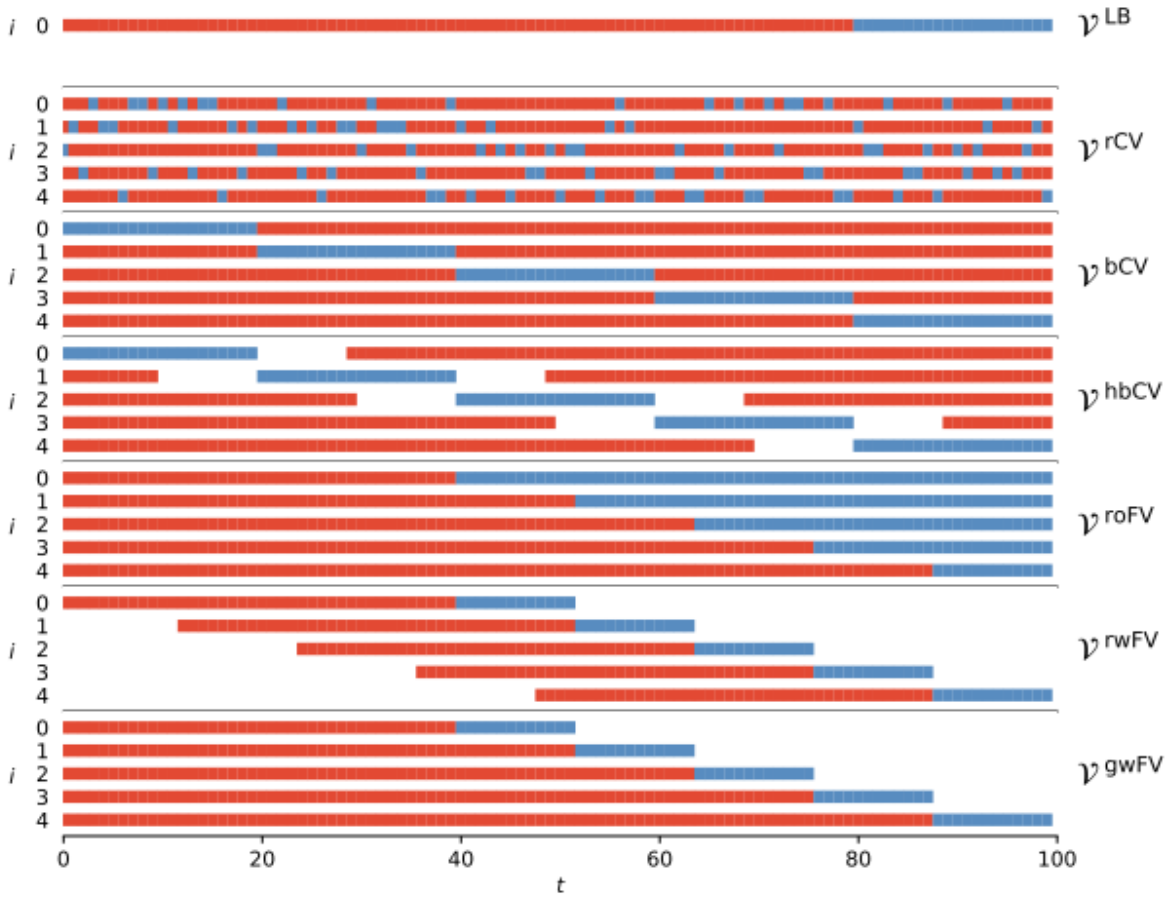


Figure 3.5: Validation schemes for timeseries data [6].

More concretely, the author of the paper proposes a total of 7 validation schemes that are to be studied in order to figure out which is the best approach. The schemes can be viewed in figure 3.5. The schemes are the following:

- **LB**: validation using the last block from the data.

- **rCV**: multiple validations using random cross-validation.
- **bCV**: multiple validations using blocked cross-validation.
- **hbCV**: multiple validations using h-blocked cross validation. The h-block principle represents the fact that, when choosing a block of cross-validation, the signal around the chosen block will be ignored for h steps at both ends.
- **roFV**: multiple validations using a rolling-origin Forward Validation approach. The size of the train set increments at each run with the size of the cross-validation set being decreased at each run.
- **rwFV**: multiple validations using a rolling-window Forward Validation approach. The size of the train and cross-validation remain constant, but the two sets are shifted forward in time for each run.
- **gwFW**: multiple validations using the growing-window Forward Validation approach. Similar to roFV, with the difference that the size of the cross-validation set remains constant.

In order to identify the optimal approach to validating timeseries data, the author has studied multiple problems of timeseries predictions using both synthetic datasets and real-world datasets. Using the *Mean Squared Error* function as a loss metric, his study concluded that the best validation scheme in the context of timeseries prediction is actually the **rolling-origin Forward Validation (roFV)** approach. Therefore, for Graph WaveNet, the thesis uses the roFV approach for validating the model.

3.4.2 Forecast Accuracy

The study of the forecast accuracy of a predictive model proves to be a challenging task, with many possible functions that might have misleading interpretational results. Therefore, in [18], this problem tries to be solved by studying different relative measures that compute a forecast measure with respect to another type of forecast, done on the same data, used as a benchmark. More concretely, the benchmark used by the authors is the **naive forecast**, where the forecasted value represents the value of the last seen observation in the input set.

Based on this type of benchmarking, the authors study different already existing metrics, such as measures based on relative errors, like **Mean Relative Absolute Error (MRAE)**, or relative measures, defined as being the fraction between the actual measure and the same measure performed on the benchmark forecasting method. Their focus on these measures is to highlight the potential problems they might have, with measures based on relative errors having an undefined mean with infinite variance distribution and relative measures being constraint by a number of data specific factors.

In this context, they propose an error measure independent with respect to the scale of the data, called **Mean Absolut Scaled Error** [18], which is able to compare the prediction accuracy of a forecasting model with the one of any benchmarking method. By using as a scale factor for the measurement the in-sample **Mean Absolute Error** from the benchmark forecasting method, the MASE result gives the following interpretability of the forecasting result:

- if it is smaller than 1, then the result of the forecast is better than the one of the benchmark forecast
- if it is equal to 1, then the result of the forecast is the same as for the benchmark method
- if it is greater than 1, then the result of the forecast is worse than the benchmark method

Taking this into consideration, the **MASE** error proves itself to be a meaningful error, having an easily interpratable scale, with the only problem that may cause inconsistancies being when all the observations are 0. Thus, the MASE scaled error will be used in the thesis for further validating the result of the prediction of the Graph WaveNet model.

Chapter 4

Analysis and Theoretical Foundation

This chapter is focused on describing the conceptual pipeline of the proposed solution for the studied research problem. The conceptual architecture is presented in the section 4.1 , followed by a description of the EEG dataset used by this project and an in-depth explanation of all the logical components of the proposed solution. The chapter concludes with an analysis of the obtained results and a concrete use case of the generated results.

4.1 Conceptual Architecture

The conceptual architecture of the proposed solution can be seen in figure 4.1. Because of the nature of the problem, the development has been done as a joint work between my colleague [12] and I. In the following sections, a detailed description of each component will be presented, focused on my contribution to that part.

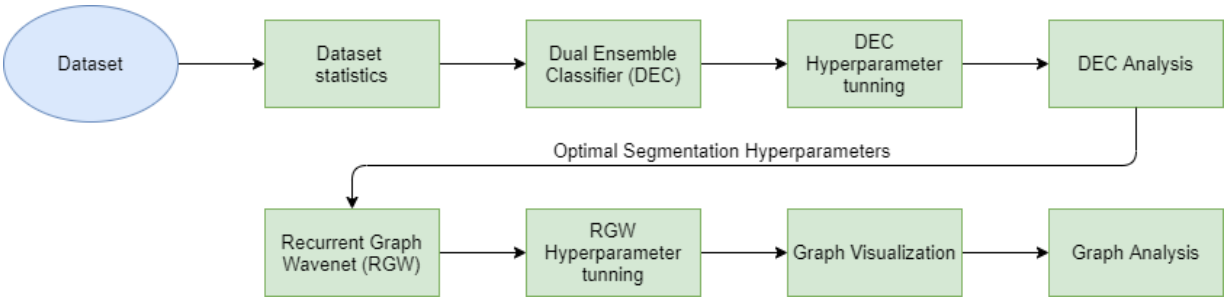


Figure 4.1: Conceptual architecture

4.2 Dataset

As it is the case in any Machine Learning problem, the most important component of the entire solution is the dataset. The proper generation, understanding and use of a dataset represents one of the most critical factors in obtaining the desired results.

The dataset used by this problem has been generated and provided by *Dr.Eng. Raul C.Muresan*, director of Experimental and Theoretical Neuroscience Laboratory at the Transylvanian Institute of Neuroscience, Cluj-Napoca. The experiment's focus was recording the brain's activity during the task of object recognition on different subjects exposed to multiple stimuli. The methodology of the experiment is concretely explained in the [7].

4.2.1 Stimuli generation

The stimuli used in this project have as a basis 30 different depicted objects representing things that are quite trivial to identify. However, in order to make them suitable for this experiment, they were converted to a lattice of dots which were then deformed. The controlled deformation of the dots lattice has been done with respect to the local contour density of the image, resulting in a parametrized deformation which is able to make the depicted object more or less visible using a gravitational constant. Based on the value of the gravitational constant, the deformation is defined as barely visible for small values and highly visible for large values. The generation process is explained in more detail in [7].

Having the transformation defined, seven gravitational constants have been chosen, starting from 0.0 until 0.3, with an incremental factor of 0.05. The result was a set of 7 pictures at different deformation levels for each object, summing up to a total 210 pictures used as the basis of the experiment. A deformation example can be seen in figure 4.2.

4.2.2 Experiment

The experiment involved 11 subjects which had the task of identifying the depicted object in the deformed picture presented. The pictures were presented in a block manner, based on the value of the gravitational constant. Thus, the first block contained all the objects at a gravitational constant of 0 (no objects visible) and the last block contained all the objects at a gravitational constant of 0.3 (objects clearly visible). It is worth mentioning that, whenever a subject was presented a picture, this situation is denoted as a **trial**. Therefore, for each subject, there were 210 trials, divided into 7 blocks, with the trial order being random in each block.

Before and during a trial, multiple steps were taken, which are denoted as **events**. These events follow the same structure throughout all trials. At first, the subject's viewing point is fixed into the middle of the screen. Next, the picture is shown to the subject. At last, the response of the subject is recorded both mechanically and verbally. The response

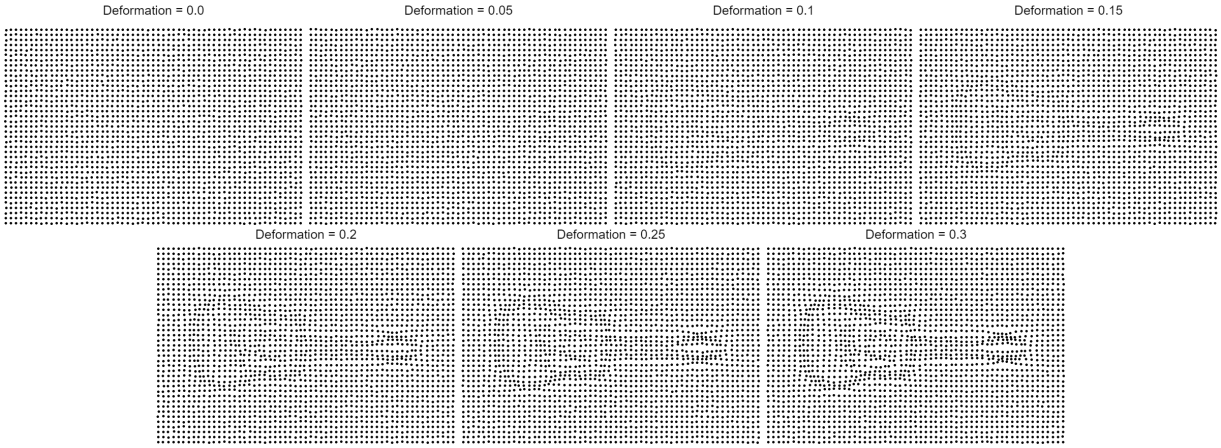


Figure 4.2: Deformation example for a guitar [7].

possibilities are **seen**, **uncertain** and **not seen**, each of them being coded with an unique code event. A detailed explanation of the event codes is further described in table 4.1.

The brain's activity during the experiment was recorded using an EEG headset. More concretely, the Biosemi 128 Cap was used which can be seen in figure 4.3.

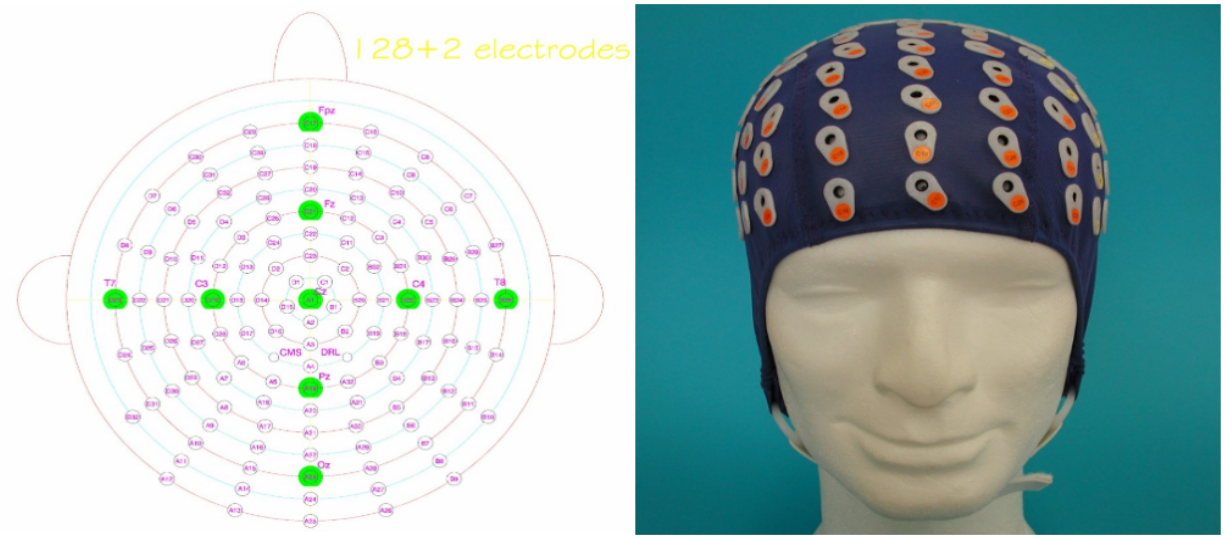


Figure 4.3: Biosemi 128 cap head placement and electrode placement [8].

The cap records the brain's activity based on the electric potential that each electrode measures. These electrodes will be further referenced as **channels**, with the number of channels being equal to the number of electrodes on the cap. The measurements take place at each millisecond and are recorded individually as a floating point value in a binary

Table 4.1: Experiment event codes.

Event code	Event name	Description
128	-	A red cross presented in the middle of a white background with the purpose of fixing the subject's position towards where the photo will be centered for 1500-2000 ms.
150	Mask	A white background is displayed for 500 ms.
129	Stimulus	The stimulus is presented to the subject until they give their response.
1,2 or 3	Response	The mechanical response of the subject done by pressing a button. The possible responses are: • 1 = seen • 2 = uncertain • 3 = not seen
131	-	A message appears on the screen asking the subject to verbally specify the object he has seen.
132	-	The subject presses the space bar in order to continue to the next trial.

file, individually for each channel and subject.

4.3 Dataset statistics

The first step in working with the provided dataset is the study and understanding of the underlying statistical and distribution properties of the trials. In this context, an analysis has been performed with respect to the trial's length and the response given by the subjects which is described in the following sections.

4.3.1 Overall trial length analysis

The first analysis performed was solely based on the recorded trial's length over all the 11 subjects. The histogram of the distribution of this property can be viewed in figure 4.4. As it can be seen, the distribution follows a long-tail pattern, having most of the values around the start of the histogram. The mean value of trials length is around 7048 milliseconds with a median of 3874 milliseconds. However, as it can be seen from the histogram, there are also some outliers, with some trials having close to 80000 milliseconds in length.

More specifically, the shortest trial has a length of 627 milliseconds, whilst the longest trial has a length of 83414 milliseconds. This is most notable when looking at the mean of the length, which can be seen in the figure as a red vertical line.

However, these properties are not necessarily respected when the analysis is performed at subject level. In figure 4.5, the trial histogram on two different subjects is presented. As it can be seen, there is a clear difference between the two subjects, with the first subject giving responses much faster than the second subject. More concretely, what can be observed is the fact that the behavior of the two subjects directly influences the outcome of the experiment, with certain subjects taking a longer time to respond, fact which explains the long tail distribution that can be seen in figure 4.4. Moreover, the behavior difference can be clearly seen by inspecting the mean of the original distribution, which is also displayed in the figure 4.5, with the first subject having most of the trials at the left of the mean, whilst the second one having most of the trials at the right.

4.3.2 Trial length analysis based on response

As it was observed in the subsection 4.3.1, the trial distribution changes based on the behavior of the subjects and their response time. However, the distribution is also susceptible to the response type. More concretely, as specified in subsection 4.2.2, there are 3 response types. Their individual distribution with respect to the trial length can be observed in figure 4.6.

More concretely, what can be observed here is the fact that, for the *seen* trials, most of the trials are located to the left of the mean of the original distribution, with very few outliers being present to the right. The majority of the trials take place in under 5000 ms, which is expected since, if a picture is easily seen, then not much thought goes into figuring out what is presented. Moreover, the distribution is also logical. Since a picture can be easily identified by a subject, it follows logically that the same picture should be identified easily by another subject.

However, this observation does not hold for the *not seen* trials. As it can be seen, the number of trials present at both the right and left of the mean are almost equally distributed. This can be explained based on the way in which the pictures are created. Some of the pictures that fall into this category have little or no information whatsoever in them (for small gravitational constants). For this pictures, the subject can not be

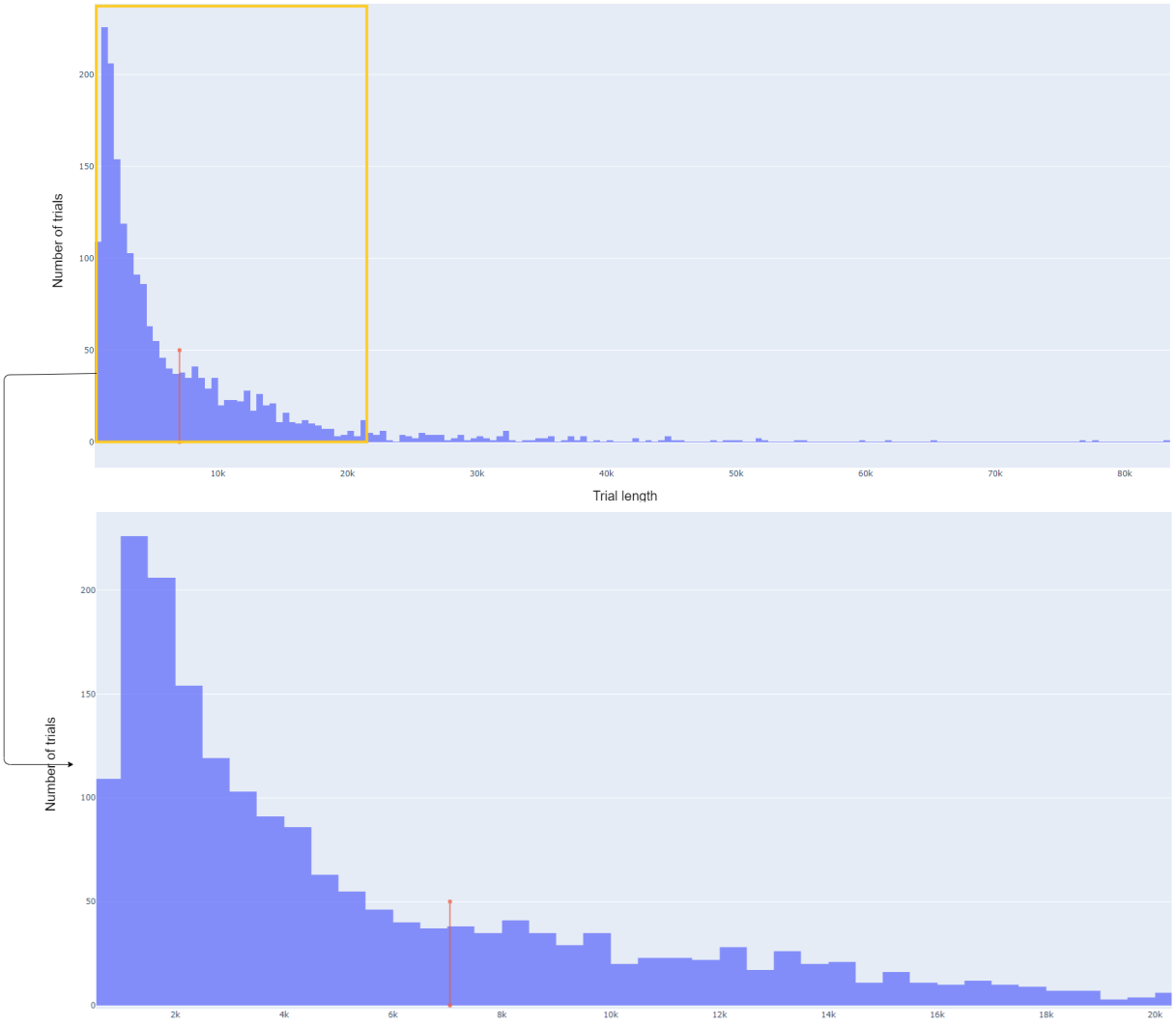


Figure 4.4: Trial length histogram. The red vertical line represents the mean.

certain instantly that there is an object in the picture that he could identify. This delay in the exploration phase of identifying if there is a depicted object explains the distribution present in this histogram.

Furthermore, the exploratory phase can also explain the distribution of the *uncertain* trials. For them, the picture holds information that makes the subject believe there is something, but not enough information for them to be capable of concretely identifying the depicted object (for medium gravitational constants). Thus, not only there are fewer trials at the left of the mean, but there are a lot more outliers at the right of the mean with trial lengths in the order of tens of thousand of milliseconds.

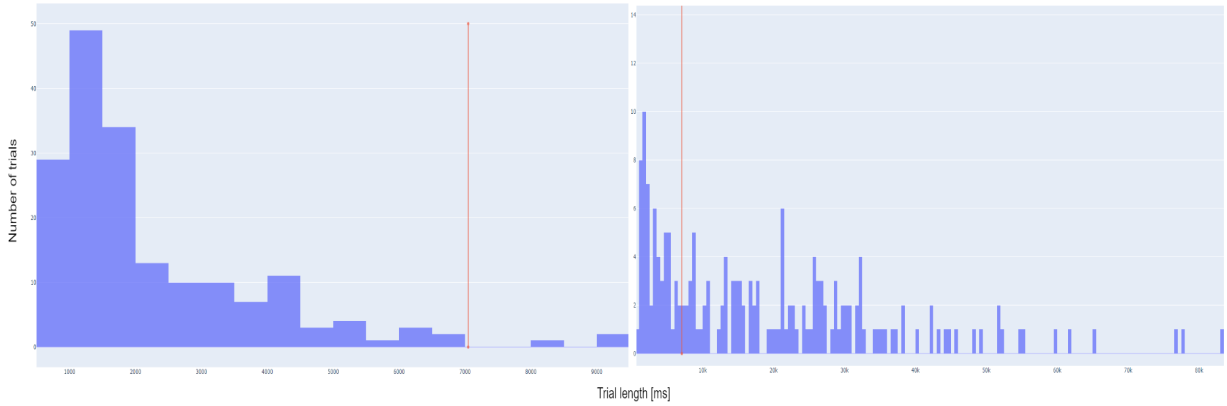


Figure 4.5: Trial histogram on subject 2 (Left); Trial histogram on subject 7 (Right); The red vertical line represents the mean of the entire set.

4.3.3 Response distribution

One more important aspect to note regarding the trials is the response distribution. More concretely, we are interested in the number of trials per response. The distribution can be viewed in figure 4.7.

As it can be seen from the figure, the predominant response class is the *seen* class, having 979 trials, the least predominant class is the *uncertain* class having 371 trials and, somewhat in the middle, the *not seen* class with 630 trials. Considering this, it can be observed that the number of trials per response type follows an imbalanced class distribution, which will prove itself difficult to work with in the following sections.

4.4 Dual Ensemble Classifier

As it was described in 2.1, the current solution's goal is the ability to extract multiple snapshots in time of the activity occurring in the brain during the trial. More concretely, we need to have the ability to be able to split the trials into windows on which the correlogram generation will take place. Thus, the *window size* needs to be chosen. Moreover, in order to have the highest chance of capturing the processes happening during the image recognition task, these windows will be created with the overlapping principle. Therefore, a *window offset* parameter will also need to be chosen.

Therefore, what needs to be found are two parameters that influence the trial segmentation process:

- Window size
- Window offset

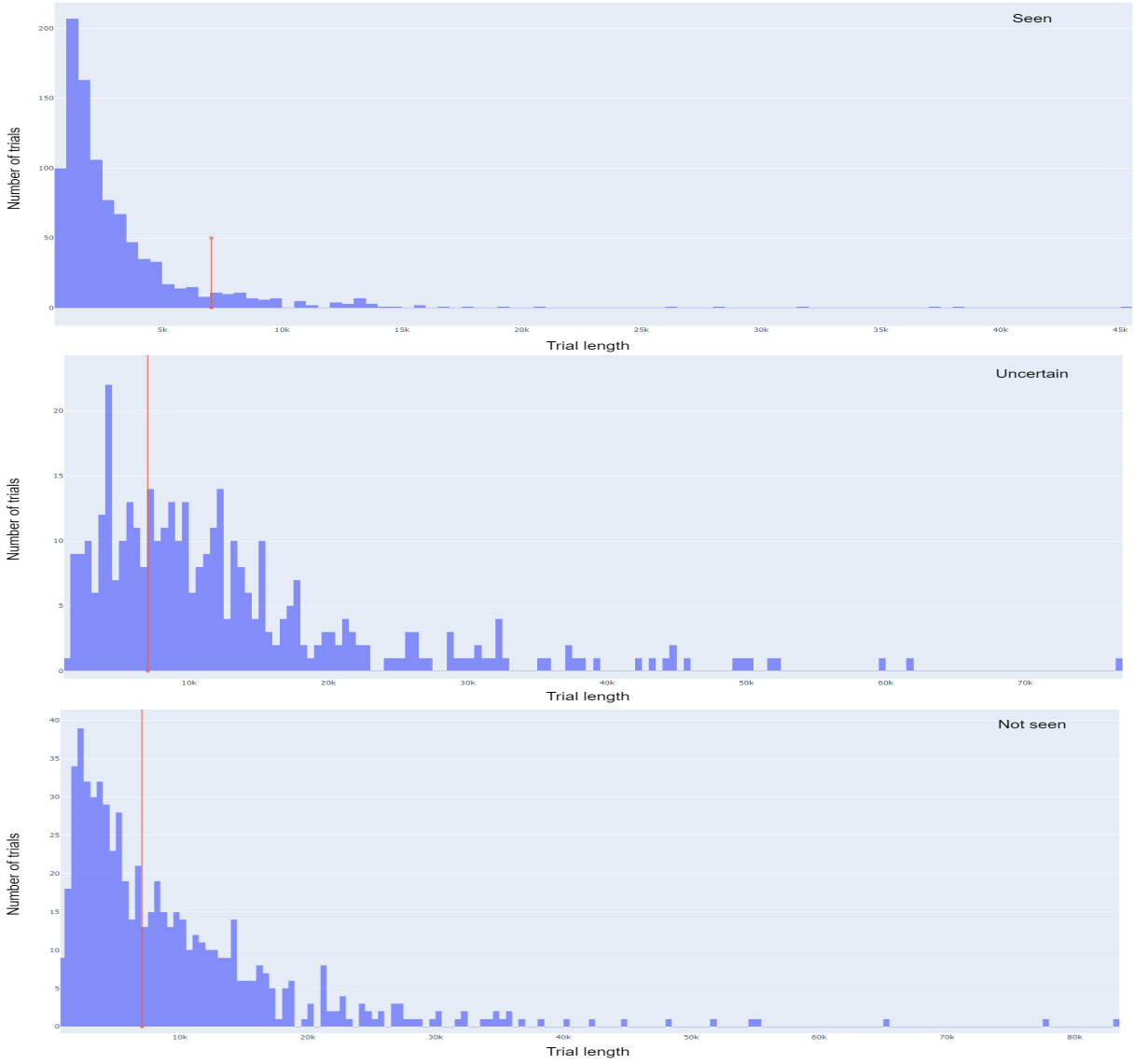


Figure 4.6: Trial histogram on seen (Top); On uncertain (Middle); On not seen (Bottom); The red vertical line represents the mean of the entire set.

The first option that comes to mind is to choose these parameters based on intuition and different properties of the dataset, which may even be physiological. However, this approach is not suitable for the task at hand because it does not ensure that a machine learning model will be able to identify the crucial properties to build the time snapshots that we are interested in. Therefore, a machine learning approach has been created which is capable of testing different segmentation parameters and choosing the best one amongst all of them.

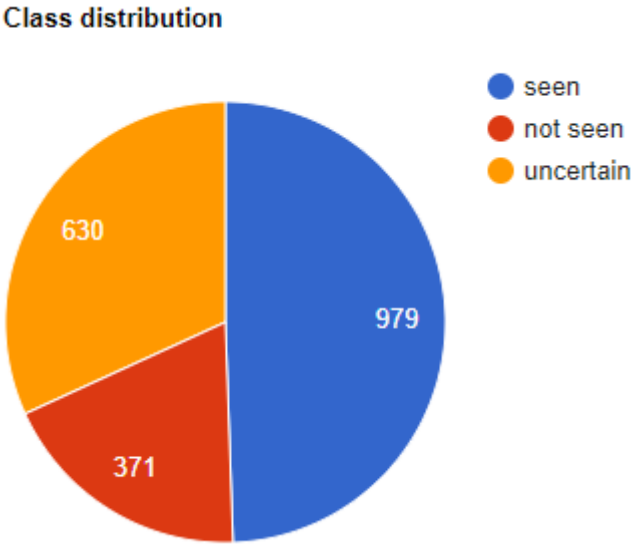


Figure 4.7: Piechart representing the number of trials per response.

4.4.1 DEC Architecture

The architecture of the machine learning model developed can be seen in figure 4.8. This model has been developed as a joint effort between my colleague [12] and I, with the blue components from the figure being the ones I have developed. In the following paragraphs, the overall architecture will be explained, with emphasis on the parts that I have worked on.

As specified previously, we are interested in being able to develop a model that is capable of identifying if certain trial segmentation parameters are suitable or not. Thus, our choice for the model was creating a classifier. However, a trivial classifier was not enough in our scenario. It is important to take into account that a trial has two important factors that describe it: the gravitational constant that influences the visibility of the depicted object and the response given by the subject. With this in mind, there are two sets of classes that could have been used. Therefore, considering the fact that a given trial has two possible classes it can work with, we have decided to actually build a *dual classifier* which is capable of simultaneously classifying a trial in these two distinct classes.

Therefore, the first component of the architecture is the *Dual Neural Network (DNN)* component, which is the core of the entire architecture. In this component, the input consists of two parts, the stimulus section of the trial and the response section of the trial, both after being transformed using the segmentation principle described in 4.4.2. Each of these inputs will be fed into its own Neural Network, consisting of the following:

- An input layer, which is the input fed to the Neural Network.
- A hidden layer, with variable size.

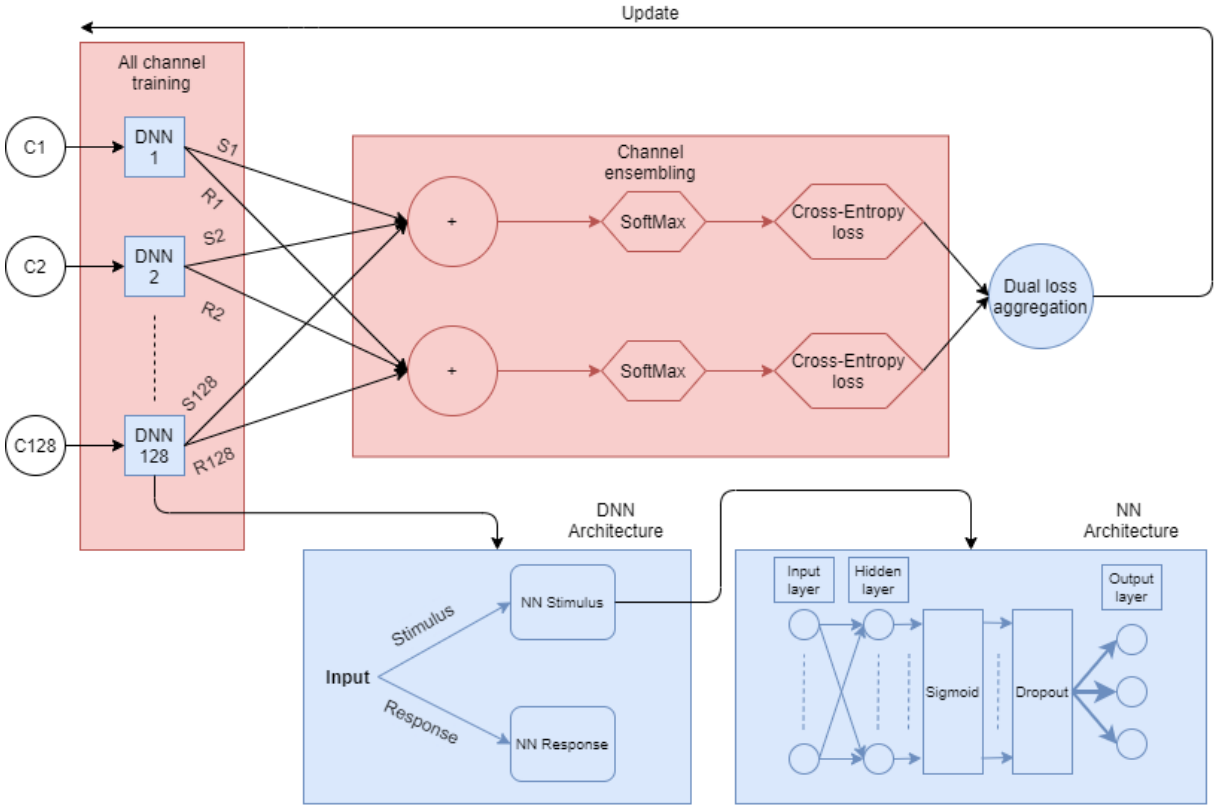


Figure 4.8: Dual Ensemble Classifier (DEC) architecture.

- A sigmoid layer, which ensures the non-linearity factor of the architecture.
- A dropout layer, which helps avoid the overfitting of the network.
- The output layer, having a number of neurons corresponding to the number of classes predicted.

Because there are two Neural Networks for each DNN cell operating on two different parts of a trial, the two DNN cells are also configured differently. More concretely, the output layer needs to be equal to the number of classes predicted, which is 3 for the response and 6 for the stimulus. Moreover, the size of the hidden layer has been chosen as a function of both the size of the input and the number of classes in order to ensure that the network is capable of learning the necessary properties of the input data. Thus, the size of the hidden layer is defined as:

$$HiddenLayerSize = \frac{2}{3}InputSize + OutputSize \tag{4.1}$$

The next step in the architecture is based on the ensemble principle, which states

that the performance of a model may be increased if there are multiple machine learning models contributing to the output result. Considering that we have 128 channels to work with, we have decided to apply the DNN cell individually for each channel. Afterwards, in order to combine the results, the *Channels ensembling* component has been developed where the so-called *ensembling vote* is performed. The output of this component is two individual losses, one for the classification performance for the response and one for the stimulus. As an intermediary result, it also has an output the predicted class for both of the class sets we are interested in, which will be further used in 4.6.

The last step of the architecture represents the *duality* component of the DEC model. Until this point, everything that has been done has been either on the response part, or on the stimulus part. As specified previously, we believe that, since a trial can be given two individual labels, these two labels should be somewhat correlated. Looking at the experiment from which the data is generated, it becomes obvious that the two classes are in fact correlated. Intuitively, one could argue that:

- For a low gravitational constant, the response should be not seen.
- For a low to medium gravitational constant, the response should be uncertain.
- For a medium to high gravitational constant, the response should be seen.

However, this is not necessarily true because, as it was discussed in the section 4.2, the behavior of each subject directly directly influences the response. Therefore, even if the two trial labels are correlated, the correlation is not quite obvious. With this in mind, we have decided to make a joint classification algorithm that is capable of optimizing simultaneously the two classifiers.

Basically, what we have tried doing is to actually make a class's DNN and Ensemble optimize based on the performance of not only itself, but also on the performance of the other class. Considering the fact that there exists a correlation between the two label classes, the joint optimization has been developed as the effect of the *Dual Loss aggregation*. More concretely, the loss of the DEC model is the aggregation of the losses of the two individual components, defined as:

$$L_{DEC} = L_r * L_s + \mid L_r - L_s \mid \tag{4.2}$$

The dual loss is defined as the sum of two components. The first part of the loss consists of the multiplication found in equation 4.2. This component is the one ensuring the dual optimization of the architecture. Mathematically, the derivative of the multiplication will include, with respect to a class's weights, the weights of the other class and vice-versa. However, a simple sum would not have sufficed because, in that case, the derivative would with respect to a model would have been 0 for the other model and then it would have been like the models were training individually. One could argue that a more complex aggregation could have been possible and he would be right, but, for what this model is interested in, we have found out that this simple operation is enough.

However, one problem arises with this type of aggregation. If only the multiplication is present, then there exists the possibility that only one of the two components of the DEC model gets optimized. Basically, what could happen is that the performance of one model remains constant whilst the performance of the second model increases. The overall loss would still decrease but the model would not learn anything in one of its components.

Therefore, in order to fix this issue, the second part of the loss was introduced in equation 4.2. Basically, what has been added is the absolute difference between the two losses. The role of this component is to actually make the DEC model not allow only one of the two components to get optimized. By introducing this absolute difference, the only solution for the DEC model to optimize is to not only decrease the loss of both the response and stimulus part, but also decrease it by the same order of magnitude.

Now, having the final loss aggregation described, the DEC model has been fully defined. Looking at its architecture in figure 4.8 and at the explanations provided in this section, the DEC model proves itself to be a viable option for identifying the necessary segmentation hyperparameters we are interested in by being capable of classifying a trial using two sets of classes, in an ensemble manner, with a joint loss aggregation able to exploit the correlation between the two possible sets of classes.

4.4.2 DEC Dataset

Figure 4.9: Trial Segmentation example.

Having the conceptual architecture defined for the DEC model, it is time to also define the way the dataset is created. Because of the nature of the architecture and the initial dataset, this proved to be no easy task.

First of all, for each trial, there are a total of 128 channels that need to be dealt with. Moreover, for each of these channels, there are 2 parts we are interested in: the beginning of the trial, right when the stimulus is presented, also called the *stimulus side*, and the end of the trial, right before the response is given, also called the *response side*.

However, these two parts need to be delimited somehow. As it was shown in the 4.2, the trials do not have the same length. Their distribution is quite chaotic, with

trials having lengths in different orders of magnitude. However, the architecture of the DEC model is static, which means that all of the examples need to be of the same size. Moreover, since we are interested in equally dividing both the response and the stimulus sides in windows, these two parts should also be of the same length. In order to solve this issue, we have made the decision of actually stripping all of the trials with respect to the length of the shortest trial, which is 627 milliseconds. Therefore, what we have done is that we have taken 627 millisecond after the stimulus and 627 milliseconds before the response, ignoring the rest of the trial. This process can be viewed in figure 4.9.

Having the two sides defined, we needed to extract the examples using the window hyperparameters, *window size* and *window offset*. In order to do this, we have created a sliding window of size *window size* which moves forward in time with an overlapping offset of *window offset* until it reaches the end of the side we are working with. Therefore, this operation was applied twice: once for the stimulus and once for the response, resulting in two sets of windows, one for each side. The final operation applied in the example creation was concatenating all the windows in one set with respect to the time, thus creating a windowed view of the trial with which the DEC model can work with.

After defining the operation based on which the examples are created, the trials are converted into this representation. It is important to note here that the trials having a gravitational constant of 0 were ignored because they do not posses any information whatsoever that would prove itself important in this part.

4.4.3 DEC Dataset distribution and class imbalance

In 4.4.2, the way the dataset is created was defined. However, there are two more things one needs to pay attention to when creating the dataset.

First of all, when creating the train, cross-validation and test dataset, one needs to make sure that the distribution of trials with respect to the subjects is balanced. As seen in 4.2, the behavior of each subject directly influences the outcome of the experiment. Therefore, one needs to pay attention of choosing the same number of trials to include in each train, cross-validation and test set from each subject, thus ensuring that the model will have enough data distribution in order to be able to generalize its observations over all the subject.

Furthermore, as seen in the 4.3.3, one more problem that needs to be dealt with is the fact that there is an imbalanced data distribution with respect to the response class. Because of this, the DEC model may suffer from the bias in the data. In order to solve this issue, the concept of *weighted classes* should be introduced which could ensure that the bias from the data would be removed by performing a *cost sensitive learning*. In order to do this, one weight should be associated to each class, such that:

$$T_{seen} * W_{seen} \approx T_{uncertain} * W_{uncertain} \approx T_{notSeen} * W_{notSeen} \tag{4.3}$$

Therefore, based on equation 4.3, the following weights have been chosen for each

of the response classes, with respect to the number of trials they have:

- Seen weight = 1.0
- Uncertain weight = 2.5
- Not seen weight = 1.5

4.5 DEC Hyperparameter tunning

After defining the DEC architecture, the next step is identifying which are the best windowing parameters. For this, we have chosen to try the following values:

- Window size = 100, 125, 150, 175, 200, 225, 250, 275, 300
- Window offset = 50, 75, 100

The windowing parameters will be the cartesian product between these two sets. Therefore, there will be a total of 27 different sets of parameters to test. After training the model with the specified parameters, we need to be able to make a comparison between them. Therefore, a set of classification metrics must also be defined that can aid the process of choosing the optimal set of hyperparameters.

4.5.1 Classification metrics

Because we are dealing with a classification problem, it is rather obvious that our choice of classification metrics will involve the concept of confusion matrix. In a binary classification problem, the confusion matrix is defined as a matrix of 2 rows and 2 columns where each of the possible outcomes of a supervised classifier are tested. The exact definition of a confusion matrix can be seen in table 4.2. This way, the confusion matrix gives a better insight than the novel accuracy, which is actually prone to problems like class imbalance.

More concretely, the confusion matrix defines 4 initial values. If the prediction was correct, depending on the class predicted, it defines the values *True Positive* (for the positive class) and *True Negative* (for the negative class). However, if the prediction is wrong, then it defines the values *False Negative* (if the class was positive and the prediction was negative) or *False Positive* (if the class was negative but the prediction was positive).

Having these 4 values defined, the confusion matrix proves to be a worthy classification measurement because of one critical function it can compute: the $f1 - score$. This function is defined as the harmonic mean between the *precision* and recall, reaching a value of 1 if the classification was perfect. Moreover, it is not prone to class imbalance

Table 4.2: Confusion matrix.

		Predicted class: Positive	Predicted class: Negative
Actual Class	Positive	*True Positive*	*False Negative*
	Negative	*False Positive*	*True Negative*

like accuracy is, therefore making it the best candidate for our problem. The function is defined in equation 4.4.

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{4.4}$$

Now, we need to define what precision and recall is. Precision is a measure of the proportion of how many positive predicted examples were actually positive. It's definition is present in equation 4.5. On the other hand, recall is defined as how many from the positive examples were correctly identified. It's definition is present in equation 4.6.

$$Precision = \frac{TP}{TP + FP} \tag{4.5}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.6}$$

However, this definition applies only in the binary classification situation. In our case, we are actually dealing a multiclass classification problem, since the gravitational constant has 6 possible values and the response has 3 possible values. Therefore, the confusion matrix definition needs to suffer a small change.

For a multiclass classification problem with C possible classes, the confusion matrix is defined as a square matrix of C rows and C columns, where the rows represent the actual classes and the columns represent the predicted classes. Therefore, for each position in the matrix (i, j) with $i \neq j$, the value present in the confusion matrix represents the amount of times class i has been mispredicted as class j. Moreover, for each position in the matrix

(i, j) with $i = j$, the value present in the matrix represents how many times class i has been correctly predicted.

Having the multiclass confusion matrix defined, the definition of the scores (precision, recall, f1-score) follows the same reasoning, providing a set of scores for each class. Since we are interested in an overview of the performance of the DEC model over all the possible classes, the final classification score will be defined as the average f1-score over all the possible classes. This can be seen in equation 4.7.

$$C_{score} = \frac{\sum_{i=1}^{C} F_1(i)}{C} \tag{4.7}$$
$$where\ C\ is\ the\ number\ of\ classes.$$

As one last observation, one must note that this score is to be applied on only one of the 2 classifiers integrated in the DEC model, namely the response classifier and the stimulus classifier. The actual score output will represent two values, an average f1-score for each of the two sets of labels the DEC model trains on.

4.5.2 Initial tuning and filtering

After defining the performance metric we are going to use, it is time to start identifying which set of window parameters is the optimal one. For doing this, we have split the work into two parts as it can be seen in figure 4.10. The first part has been done by me and the second part has been done my colleague [12].

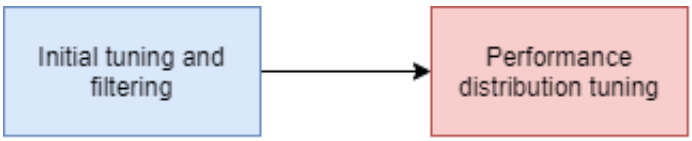


Figure 4.10: DEC Tuning.

In the first part of the tuning, we needed a way to be able to choose a smaller subset out of the 27 possible window parameters because of hardware and time limitations. Therefore, this part of the solution was developed which has the goal in mind to perform a shallow tuning that is capable of filtering out the parameters that do not seem to be optimal.

More concretely, the first step is to actually train a DEC model for each one of the 27 window parameters possible, ignoring the fact that the result is prone to change because of the random component of the training, such as getting stuck in local minimas. Afterwards, the model's performance is computed against the test set, obtaining two values for each DEC model, one representing the value of the average f1-score on the response part, and one representing the value of the average f1-score on the stimulus part.

After having these two values, what we actually do is to filter out the models that had a low performance. The premise is that, if a model behaved badly in this context, it

will behave badly even for a more in-depth analysis. Moreover, we also want to keep the models that had a good performance on both classification problems, not only on one of them. In order to filter them out, the average performance score was computed for each of the two parts, resulting in two threshold values that represent the lower-bounds based on which a model would be considered to have the potential of being optimal.

Having the two values defined, the filtering happens only if the condition present in equation 4.8 holds true. More specifically, if either of the two parts of the model behaved worse than the average performance of all the models on that classification task, then the parameters are completely discarded from the analysis.

$$R_M < R_{Avg} \; and \; R_S < R_{Avg} \tag{4.8}$$

After this filtering has taken place, a subset of the initial hyperparameters has been obtained which have a performance score above average on each of the two classification problems. Because of hardware limitations, we needed to take this filtering one step further in order to decrease this subset even more. In order to do this, three sets of hyperparameters that will be further studied have been chosen from this subset as follows:

- the hyperparameters which yield the best performance on the stimulus classification.
- the hyperparameters which yield the best performance on the response classification.
- the hyperparameters which yield almost the best performance on both stimulus and classification.

In this context, one might wonder why was the initial filtering needed (using the condition from equation 4.8). The reason behind this choice was the fact that, for example, for the first item in the list, the model which had the best performance on the stimulus classification on the unfiltered set might have a really low performance on the response classification. However, if the three sets of hyperparameters are chosen from the filtered set, this problem is actually removed since those types of outliers are completely discarded from the hyperparameters set.

At the end of this part, there will be only a subset of 3 sets of hyper-parameters that need to be checked out to figure out which are the optimal ones. For doing this, my colleague [12] has developed a module that is able to compare the DEC models' performances by also taking into account the random component of the training that might occur (such as different local minimas). However, his module will be applied only on this subset of parameters which are the result of the initial filtering.

4.6 DEC Model analysis

After identifying the best hyperparameters that maximize the performance of the DEC model, the next step we have chosen to do is to actually analyse the model with

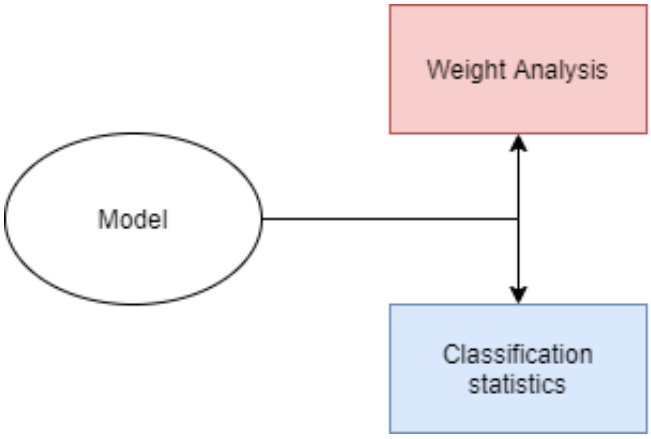


Figure 4.11: Model analysis components.

respect to what it has learnt, how well it performs and to figure out if there are certain external limitations that stop it from having an even better performance.

In this context, two modules were developed which can be seen in figure 4.11. The first one, namely *Weight Analysis*, has been developed by my colleague [12]. The purpose of this module is to actually analyse the interpretability part of the model. More concretely, the goal of this module is to actually what the model was capable of learning from the input dataset as an overall observation over multiple runs on the optimal set of hyperparameters.

The second module, namely Classification Statistics, has been developed by myself with the purpose of actually identifying what were the limitations that stopped the DEC model from gaining an even better classification performance than the one it currently has. Moreover, the analysis is performed only on one single model, namely the model which had the best classification performance. There was also the possibility of making an analysis over the overall performance of multiple models trained on the same set of optimal hyperparameters, but this option was quickly discarded since the analysis can yield more meaningful results on an individual model with the best performance.

In this context, what I actually wanted to analyse is to figure out if there were some properties of the input dataset used for training the DEC model that stopped it from being capable of having an even better performance. More concretely, I wanted to find out if there are certain patterns in the way that the model is able to classify examples that are correlated with patterns in the input dataset.

Based on the way the model works, the working hypothesis was to actually analyse the classification distribution with respect to the trial length. As noted in 4.4.2, because of the architecture of the DEC model, a trial segmentation was required, where a component of variable length has been removed from the trial. Since the length of the part removed is a function of the trial length, loss of the relevent information from long trials may occur.

Following this, the first step was to actually label each trial from the input dataset as *classified* of *misclassified*. Afterwards, two histograms were created:

- In the first histogram, with respect to the trial length, the number of examples classified and misclassified have been plotted one against the other.

- In the second histogram, with respect to the trial length, what was actually plotted was the percentage of misclassified trials for a certain bin, using the formula from equation 4.9.

$$\%_{misclassified} = \frac{misclassified}{misclassified + classified} \tag{4.9}$$

One might wonder which is the reason for which there are two histograms instead of just the second one, from which an overall trend might be able to be seen by itself, with no need of the first histogram. The reason behind having two histograms is closely related to the distribution of the trials lengths. As seen in 4.3, after the mark of a few tens of thousands of milliseconds, there are a lot of outliers that appear that, in the case of a histogram, will be placed in a small number (or even independently) in a bin. Because of this, the second histogram is actually biased based on the distribution of the trials lengths and, therefore, might be misleading. However, if one were to use the first histogram as a complement of the second one, any observations that might be biased are discarded and unbiased observations might be discovered.

4.7 Recurrent Graph WaveNet

After identifying the windowing hyperparameters using the DEC model, namely the window size and the window offset, it is time to actually extract the meaningful information we are interested from this windows. More concretely, as specified in the 2.1, we are actually interested in being able to extract a matrix of correlations between the channels at a window level inside the trial.

4.7.1 Graph Wavenet core

The *Graph WaveNet* model represents the core of this solution. As described in the [9], this model is capable of extracting an adaptable learnt matrix of correlations from a multivariate timeseries data, taking into consideration the evolution of multiple signals with respect to both the time and the features that define the signals.

The novelty of this approach actually comes from its unique architecture, where two deep learning approaches have been combined in order to allow the timeseries modelling of multiple signals correlated in a graphwise manner, namely the *WaveNet* model [16] and the [5]. More concretely, the WavNnet model has been used as an enhancement in the DCRNN model, replacing its timeseries modelling component and, therefore, creating a model that has the rank of state-of-the-art on the traffic prediction problem.

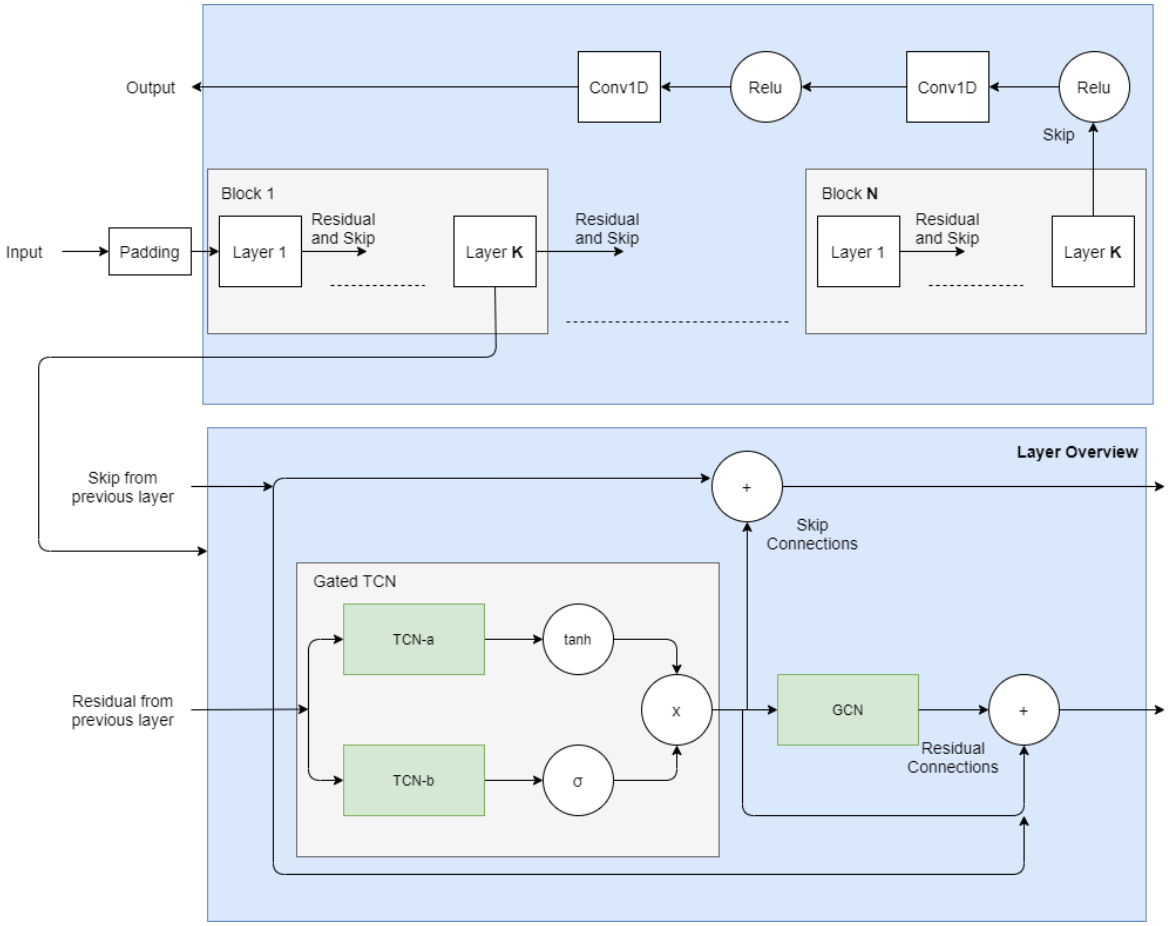


Figure 4.12: GW Architecture.

The overall architecture of the Graph WaveNet model can be seen in figure 4.12. The architecture consists of multiple blocks, each with the same number of layers. The layers inside a block are connected to one another in a sequential manner, where the output of one layer is the input of the other. Moreover, this sequential mechanism can also be found at the block level, where the output of the last layer of a block is actually the input of the first layer of the next block. However, for the first block, the input of the first layer is actually the timeseries example it models and, for the last block, the output of the last layer is actually used to make the prediction using multiple one-dimensional convolutions.

Each layer consists of 4 components, each with its own purpose. The first part of the layer consists of the WaveNet principle, implement through a *Gated Temporal Convolutional Network*, which has the purpose of modelling the time component individually on each signal. This component consists of two TCN modules, each applied on the same input. The TCN module actually applies a dilated causal convolution operation over the input, where the dilation factor is given by the index of the layer. More concretely, for

a layer of index **k** inside a block (the counting starts from 0 for each block), the dilation factor is equal to 2^k.

More concretely, the dilated causal convolution is defined as a convolution where, based on the dilation factor, a skipping distance between the convoluted elements is defined. The formula for a 1D dilated causal convolution is defined in equation 4.10 [9].

$$x \star f(t) = \sum_{s=0}^{K-1} f(s)\mathbf{x}(t - d \times s) \tag{4.10}$$

The power of the TCN module actually comes from the layered part of the architecture where, by stacking multiple dilated causal convolution one on top of the other, with an increasing dilation factor in powers of 2, the receptive field grows exponentially, making the model capable of modelling longer signals with a few number of layers, hence improving the overall efficiency of the architecture [9]. The stacking effect of multiple layers can be seen in figure 4.13.

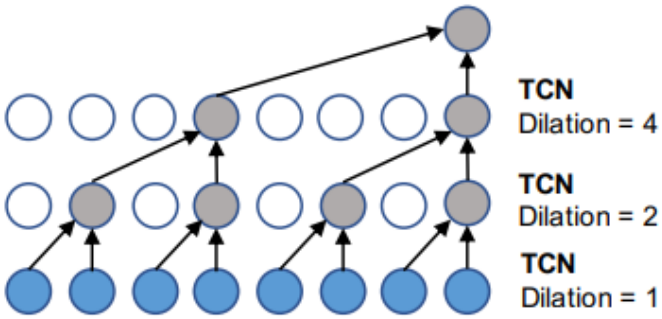


Figure 4.13: Exponential receptive field [9].

However, the TCN is actually enhanced using the concept of the Gate, which is actually an implementation of a thresholding mechanism used by a network to filter out the amount of information which is to be passed over a gate. More concretely, in the case of Graph WaveNet, two TCN models are defined with their corresponding results being passed through two activation functions, *hyperbolic tangent* and *sigmoid*. The gate concept is applied by multiplicating in an element-wise manner these two results, with hyperbolic tangent acting as an activation function and the sigmoid acting as the thresholding mechanism, specifying for each element how much information should be passed through the gate. Therefore, by actually applying the gate mechanism over the TCN concept, the model is able to learn more complex temporal dependencies between elements [9].

Moreover, because of dilated causal convolution happening in a layer, the size of an input is actually decreased with respect to the time component. More concretely, each layer with index **k** inside a block decreases the temporal length with a value of 2^k. Considering that this principle is applied recursively on each block, the receptive field of the architecture is defined as described in formula 4.11. This fact determines the size of the input temporal length because the Gated TCN is the only component of the Graph Wavenet which modifies the size of the input.

$$size_{ReceptiveField} = (2^{\ number\ layers} - 1) * number\ blocks \tag{4.11}$$

The second component of the Graph WaveNet architecture is actually the *Graph Convolutional Network*. As for the Gated TCN component, this module is present in each layer. The power of this module comes from the fact that is able to model the dependencies between multiple signals and their corresponding features at each timestep. For a set of **N** signals, it defines two types of adjacency matrices:

- Predefined adjacency matrices of size $N \times N$
  In some cases, certain matrices of adjacency might be known beforehand when modelling a graphlike structure. For this, the Graph Wavenet module takes into account this concept and defines the forward and backward transition matrix for a directed graph as described in equation 4.12 [9].

$$P_f = \frac{A}{rowsum(A)}; P_b = \frac{A^T}{rowsum(A^T)} \tag{4.12}$$

- Self-adaptive Adjacency Matrix of size $N \times N$
  In addition to the known adjacency matrices, the model is also capable of learning the hidden spatial dependencies in a self-supervised manner, resulting in a transition matrix of a hidden process. The formula for this matrix is defined in equation 4.13, where E_1 and E_2 are two node embedding dictionaries of size $N \times c$, where **c** is the depth of the embedding [9].

$$\widetilde{A}_{adp} = SoftMax(ReLU(E_1 E_2^T)) \tag{4.13}$$

Having the two types of matrices defined, the graph convolutional layer is defined in equation 4.14, where **X** is the input matrix of size $N \times D$, with **D** denoting the number of features. As it can be seen from the formula, the goal of this component is to model the graph with respect to its features at different orders of neighbourhood, which is possible through the fact that the adjacency matrices used are raised to multiple powers [9].

$$Z = \sum_{k=0}^{K} P_f^k X W_{k1} + P_b^k X W_{k2} + \widetilde{A}_{adp}^k X W_{k3} \tag{4.14}$$

After defining the two main components of the Graph WaveNet, the next component of a layer we need to focus on is the residual connections. Because of the architecture of the model, residual connections are required in order to be able to actually improve the performance of the model and how it learns with respect to its depth. Therefore, residual connections are introduced that allow the gradients to *flow* through a network directly, without necessarily passing through a non-linear activation function. More concretely, in the case of the Graph WaveNet, the residual connection is defined as the sum of the input of the layer, output of the Gated TCN and output of the GCN, with the second connection being introduced by [17]. The result of the residual connections is fed to the next layer.

The last component of a layer is the skip connections. Similarly to the residual connections, the purpose of the skip connections is to allow deep networks to train faster, discarding problems such as vanishing gradient. For the Graph WaveNet, they have been implemented as the sum between the skip result of the previous layer, the input of the current layer and the result of the Gated TCN. The result of the skip connections is also fed to the next layer.

However, for the last layer, the result of the skip connections is actually used to compute the output. Because of the architecture of the Graph WaveNet, the model is capable of predicting single-featured multi-step sequences artificially, by adjusting the architecture such that the result of the last skip layer will have a temporal length of 1. In order to build the output of the desired length, several 1D convolutions are applied in order to create the desired output dimension [9].

4.7.2 RGW Approach

As noted in 2.1, the goal of this thesis is the ability to extract at a window level, from a trial, the dependencies between the 128 channels, which can be translated into a correlation between brain regions. For this, we have firstly defined the DEC model which identified the necessary windowing parameters for a trial, and the GW model, which is able to extract from a timeseries dataset the meaningful dependencies we are interested in a self-supervised manner, namely the self-adaptive adjacency matrix.

Therefore, in order to combine these two concepts, we have defined the following pipelined architecture called *Recurrent Graph WaveNet*, which is able to create a series of snapshots in time of the brain's activity using the trials from a subject.

More concretely, for each subject, we follow the evolution of trials with respect to the time. For each trial, we divide it into windows using the same windowing concept as it was displayed in figure 4.9. However, it is important to note here that we have actually increased the length of each of the two segments to **1 second** as a result of the observation from 6.2.2.

After defining the windows, Graph WaveNet is applied on each of them. More specifically, the model is trained to solve a single-step timeseries prediction problem on the corresponding window, learning the hidden dependencies between channels as it improves the prediction performance. The results we are interested from each window is actually

the learnt matrix, which represents a snapshot in time of the correlation between the brain regions that have been recorded during the experiment.

Moreover, we have also made use of the ability of the Graph WaveNet model to include known adjacency matrices in order to boost its performance. More concretely, we have made use of both **Functional Networks** and **Previous window learnt matrix**.

As specified in [2] [3], Functional Networks represent a statistical correlation between the brain regions over the entire length of the trial. Therefore, they are capable of capturing the most significant correlations between the channels and represent a basis on which our time snapshots could be built upon.

Furthermore, considering the fact that the brain has a built-in mechanism of using previous knowledge in order to improve its efficiency in solving tasks, we have decided to actually try duplicate this behavior in this context. More concretely, since the brain's state of dependencies evolves in time, it might be helpful for the model to have as a basis what the model before it, with respect to time, has learnt. In doing so, a sequential pipeline of self-learning snapshots is defined, where each snapshot is built not only on the observations from the current window, but also on the entire set of dependencies that have evolved on the previous windows.

This conceptual architecture of the RGW model can be seen in figure 4.14. It is important to note that the structural decisions that we have taken regarding the way Graph Wavenet is applied at a window level and what adjacency matrices to feed to the model have been thoroughly tested and will be displayed in the section of 4.8.

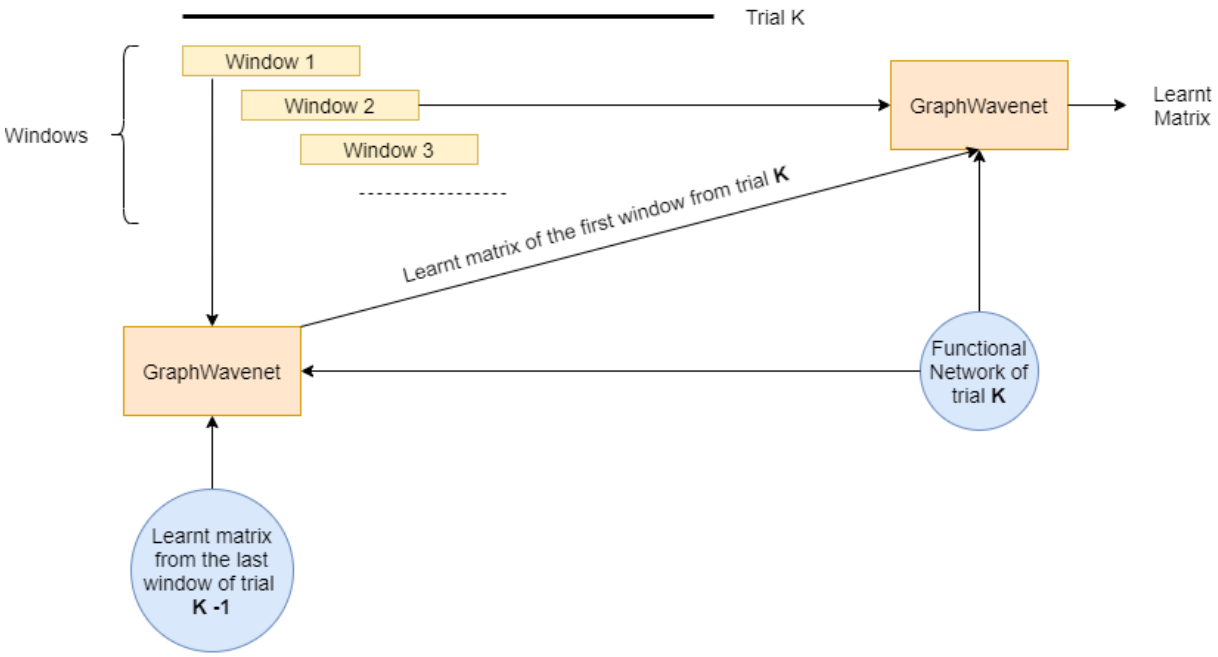


Figure 4.14: RGW Conceptual Architecture

4.7.3 RGW Dataset

As it can be seen from 4.7.2, there is not only one dataset for the entire RGW, but rather multiple smaller datasets, one for each of the Graph WaveNet models used. Moreover, the RGW is only applied on a subject. Therefore, considering that the Graph WaveNet is applied at window level, the first step is to actually take all the trials from one subject and create the corresponding windows using the windowing concept having as parameters the one discovered by DEC, namely the window size and the window offset. The windowing process is the same one as the one from figure 4.9, with the obervation that we have increased the length of the two sides from 627 milliseconds to **1 second**. This decision has been made with the observation coming from the performance of the DEC, where longer trials had a likely hood of being misclassified because not enough information was present in the input example. This observation is detailed in section 6.2.2.

After defining the windows, it is time to actually create the dataset for the Graph Wavenet model applied on the corresponding window. More concretely, an input example for the GW has the shape of **(BS, NF, NN, TL)**, where:

- **BS** is the batch size
- **NF** is the number of features per channel
- **NN** is the number of nodes
- **TL** is the temporal length

In our case, the BS parameter is a hyperparameter and does not depend on the data. The NF parameter, however, depends on our data and has the value of 1. This is because, since the recording happens with an EEG headset which, at each millisecond, reads the voltage flunctuations from the brain, the output of the headset is one value per channel. Moreover, the number of nodes is also dependent on the data. More concretely, the number of nodes represent the number of individual signals the models tries to predict that, in our case, is equal to the number of channels from the EEG headset, which is 128.

The last parameter, **TL** is closely related to how the examples are created for the Graph Wavenet model. More concretely, since it's a timeseries prediction problem, the examples are created following the same windowing concept as the one it was used until now, but with different parameters. This process can be seen in figure 4.15.

More concretely, there are two main differences regarding how we create the examples now compared to how we have created them for the DEC model. First of all, as it can be seen romr the figure, we actually define one more set of values called the **output**. This is needed because we are dealing with a timeseries prediction problem.

The second difference regarding the previous windowing approach is the stride. Since the purpose of the Graph Wavenet model is the ability of capturing and modelling different timeseries properties from the **input**, we need as many examples as possible.

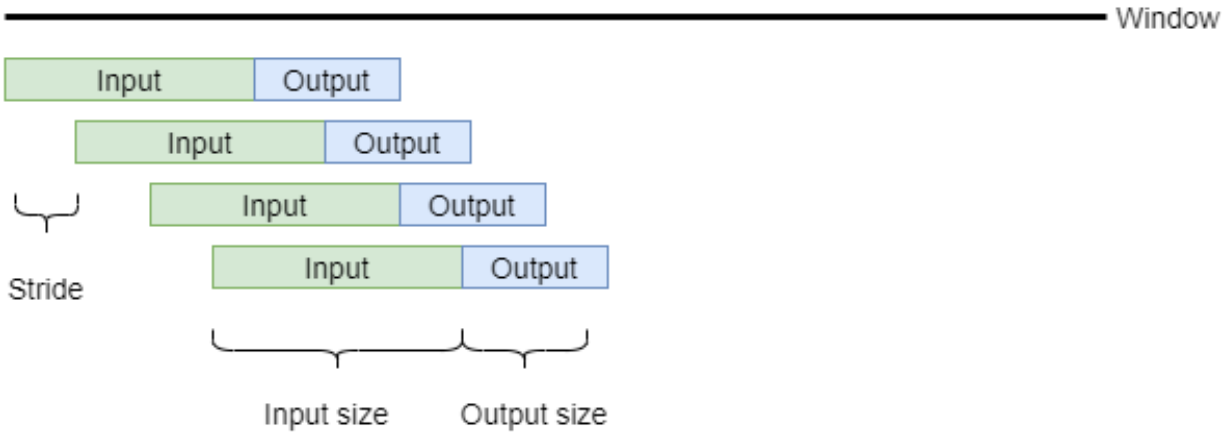


Figure 4.15: GW Example creation

The maximum number of examples we can create is when we set the value of the **stride** parameter to 1.

Now, all that is left to define is the value of the **input size** (which is equal to the parameter **TL**) and **output size**. However, it is important to note here that the decision for these parameters has been made with respect to the hardware limitations that the computers we have used have imposed.

More concretely, the **input size** parameter proves to be hardest one to choose because, as described in 4.7.1, the temporal length of the input example denotes the architecture. Moreover, since we are trying to capture as much information as possible, it logically follows that this value should be as large as it can be. Looking at the original paper, they have chosen an input size of 12. However, their sampling rate for the signal creation was actually 5 *minutes* which actually proves that their input observation is capable of covering an entire hour. In our case, we have chosen the value of 30 milliseconds because of hardware limitations and computational requirements. It is the largest value that we were able to train a model on in an acceptable time frame. A value larger than this would have made it so the model was not capable of training on the hardware we had available.

Furthermore, the same type of decision was made for the **output size**. However, since we wanted to have the ability of including a large amount of information in the input example, we were faced with the fact that we needed to decrease the size of the prediction in order to accommodate to the hardware requirements. Therefore, we have decided to make the Graph WaveNet model actually predict in a single step manner, having an output size of value 1, which still proves to be capable of capturing the temporal dependencies required in making an accurate prediction.

However, besides this example, the Graph WaveNet model also receives as an input parameter two adjacency matrices (which are independent with respect to the examples).

More concretely, for each trial, we have a **Functional Network** which will be used by all the models inside that trial and, for each window, we have the **self-adaptive learnt adjacency matrix** discovered by the previous window. These adjacency matrices are both directed and weighted and, in order to be used by the model, are transformed corresponding to the equation 4.12.

4.8 RGW Hyperparameter tuning and training

There are two sets of hyperparameters that need to be identified in order to be able to use the RGW model. The first set of hyperparameters deals with the Graph WaveNet component by controlling its structure by defining the number of **blocks** and **layers** the architecture has. The second set of hyperparameters deals with the RGW approach by controlling which adjancency matrices are to be used from the options we have, namely **Functional Networks** and the **self-adaptive learnt matrix** from the previous window.

4.8.1 Cross-validation set definition

The first step in finetuning the hyperparameters is to actually identify what the cross-validation set looks like. As in any machine learning problem, when trying to discover the optimal hyperparameters, the model needs to be trained on the training set and then validated on the cross-validation set in order to see if the model has actually been capable of learning patterns that can be applied on previously unseen data.

However, since we are dealing with a timeseries prediction problem, the construction of this set is not quite straightforward. As described in [6], the problem we face is that the traditional way of choosing examples randomly from an example set is not suitable in this context because:

- Examples from the cross-validation set may contain overlapping information with examples from the train set due to the windowing process which extracted the examples.
- Examples from the cross-validation set might have taken place during the training window.
- Examples from the cross-validation set capture only one process that occurs that might have not happened in any other place in the train set.

Because of these three problems, we have chosen to build our cross-validation set using the **rolling origin forward validation** approach [6]. The process can be seen in figure 4.16. The idea behind this approach is that there are multiple training periods for a model, where, incrementally, the size of the input set increases with a constant factor and, respectively, the size of the cross-validation set decreases with the same factor.

Figure 4.16: roFV example [6].

In our concrete use case, we have decided to start with a train set of size 40% of the initial dataset, with an incremental factor of 20%, resulting in three independent runs based on which the hyperparameters are chosen. The performance of a model trained with this approach is actually the **average** score of the performance over each cross-validation set defined.

Moreover, it is important to note that, in order to completely separate the train and cross-validation examples one from the other, the windowing process has been performed only after the two sets were defined, thus eliminating the examples from the cross-validation set that might have components from the train set.

4.8.2 Performance metrics

In order to be able to choose between different sets of hyperparameters, we first need to define the performance metrics we will use in comparing them. For this task, we have made use of two functions: **Mean Absolute Error (MAE)** and **Mean Absolute Scaled Error (MASE)**.

The **MAE** function is also the loss function used during the model training. For the Graph Wavenet problem, where there are multiple signals and multiple timesteps, its definition can be seen in equation 4.15, where $\hat{X}$ represents the prediction of the model.

$$MAE = \frac{1}{TN} \sum_{i=1}^{i=T} \sum_{j=1}^{j=N} \mid \hat{X}_i^t - X_i^t \mid \tag{4.15}$$

However, there are certain problems with the **MAE** function. More concretely, in our case, as specified in [18], the problem that might arise is the fact that this function is not actually capable of identifying by itself if the model actually learnt anything useful. Therefore, the result of the function might be misleading with respect to the data.

In order to overcome this issue, for the analysis of a model, we have also made use of the **MASE** function. The idea behind this function is to actually test if the prediction performance of a model with respect to the **MAE** value is better than another benchmark prediction approach, such as the naive forecast, which is defined as building the prediction of a single-step prediction model using the last value from the input example.

In order to do this, the **MASE** function computes a scaled error independent of the scale of the data, having the output value as an indicator of whether the prediction is

better or worse than the prediction of a naive forecast approach [18]. More concretely

- For a value smaller than 1, the prediction is better than the one of the naive forecast.
- For a value equal to 1, the prediction is the same as for the naive forecast approach.
- For a value greater than 1, the prediction is worse than the one of the naive forecast.

The definition of the **MASE** function can be seen in equation 4.16 [18]. As it can be seen from the definition, the value of the function is the average over all the predictions made by the model, hence having a singular value per model.

$$\begin{gathered} MASE = mean(|q_t|) \\ where\ q_t = \frac{MAE_t}{\frac{1}{n-1}\sum_{i=2}^{n} \mid Y_i - Y_{i-1} \|} \end{gathered} \tag{4.16}$$

Having the two functions defined, the performance of the model will actually be established using these two values. Since there are multiple cross-validation sets, the performance of a set of hyperparameters is defined as two values: the average **MAE** score and the average **MASE** score over all the independent runs.

Moreover, using the performance metrics on the cross-validation set, one additional feature has been used, namely the **early stopping** mechanism. This mechanism has been deployed in order to have the ability of forcing the model to not overfit. More concretely, if the value of the performance of the model on the cross-validation set does not improve for a certain number of epochs, then the training stops and the model with the best value on the cross-validation set is kept in order to be compared with the other models.

4.8.3 RGW final training

After identifying the optimal hyperparameters for the problem, the last step that needs to be done is to actually build the RGW model with respect to the hyperparameters. For doing this, we have chosen to train the model on only one subject because of time constraints. Moreover, we have ignored the first block of stimulus with a gravitational constant of value 0 because we believe that the corresponding block does not bring any value to the generation whatsoever.

In order to actually train the RGW model over the trials of a subject, two decisions had to be made. First of all, we have taken the decision to include the entire window in the training dataset and to not have any cross-validation whatsoever. This decision has been made in order for the RGW model to actually be able to capture as much information as possible from a window.

The next decision that had to be made was when to set the stopping criterion of the training. Until this point, the stopping mechanism of the training was the implementation of the **early stopping** mechanism. However, in this case, early stopping is not possible

because there is no cross-validation set on which this decision could be made. Therefore, what we have actually done was to analyse all the runs done for the hyper-parameter tuning and identify the most likely epoch at which the model would stop learning, which, in our case, proved to be 45.

4.9 Graph Visualization

After creating the graphs, the next step is actually to try and visualize them. However, this proves to be not an easy task because of the amount of nodes the graphs have. More concretely, the number of nodes in a generated graph is actually equal to the number of channels on the EEG headset, hence it's equal to 128. Moreover, because of the nature of the Graph WaveNet model, the graphs are not only weighted and directed, but are also complete. Therefore, in the following subsections, the process of normalizing and sparsifying the graphs is presented, together with the aggregation process of multiple channels into brain regions.

4.9.1 Graph normalization and sparsification

The graphs generated by the Graph WaveNet model are weighted, directed and complete graphs. In order to be able to visualize them easily, we have made the decision of actually removing the **Softmax** component from the equation 4.13. Therefore, the range of the values of the weights of a graph are only lower bounded because of the **ReLU** function which sets negative values to 0, but are not upper bounded.

Because of this, we have decided to normalize the weight matrix of a graph in order to bring all the values in the matrix in the interval $[0, 1]$ whilst also keeping the same magnitude of difference between them. The normalization formula for a set of values is defined in equation 4.17. In our case, since the set of values is actually a matrix, the minimum and maximum of the matrix are computed based on all the values inside the matrix, not row wise or column wise.

$$x^i_{normalized} = \frac{x^i - min(\mathbf{X})}{max(\mathbf{X}) - min(\mathbf{X})} \quad where\ x^i \in \mathbf{X} \tag{4.17}$$

After normalizing the graph's weight matrix, the next step is to actually sparsify it. This operation is needed because we are only interested in the most meaningful connections between nodes. Therefore, in order to sparsify them, we have made the decision of keeping only the top **5%** most strong connections, resulting in a sparsified graph which has 819 connections.

4.9.2 Graph aggregation

Considering the sheer amount of nodes a graph has, namely 128, it goes without saying that displaying so many nodes on a picture is not a feasible thing to do, mostly because it will prove almost impossible to actually be capable of understanding what connections are relevant and between what nodes. After consulting with the researchers at the TINS institute, we have made the decision of aggregating the graph with respect to the brain regions from where the channels have sampled the EEG information. The defined brain regions and the corresponding channels which lie inside them can be seen in figure 4.17.

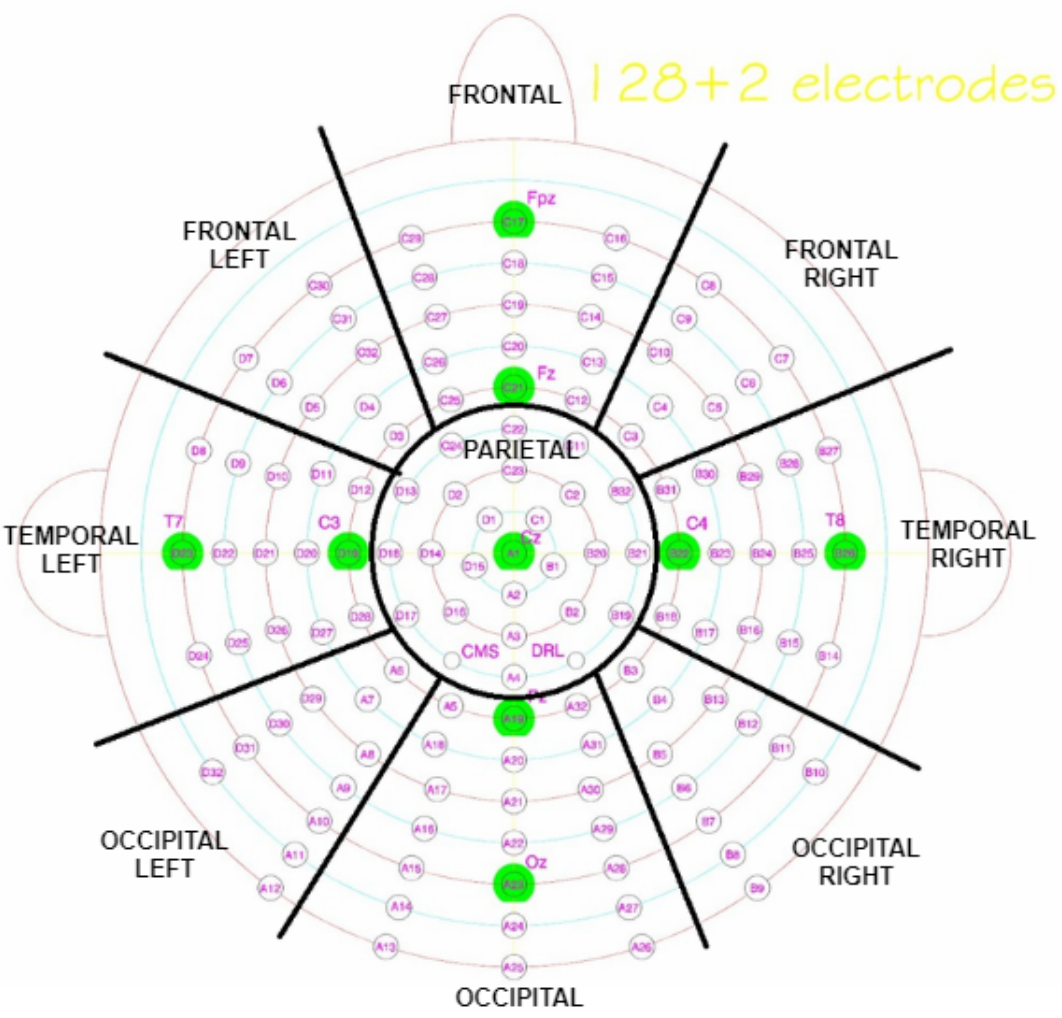


Figure 4.17: Channel aggregation with respect to the brain regions.

As it can be observed from the figure, 9 regions were defined with respect to the brain's main lobes. However, before aggregating the channels in the corresponding brain

regions, we first need to define how are the edges aggregated. There are 2 types of edges that need to be taken into consideration:

- Considering the fact that the graph is directed, certain edges will start from one brain region and go into another. Because of this, we have made the decision of actually keeping the aggregated graph as a directed one. Therefore, between any two regions, two types of edges are defined. More concretely, there is the outgoing edge which represent the sum of all the edges going out from the first brain region and into the second one, and there is an incoming edge representing the sum of all the edges going out from the second brain region and coming into the first one.

- The second type of edges are the edges that actually start and end in the same brain region. As it can be seen from the figure, there are multiple channels in each brain region, channels that might be connected. Because of this, we have made the decision of having a graph that has not only edge weights, but also node weights. The node weight of a brain region is defined as the sum of all the edges that start and end in the same brain region and it can be viewed as a self loop on the graph.

After performing the aggregation, the result is a directed graph, consisting of 9 nodes, one for each brain region defined, having not only weights on the edges of the graph but also weights on the nodes. Because of the aggregation mechanism being the **sum** operation, one final normalization needs to be applied in order to bring all the values in the interval $[0, 1]$. The formula defined in equation 4.17 is used and, because the node weights are defined as self loops, they will be normalized together with the edge weights.

4.10 Graph Analysis

After generating the graphs, the final step of the solution is actually to analyse the generated graphs with respect to some of the statistical properties they might have. For this, multiple metrics were defined and analysed in two manners as it can be seen from figure 4.18. The graph metrics have been generated by both my colleague [12] and I but they have been studied independently, with me studying the **DTW Matching** aspect of the metrics.

4.10.1 Graph Metrics

For the analysis of the graphs, we have followed to pipeline described in the previous section, without the aggregation part. Since we wanted to see how different statistical properties can emerge from the graphs, the results from the analysis would be more meaningful if all of the original nodes are included.

Moreover, when choosing the metrics, we were interested in different graph algorithms that could yield results representing a single value. This fact was needed because

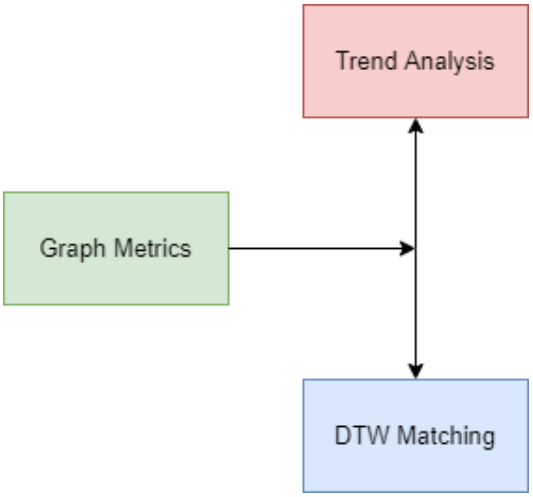


Figure 4.18: Graph analysis structure.

we want to analyse the evolution of this properties with respect to the time, hence it would not have bean feasible if the results were multivalued.

4.10.1.1 Maximum Spanning Arborescence weight

The first metric studied is the sum of the weights from **Maximum Spanning Arborescence** that can be constructed on a graph. For doing this, the **Minimum Spanning Arborescence** algorithm has been used which can extract the spanning arborescence from a directed weighted graph having the sum of the weights as small as possible.

However, since, in our case, the larger the weight of an edge, the more important it is, we actually needed to extract the tree that is capable of maximizing the sum of the weights of the edges it extracts. Therefore, we needed to do the following:

- First, the order relationship between the edges needs to be inverted. More concretely, since all the edges have their values between [0,1], an inverse operation was applied as described in equation 4.18. In doing so, the larger weights become the smaller ones and the smaller ones become the larger ones.

- After reverting the order relationship, the **Minimum Spanning Arborescence** algorithm is applied.

- Finally, all the edges from the tree are extracted, the inverse operation is applied again and the weights are summed.

$$inverse(w) = 1 - w \tag{4.18}$$

4.10.1.2 Size of the max clique

The next metric that was studied was the number of nodes in the maximum clique that can be found in the graph. The maximum clique of a graph is defined as the maximum complete subgraph. Because of the sparsification applied and the low number of edges that remained, we have decided to extract the maximum clique from the undirected version of the graph, in order to see which nodes are connected with which nodes, regardless of the direction of the connection. The conversion to an undirected graph from a directed graph is defined in equation 4.19.

$$\begin{gathered}\forall A \in nodes(G^d),\ B \in nodes(G^d) \Rightarrow\ E(A,B) \in edges(G^{ud}) \iff \\ E(A,B) \in edges(G^d)\ or\ E(B,A) \in edges(G^d)\end{gathered} \tag{4.19}$$

4.10.1.3 Average length of the shortest path

The next metric studied was the average length of the shortest path. This metric was applied on the unweighted version of the graph, meaning that we were only interested if there existed a path between any two nodes and, if it were, how many edges it contained. For doing this, we have actually made use of Dijkstra's algorithm and, after obtaining all the shortest paths inside the graph, the corresponding lengths have been averaged.

4.10.1.4 Average betweenness centrality

The last metric that has been studied is the betweenness centrality. The betweenness centrality is a measure of the centrality in a graph with respect to the nodes, having at its core the concept of the shortest paths. More concretely, the betweenness centrality for a node is defined as the number of shortest paths that pass through that node [19]. The formula for a node can be seen in equation 4.20.

$$B(u) = \sum_{u \neq v \neq w} \frac{\sigma_{v,w}(u)}{\sigma_{v,w}} \tag{4.20}$$

In our case, the shortest path for the metric has been defined on the unweighted version of the graph. Moreover, because the result of the metric is not a single value, but a measure for each node in the graph, the final result of the metric was the average of the betweenness centrality on all the nodes in the graph, therefore representing an overview of the centrality measurement of the graph being studied.

Algorithm 1 Dynamic Time Warping algorithm.

Input: s: array [1..n], t: array[1..n], w:int
Result: Similarity match computed using DTW.

```
// adapt the window size
 w := max(w, abs(n-m))

// initialize cost matrix
 DTW := array [0..n, 0..m]


for i := 0 to n do
    for j := 0 to m do
        DTW[i,j] := infinity
    end
end
// set starting point to 0
 DTW[0,0] := 0


// add locality constraint for i := 1 to n do
    for j := max(1, i-w) to min(m, i+w) do
        DTW[i,j] := 0
    end
end
// compute similarity
 for i := 1 to n do
    for j := max(1, i-w) to min(m, i+w) do
        cost := distance(s[i], t[j])
         DTW[i,j] := cost + minimum (
         DTW[i-1, j], // insertion
         DTW[i, j-1], // deletion
         DTW[i-1, j-1], // match
         )
    end
end
// return similarity
 return DTW[n,m]
```

4.10.2 DTW Matching

Considering the nature of the **RGW** approach, multiple graphs are generated from each trial. These graphs can be ordered with respect to the snapshot in time which they represent. Considering that each graph can be summarised using one of the metrics previously defined, the purpose of this final component is to identify if the evolution in time of the metrics holds any meaning.

For doing this, I had to define first which are the metric evolution sequences that are to be analysed. Considering that the purpose of the component is to compare between different evolutions, I have decided to actually study the evolution of graphs with respect to the stimulus they come from. More concretely, as specified in 4.3.2, there were 30 stimulus present in the dataset at 7 different gravitational constants. By ignoring the first value of the gravitational constant, each stimulus is present in the dataset 6 times. Each presence of a stimulus is counted as a trial and, for a trial, multiple graphs are generated.

The final decision that had to be made was how to compute the similarity between the metric evolution of two different stimuli. For this, I have decided to make use of the **Dynamic Time Warping algorithm** which is capable of measuring the similarity between two temporal sequences of different length, by creating a constrained path of matches between pairs of elements from the two sequences in order to minimize the cost of a distance function [20]. The pseudocode of the **DTW** algorithm is defined at 1. In our case, the distance function I have used was the absolute mean function. Moreover, because of the fact that a trial has on average around 20 windows, a locality constraint was also imposed, which meant that the matching could only take place for an index difference between the two points of maximum ± 10.

After applying the **DTW** between two stimuli, what we are actually interested in is not the similarity score, but rather the warping path that yields the best similarity measure.

Chapter 5

Detailed Design and Implementation

In this chapter, the detailed implementation of the conceptual components presented in the conceptual solution in 4.1. The design of the implementation is presented alongside relevant code together with the way the different modules communicate with eachother.

5.1 Technology Stack

The solution is implemented using the **Python** programming language. Python is an interpreted, high-level, general purpose programming language, integrating multiple programming paradigms such as the object-oriented paradigm or the functional paradigm. Moreover, it is not only garbage collected, but also dynamically typed, therefore proving itself to be a language not only robust, but also easy to use for complex task. However, it also presents some disadvantages, having a high memory consumption and being slower than other language alternatives.

Moreover, we have also made use of several python libraries and packages to help with the development of the solution. More concretely, the following libraries have been used:

- **Pandas**: Pandas is a library that offers specialized data structures and operations for manipulating tabular data, such as handling, parsing and creating *comma separated values* files. Moreover, the library is highly optimized because, even though its built on top of python, the critical operations are implemented in C.

- **Numpy**: Numpy offers specialized data structures and operations for handling and manipulating large, multi-dimensional arrays, together with a set of different mathematical tools and functions that can be computed and applied on them. This library is also highly optimized, having its critical components implemented in C.

- **Scikit-Learn**: Scikit-Learn is an open-source machine learning library, offering not only a set of classical machine learning algorithms, but also different tools for the

analysis of the corresponding models. It is designed to interoperate with scientific libraries such as Numpy.

- **DTW-Python**: DTW-Python is an open-source package offering a robust and comprehensive implementation of the DTW algorithm, together with its possible variants.
- **Networkx**: Networkx is a library offering a large set of data structures for handling graph-structured data, together with not only a robust and fast implementation of different graph algorithms, but also the means of visualizing the graph data.
- **Matplotlib**: Matplotlib is an open-source plotting library, offering a MATLAB-like interface and the ability of highly-customizing the generated plots.
- **Seaborn**: Seaborn is a library built on top of the Matplotlib library providing a high-level interface for drawing more aesthetically pleasing graphics.
- **Plotly**: Plotly is an open-source library providing an interface for generating highly-interactive graphs with a quality that is publication worthy, easing the analysis of the generated plots.
- **Visdom**: Visdom is a library allowing the generation of live, interactive plots. It provides the necessary tools for being able of generating rich, live visualization that can be viewed even remotely by having access to the server where visdom is enabled.
- **Torch**: Pytorch is an open source machine-learning library, offering a set of datastructures for creating, handling and manipulating complex mathematical operations with an easy to use interface. It provides tensor computing (similar to numpy), a large set of already built-in machine-learning modules and the ability to actually define your own machine-leraning algorithms. Moreover, it is highly optimized by having its Autograd component completely implemented in C.

5.2 System architecture

In this section, the detailed architecture of the solution is presented, with each component of the system being the direct result of the conceptual architecture present in 4.1. The architecture can be viewed in figure 5.1.

As it can be seen from the figure, our design choice was to actually implement each part of the system as a different component, completely independent with respect to how it's run but dependent on the results of the other components. This choice has been made with respect to several design principles, most notably the reusability and modularization aspects of the solution. However, in some cases, because of certain hardware limitations encountered, some components needed to be created in order to overcome this issues, such as *Raw Data Filter module* or *Trial Window Configuration module*. Moreover, the

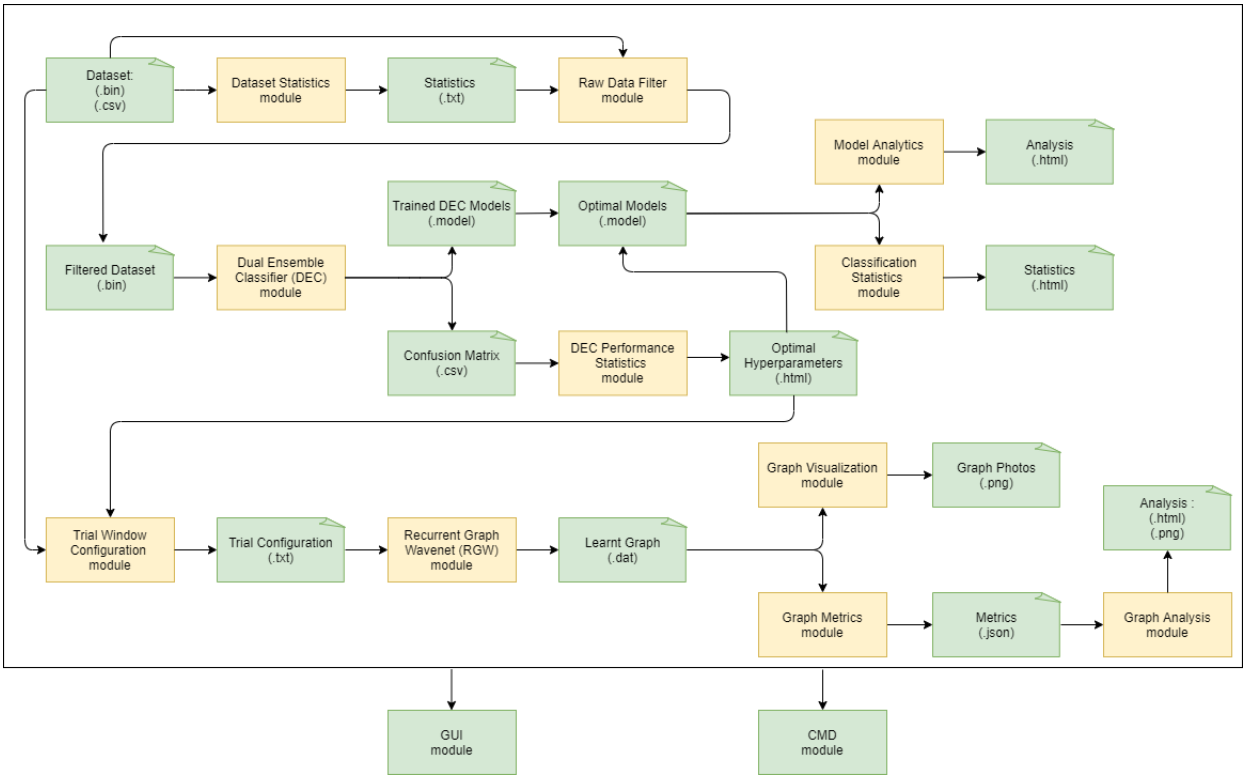


Figure 5.1: Detailed architecture of the system.

communication and flow of information in the architecture is actually sequential, having the starting point in the top left corner of the figure.

The result of the design phase is a highly modularized application, having multiple independent components with their own configurable running interface. Moreover, the communication between modules respects the principle of low coupling, with the communication happening using files. More concretely, the results of one module in a file form are to be used afterwards by another module. Moreover, each module is also parametrized, giving the programmers the ability to use it however they see fit, therefore making this application highly generalized.

In addition to the application, two additional wrapping interfaces have been provided for the programmer to interact with the solution as a whole. More specifically, a *Graphical User Interface* and a *Command Line Interface* have been developed which will be explained in detail in the following sections.

As one last thing, following the architecture's principle of separation of modules, the folder structure of the project's implementation follows the same reasoning, with each module having its own separate folder (with a variable depth). This fact can be seen in figure 5.2. Having the structure of the project defined in this manner eases the imple-

```
graph_generation/
├── cmd/
├── dataset_statistics/
├── dual_ensemble_classifier/
│   └── ml/
├── dual_ensemble_classifier_performance_statistics/
├── graph_analysis/
├── graph_metrics/
├── graph_visualization/
├── gui/
│   └── tabs/
├── model_analytics/
├── model_classification_statistics/
├── raw_data_filter/
├── reader/
├── recurrent_graph_wavenet/
│   └── ml/
├── trial_window_configuration/
└── util/
```

Figure 5.2: The folder structure of the project's implementation.

mentation process and gives the programmer an overview feeling of how the application's architecture is actually structured.

5.3 Detailed Functionality

In this subsection, each component of the project will be presented. It is important to note here that the development process has been a sequential one, meaning that each component has been developed one after the other. More over, the process of creating this solution was the result of a joint effort between my colleague [12] and I. Therefore, in this section, I will focus only on the functionality that I have provided and, for my colleague's part, I will just specify the need for those functions.

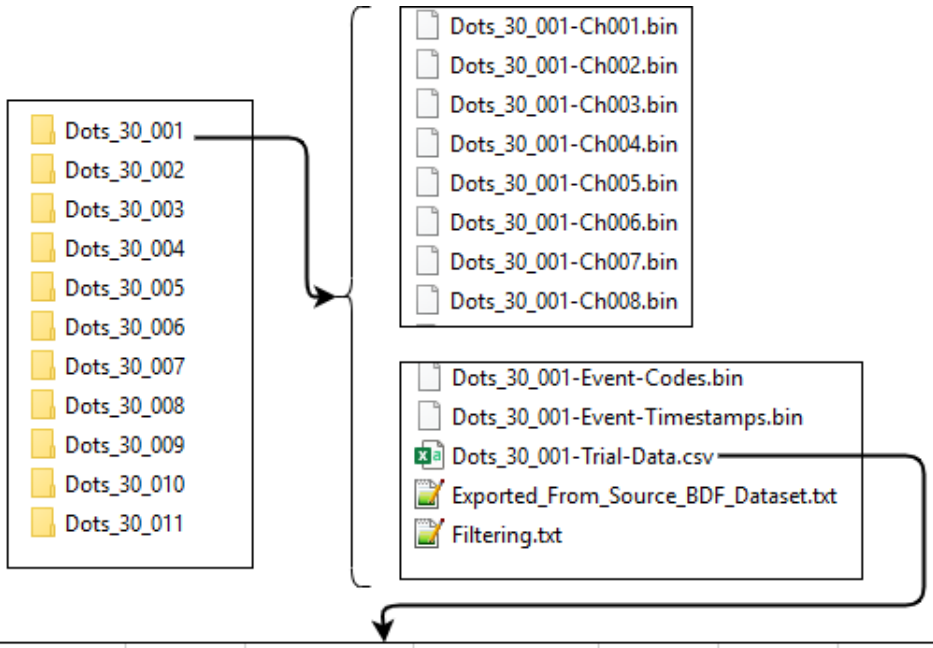
Moreover, it is important here to note that some aspects are a general truth for all the modules presented in this section and, therefore, they will be introduced here. More concretely, the following facts apply to all components:

- The entry point of each component is a wrapper function. In order to use the module, this function needs to be called. The name of the function is actually the name of the component written in a snake case manner.
- Each component has as an input a folder or a file from where it reads data. The path to this folder is specified as a parameter to the wrapper function.
- Each module is parameterized to the greatest extent possible, therefore allowing a wide set of operations. These parameters can be set through the wrapper function.

- The result of each module is one folder where it generates data. This data is either used by a programmer (looking at plots) or is to be used by the following module in the architecture (as seen in figure 5.1). Moreover, this folder is created by the module at the specified path, which is also parameterized.
- For each module, the focus will be on the functionality it provides with respect to the functions and classes it defines together with the design aspects and challenges that occurred when developing them.

5.3.1 Dataset Statistics

The purpose of this module is the analysis of certain statistical properties of the dataset as specified in 4.3. The goal of this module is identifying if there are certain constraints imposed by the dataset that will affect the development and functionality of the following components, with respect to both hardware resources but also implementation limitations.



Trial	Filename	G	Stimulus	CorrectResp	%	Response	Vb response
1	Stim_29_G_0.00.bmp	0	29	poseta / geanta	100	3	Nothing
2	Stim_08_G_0.00.bmp	0	8	topor / secure	100	3	Nothing
3	Stim_05_G_0.00.bmp	0	5	oala / cratita	100	3	Nothing

Figure 5.3: The structure of the input dataset.

The input of this module is actually the dataset provided by the TINS institute. The structure of the dataset can be seen in figure 5.3. For each subject, there is one folder

provided containing multiple binary files where the values recorded from each channel are recorded. Moreover, it provides an additional set of binary files based on which the events and the trials from the data can be identified and a tabular file where the trial properties (such as what stimulus was shown) are defined.

This module consists of the following functions:

• **dataset_statistics()**

This function represents the wrapper function of the module. It is used in order to access the entire functionality of the module by specifying the input path of the dataset and the output path where the results of the analysis will be saved.

• **read_trial_metadata()**

This function has been implemented by my colleague and has the purpose of reading the necessary data for the analysis.

• **compute_statistics()**

This function uses the data provided by **read_trial_metadata()**. It computes multiple histograms where the distribution of the trial lengths are shown with respect to the subjects and the response and stimulus label classes. Moreover, it also computes certain properties of these distributions such as the mean value, the median value, how many trials are in certain ranges and so on.

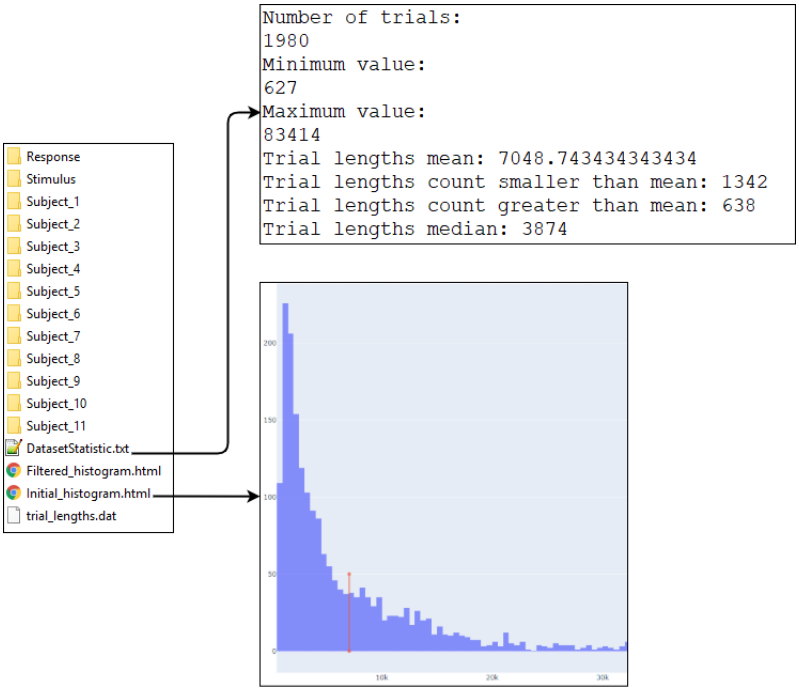


Figure 5.4: The output structure of the DatasetStatistics module.

The results of this module are saved in a folder called *DatasetStatistics* in the provided output path. The structure of the results can be seen in figure 5.4. The results are organized based on the way the aggregation for the statistics has been made. For the

distribution, a file is created where the results are displayed. Moreover, multiple plots are generated having the layout as seen in the figure. The results of the analysis are structured to the way the aggregation took place, either on label classes or on subjects.

5.3.2 Raw Data Filter

This module has been developed by my colleague and is the result of the observations from the module 4.3. Because of the distribution of the data, this preprocessing step was needed.

5.3.3 Dual Ensemble Classifier

The purpose of this module is actually the defining and training of the *Dual Ensemble Classifier* model as specified in 4.4. Based on a set of specified hyperparameters for trial segmentation, it creates the necessary dataset structure for the model. Moreover, it also trains the DEC model computes the classification performance.

The input of this module is the result of the *Raw Data Filter* module, which is actually a set of of binary files for each subject where the trial data has been saved after the preprocessing step. The structure of these files has been defined by my colleague [12].

This module consists of the following functions:

• **dual_ensemble_classifier()**

This function represents the wrapper function of the module. In order to access the entire functionality of the module, this function needs to be called and a set of parameters need to be specified. First, it needs to receive the input path of where the results of the preprocessing step are stored. Moreover, it needs to receive the parameters controlling the trial segmentation process, namely the window size and window offset. Finally, it also requires an output path where a folder will be created and the results will be stored.

• **read_trial_data()**

This function has been developed by my colleague and has the purpose of reading the trial data for each subject from the result of the preprocessing step.

• **create_dataset()**

This function has also been developed by my colleague and has the purpose of actually creating the dataset for the DEC model by applying the Trial Segmentation principle explained in 4.4.2. Moreover, it also splits the dataset into 3 individual sets: training set, cross-validation set and test set.

• **standardize_dataset()**

In this function, after the dataset has been created and the 3 training sets defined, the standardization step occurs. Standardization is necessary in any machine-learning application because it standardizes the range of all the values of all the features by making them have the same distribution with a mean of 0 and a standard deviation of 1. In our concrete case, its purpose is to actually standardize the values amongst channels in order for all of them to obey to this principle.

Moreover, each of the defined datasets needs to be standardized. However, since the cross-validation set and the test set are used for the analysis of the model with respect to how well it generalizes what it has learnt, these two sets will not be standardized using their own internal distribution, but rather the distribution of the train set on which the model was trained on.

Furthermore, there was one more challenge that needed to be faced here. Because the EEG headset is a physical device measuring the electrical potential inside the skull of a subject, it is biased by the physiological condition of that subject, such as fat, skin, hair and so on. Because of this, the values recorded by the headset will be consistent only on the same subject, but will be inconsistent amongst subjects. Therefore, the data from each subject needed to be standardized on its own with respect to the 3 sets defined.

```
# for all train examples
for index in range(len(channel_response_train[channel_index])):
   # if the example is for the current subject
   if subjects_train[index] == subject:

      # standardize examples
      channel_response_train[channel_index][index] = list(
         map(lambda elem: (elem - mean) / std,
         channel_response_train[channel_index][index])
      )
      channel_stimulus_train[channel_index][index] = list(
         map(lambda elem: (elem - mean) / std,
         channel_stimulus_train[channel_index][index])
      )
```

Listing 5.1: Train set standardization.

The standardization process can be seen in 5.1. Having the standardization parameters defined on the train set for a certain subject, the examples in the train set are standardized only if they belong to the corresponding subject. Otherwise, they are to be standardized at the next iteration. Moreover, the same principle will be applied to the other two sets, with the observation that the same standardization parameters from the training set will be used.

• **create_loaders()**

This function has been developed by my colleague and has the purpose of modifying the datasets into a structure with which the model can be easily trained on.

• **save_datasets()**

This function has the purpose of actually saving the defined datasets of the current DEC model. This decision has been taken because of the fact that the creation and standardization of the dataset of the DEC model is quite a long process and there are certain modules that need these datasets in order to actually analyse the model. Therefore, in order to improve the other modules' computational time, I have decided to make the trade-off between time and memory by actually saving all the datasets the model is trained on. In order to try and save space, the datasets have been saved in binary files, with their

name indicating which they are and certain parameters that help distinguish their actual structure. This process can be viewed in 5.2.

```
# save stimulus train dataset to file
channel_stimulus_train = np.array(channel_stimulus_train)

channel_stimulus_train.tofile(os.path.join(training_path,
    f"channel_stimulus_train"
    f"_{channel_stimulus_train.shape[0]}"
    f"_{channel_stimulus_train.shape[1]}"
    f"_{channel_stimulus_train.shape[2]}.dat"))
```

Listing 5.2: Train stimulus dataset save.

• **initialize_model()**

In this function, after having the dataset loaders defined and standardized, the DEC model is initialized, the training is started and, finally, after the training finishes, the prediction is made. It is important to specify here that this function is merely an orchestrator of the training because the actual functionality is provided in the *DualEnsembleClassifierModel* class.

• **DualEnsembleClassifierModel class**

This class represents the core of the *Dual Ensemble Classifier* module. Inside this class, the architecture of the model is defined, the training takes place and the prediction is made.

• **DualEnsembleClassifierModel.create_DEC()**

This function has been implemented by my colleague and has the purpose of creating the architecture of the DEC model consisting of multiple DNN models, as specified in 4.4.1.

• **DualEnsembleClassifierModel.create_DNN()**

In this function, the core cell of the DEC model is defined. Two neural networks are defined separately, each having one hidden layer, one activation function, one dropout layer and one output layer. The sizes of these layers are parametrized, the hidden layer having a size proportional to the input size and the output layer having a size that is equal to the number of labels a class has, namely 6 for stimulus and 3 for response. The computation of the hidden layer size can be seen in 5.3.

```
def get_hidden_layer_size(example_length, output_size):
    """
    Computes the hidden layer size proportional to the input size
    """
    return int(example_length * 2 / 3 + output_size)
```

Listing 5.3: Computation of the hidden layer size.

• **DualEnsembleClassifierModel.forward()**

This function has been developed by my colleague and has the role of propagating an input example through the DEC model.

• **DualEnsembleClassifierModel.fit()**

This function has been implemented by my colleague and has the role of performing the training of the DEC model.

• **DualEnsembleClassifierModel.dual_loss_aggregation()**

This function has the purpose of implementing the duality of the DEC model. As specified in 4.4.1, the duality of the model actually comes from the way the losses are aggregated. Since the purpose of this aggregation is to actually force the two independent models to optimize eachother, a multiplication between the two was necessary. Moreover, since certain problems could arise such as only one model would actually train and the other would not, the aggregation also needed to take into consideration the difference between the two losses. With this in mind, the loss definition can be seen in 5.4.

```
# mutual optimization
loss = first_loss * second_loss

# avoid single sided training
if first_loss < second_loss:
   loss += (second_loss - first_loss)
else:
   loss += (first_loss - second_loss)

return loss
```

Listing 5.4: Dual loss aggregation.

• **DualEnsembleClassifierModel.plot_to_vizdom()**

This function has the role of actually creating learning curves that are displayed live. More concretely, by making use of the library **vizdom**, we can actually see how the model trains in real time by starting a server on the machine where the training takes place and watching it live remotely. This feature is especially useful when the training takes a long time and you want to see how the model performs whilst it trains. An example for adding the loss of an epoch on the train set and cross-validation set can be viewed in 5.5. The access to the server where the live plotting takes place is made through the **viz** object, where we specify what values to add and to what plots.

```
viz.line(X = np.array([epoch]), Y = np.array([epoch_loss / count]),
   win = 'Epoch number', name = 'ELT ' + viz_name, update = 'append')

viz.line(X = np.array([epoch]), Y = np.array([cv_loss]),
   win = 'Epoch number', name = 'ELCV ' + viz_name, update = 'append')
```

Listing 5.5: Live plotting.

• **DualEnsembleClassifierModel.plot_learning_curves()**

This function has the purpose of plotting the same learning curves as the ones plotted with vizdom, with the difference that these plots are actually saved together with the model at the end of the training. Since vizdom's interface did not allow for the plots to actually be saved after the execution finished, we needed to do this by hand in order to see how the model has learnt if we wanted, at a later point in time after the execution, to manually analyse the model.

• **DualEnsembleClassifierModel.predict()**

This function has been implemented by my colleague and has the purpose of generating the confusion matrices with respect to the model's performance after the training finished.

• **DualEnsembleClassifierModel.predict_for_classification_statistics()**

This function has been developed in order to aid one of the modules defined later in the pipeline, namely *Classification Statistics.*. More concretely, this function, given a loader that could be either the train loader, cross-validation loader or test-loader, creates a total of 4 arrays as follows:

- An array containing all the examples from the loader where the stimulus was correctly classified.
- An array containing all the examples from the loader where the stimulus was incorrectly classified.
- An array containing all the examples from the loader where the response was correctly classified.
- An array containing all the examples from the loader where the response was misclassified.

In order to do this, each example had to be manually passed through the DEC model and the values of the output layer analysed for each passing in order to see which was the prediction. Afterwards, depending on the actual label and the predicted label, each example needed to be placed in the corresponding array. This process can be viewed for the stimulus class labels in 5.6.

```
# Get predictions from the maximum value
# Get predictions from the maximum value
_, first_predicted = torch.max(first_agg_output.data, 1)
_, second_predicted = torch.max(second_agg_output.data, 1)

first_predicted = first_predicted.tolist()
second_predicted = second_predicted.tolist()

# get labels
first_labels = batch[-4].tolist()
second_labels = batch[-3].tolist()

# add to corresponding array
if first_predicted[0] == first_labels[0]:
    stimulus_correctly_classified[first_labels[0]].append(example_count)
else:
    stimulus_incorrectly_classified[first_labels[0]].append(
   example_count)
```

Listing 5.6: Stimulus predictions.

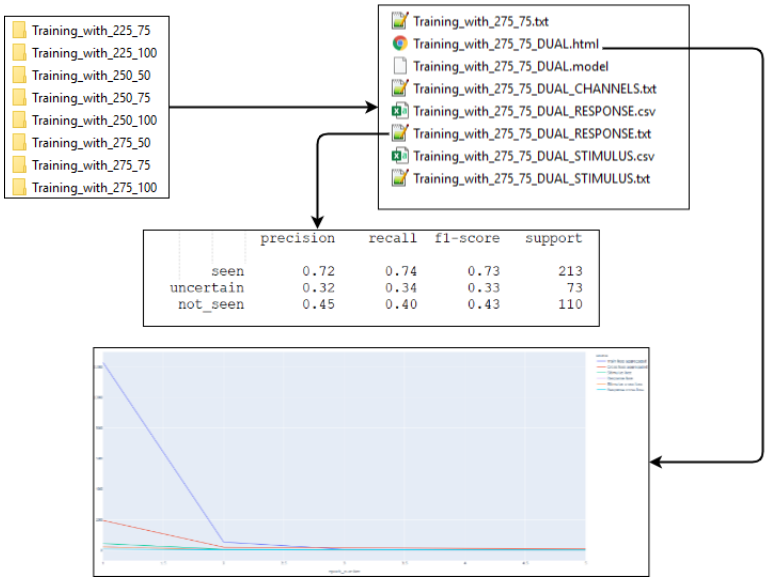


Figure 5.5: The output structure of the DualEnsembleClassifier module.

At the end of the module's execution, one folder is created in the output path where the results are stored. In order to distinguish between different runs, the name of this folder contains the hyper-parameters of the trial segmentation. Inside this folder, the actual model has been saved in a binary format together with its prediction performance under the form of the consufion matrix and the learning curves created during training. The structure of the results can be seen in figure 5.5.

5.3.4 Dual Ensemble Classifier Performance Statistics

The goal of this module is to actually analyse the performance of the DEC model with respect to the choice of the hyperparameters in order to identify the optimal set of segmentation parameters, namely window size and window offset, as described in 4.5.

The input of this module is actually the output folder of the DEC module. More specifically, all of the folders inside the output folder of the previous module are to be used inside this module in order to analyse the performance. However, the only data that this module uses is the confusion matrices which have been saved as tabular files.

This module consists of the following functions:

- **dual_ensemble_classifier_performance_statistics()**

This function represents the wrapper function of the module. In order to access the functionality provided by this module, the path to the output folder of the DEC module needs to be provided. Moreover, it also needs to receive the output path where the results of the analysis will be saved.

• **plot_initial_performance()**

The purpose of this function is to actually perform the first step in the performance statistics module, namely the *Initial tuning and filtering* as specified in 4.5.2. More concretely, for each set of hyperparameters on which the DEC model has been trained on, the confusion matrices for both the stimulus and response classifiers are read. Afterwards, for each individual model, the average f1-score is computed. Then, the results of the analysis are written to a text file in order for the user to be able to see how the initial performance of the DEC model looks like. Finally, the joint performance of the two classifier for each set of hyperparameters is plotted. Nevertheless, the choice of the subset of hyperparameters to be further studied is also defined.

```
# get minimum and maximum for both axes
minimum_x = min(points_combined, key = lambda x: x[0])
maximum_x = max(points_combined, key = lambda x: x[0])
minimum_y = min(points_combined, key = lambda x: x[1])
maximum_y = max(points_combined, key = lambda x: x[1])

# plot the points
for i in range(len(points_combined)):
    x = points_combined[i][0]
    y = points_combined[i][1]
    plt.plot(x, y, 'bo')
    plt.text(x * 1.005, y * 1.005, division_list_stimulus[i], fontsize =
    8)

# set axis limits
plt.xlim((minimum_x[0] - 0.005, maximum_x[0] + 0.005))
plt.ylim((minimum_y[1] - 0.005, maximum_y[1] + 0.005))
```

Listing 5.7: Joint performance plotting.

The code for the joint plot can be viewed in 5.7. The most difficult part here was to actually make the plots in order to be able to contain all the 27 sets of hyperparameters displayed as dots inside the plot with each point being individually labeled. For this, a manipulation of the axis was necessary in order for the labels to actually be visible and distinguishable.

• **plot_distribution_for_multiple_runs()**

This function has been developed by my colleague and, based on the analysis generated by **plot_initial_performance()**, the analysis is taken one step further and the final choice of hyperparameters is defined.

At the end of this module's execution, at the specified output path, a folder is created inside which multiple plots and text files are present as seen in figure 5.6. The plots present here are different visualizations of how the performance of the DEC model fluctuates with respect to the choice of hyper-parameters and the classification problem being studied. Moreover, in the text files, different results from the analysis are presented, such as the best choice of hyperparameters for one classification problem or the average f1-score over all the DEC models.

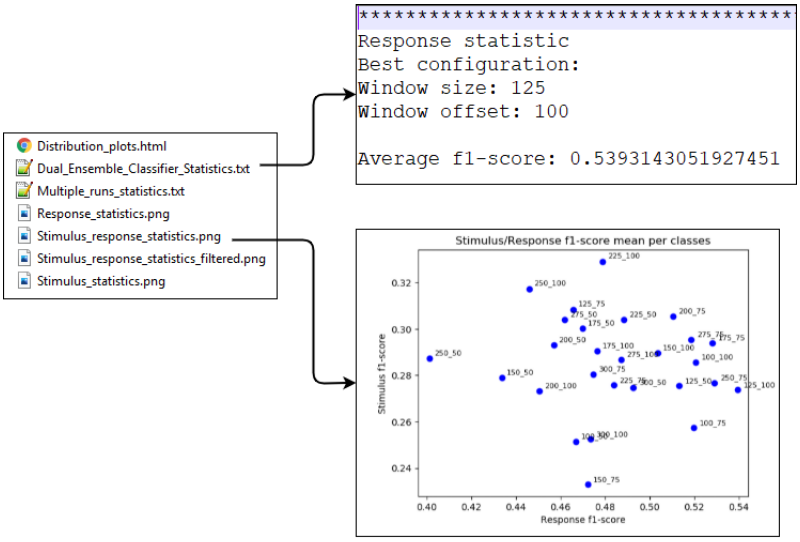


Figure 5.6: The output structure of the Performance Statistics module.

5.3.5 Classification statistics

The purpose of this module is to actually analyse the classification performance with respect to the two classification problems and the trial's length from where the examples were sampled, as explained in detail in 4.6. The final goal of this module is to actually discover the limitations that stopped the DEC model from reaching an even better classification performance.

The input of this module consists of the result of the *DEC module*, namely an individual train on an individual set of hyper-parameters. Moreover, it also uses the output of the *Dataset Statistics* module, where the length of the trials are defined.

This module consists of the following functions:

• **classification_statistics()**

This function represents the wrapper function of the module that is to be used in order to access the functionality provided. In order for it to start executing, several parameters need to be specified. First of all, the path to the DEC model that needs to be analysed must be specified, together with the path to the result of the *Dataset Statistics* module. Afterwards, since the analysis is done on a set of examples, which set from the train, cross-validation and test set should be used must be specified.

Once the function is called, the first step is to actually load the model. Luckily, with the help of **pytorch**, this step was quite straightforward as seen in 5.8. However, in order to be able to use the library to help us load the model, the model actually needed to be instantiated first and the entire architecture defined. Luckily, because of the class **DualEnsembleClassifierModel**, this step proved to be easy.

```
# instantiate model
model = DualNeuralNetwork(
```

```
    (
        [
            example_length,
            stimulus_hidden_size,
            STIMULUS_OUTPUT_SIZE
        ],
        [
            example_length,
            response_hidden_size,
            RESPONSE_OUTPUT_SIZE
        ]
    ),
    NUMBER_OF_CHANNELS
)

# load the model
model.load_model_from_file(os.path.join(model_directory,
   f"Training_with_{window_size}_{window_offset}_DUAL.model"))
```

Listing 5.8: Load model.

After loading the model, the next step in the function is to read the datasets on which the model was trained on. Because of the function **save_datasets()** from the DEC module, this step is relatively fast since all we need to do is to just read the saved datasets and bring them into the array shape we can work with.

Afterwards, it is time to make use of the function **predict_for_classification_statistics** from the DualEnsembleClassifierModel class. As previously specified, this function, given a loader, will actually create 4 arrays where the number of correctly and incorrectly classified examples for each of the two classification problems is present.

Having these arrays defined, the last step is to actually aggregate and count them, by grouping them with respect to either the set they come from, either the classification problem or the label itself. Each of these aggregations is plotted individually with respect to the trial length under the form of a histogram using the function **plot_histogram()**.

- **plot_histogram()**

For each grouping and aggregation, this function is called in order to draw the necessary histogram where the bars represent either the number of correctly classified examples or incorrectly classified examples from the corresponding grouping. In addition, one last histogram is created where the misclassification percentage is presented with respect to the trial length.

At the end of the module's execution, at the specified output path, a folder is created containing inside multiple folders (one for each dataset), each of them containing multiple plots depending on how the grouping and aggregation has been performed prior to the ploting. The structure can be seen in figure 5.7.

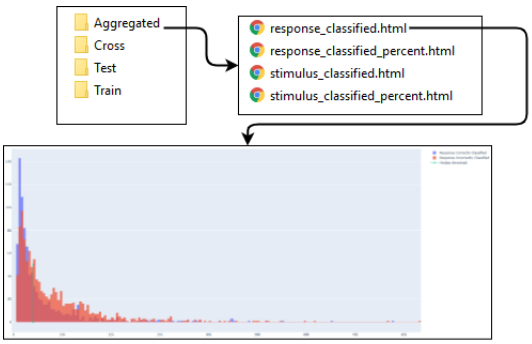


Figure 5.7: The output structure of the Classification Statistics module.

5.3.6 Model Analytics

This module has been developed by my colleague and has the goal of analysing what the model has learnt with respect to the adjustable weights of the DEC model on the input layer.

5.3.7 Trial Window Configuration

The goal of this module is to actually ease the load that will be done by the next module, namely *Recurrent Graph Wavenet*. As it was the case for the *RawDataFilter* module, based on the observation from *Dataset Statistics* and *Classification Statistics*, a preprocessing step is necessary before training the RGW model. More specifically, in this module, the concept of the *Trial Segmentation* is applied on the initial dataset to prepare the data for the next module by creating, for each trial, a set of limits representing the bounds of each window on which the RGW model is to be applied on.

The input of this module actually consists of the dataset provided by the TINS institute, having the structure laid out in figure 5.3. However, in this part, the only data that will be used is how the trials are defined with respect to the event codes present in the dataset.

The module consists of the following functions:

• **trial_window_configuration()**

This function represents the wrapper function of the module that needs to be called in order to have access to the provided functionality. It needs to receive as an input parameter the location of the dataset together with the location of the output path where the results are to be generated. Moreover, it also needs to receive the trial segmentation parameters in order to be able to construct the necessary windows, namely the window size, the window offset and the threshold specifying how much should be left from the trial after filtering it out. The trial segmentation process can be viewed in figure 4.9.

In this function, for each subject, the corresponding trials' metadata is read and, for each trial, only the values left after filtering out the middle of the trial are kept. After the

trials are filtered, each trial is split into the corresponding windows using the **split_trial()** function as seen in 5.9.

```
# create windows for stimulus
split_trial(window_file, trial_start_timestamp,
    trial_start_timestamp + threshold // 2 - 1, window_size,
    window_offset)

# create windows for response
split_trial(window_file, trial_end_timestamp - threshold // 2 + 1,
    trial_end_timestamp, window_size, window_offset)
```

Listing 5.9: Trial filtering.

- **split_trial()**

This function serves the purpose of actually creating the window limits using the provided trial segmentation parameters. Based on a start and end value, it creates a window which is slided over the data in order to compute the bounds of each window present between the two timestamps of the trial. This process can be viewed in 5.10.

```
while True:

    # compute window limit
    window_end = window_start + window_size - 1

    if window_end == trial_end:
        windows_coordinates.append([window_start, window_end])
        break

    if window_end > trial_end:
        windows_coordinates[-1][1] = trial_end
        break

    # append window bounds
    windows_coordinates.append([window_start, window_end])
    window_start += window_offset

# write window limits to a file
for coordinate in windows_coordinates:
    print(f'{coordinate[0]} {coordinate[1]}', file = file)
```

Listing 5.10: Window Bounds Computation.

At the end of the execution of this module, at the specified output path, multiple files are created having. Each of these files corresponds to a trial of a specific subject. The content of one of these files represents all the window bounds for the corresponding trial, each bound written on a separate line, with the windows being ordered ascendingly with respect to the time. The structure of the output result of this module can be viewed in figure 5.8.

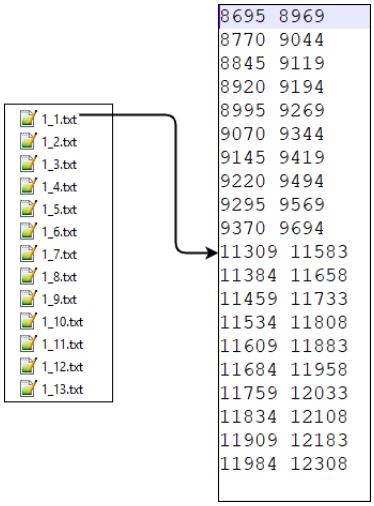


Figure 5.8: The output structure of the Trial Window Configuration module.

5.3.8 Recurrent Graph WaveNet

The purpose of this module is to actually implement the conceptual architecture described in 4.7.2. More specifically, it creates the dataset from the specified window, defines and trains the Graph Wavenet model and, finally, it computes the performance analysis.

The input of this module is the dataset provided by the TINS institute which has the structure presented in 5.3. Moreover, it also uses the data generated by the *Trial Window Configuration* module in order to be able to actually create the windows. Nevertheless, it also needs access to the support matrices being used, namely the *Functional Networks* and the *self-adaptive learnt matrix of the previous window*.

The module consists of the following functions:

• **recurrent_graph_wavenet()**

This function represents the wrapper function of the module. It incorporates the entire functionality presented so far and has a great number of parameters that need to be set in order to be able to run this module. First of all, the input data paths need to be defined together with an output path to where the model should be saved. Moreover, it also needs to receive the paths towards the place where the functional networks are stored and the weight matrices of the previous windows are defined. Furthermore, it also needs to receive the parameters regarding the window on which the training should take place, such as the subject, the trial and the window index. Finally, it needs to receive the parameters specifying different configurations for the Graph WaveNet model and the training itself, such as the number of blocks and layers, the learning rate, the batch size, the parameters controlling the cross-validation set creation and so forth.

• **save_running_parameters()**

This function has been developed by my colleague and has the purpose of saving all

the parameters of the module for a specific execution.

- **load_support_matrices()**

This function has also been developed by my colleague with the goal of loading and processing the necessary support matrices that the model should use.

- **create_loader_window()**

This function has the purpose of actually reading, processing and creating the loaders for both the train set and the cross-validation set. After reading the data based on the input parameters, the next step of the function is to actually split the dataset into the train set and the cross-validation set with respect to the *rolling origin forward validation* concept presented in figure 4.16. For each of these splits, it firsts applies the standardization process. More concretely, both of the two generated sets are standardized using the standardization parameters computed only on the train set.

Afterwards, it applies the principle of the rolling window in order to actually define each example individually on both these sets as seen in figure 4.15. This process can be viewed in 5.11.

```
# define sequence bounds
sequence_start = 0
sequence_end = len(values[0]) - 1

train_x = []
train_y = []

# while an example can be created
while (sequence_start + input_length + output_length - 1) <= sequence_end:
    train_x.append([])
    train_y.append([])

    # for each channel
    for channel_index in range(NUMBER_OF_CHANNELS):

        # input window
        train_x[-1].append(values[channel_index][sequence_start:(sequence_start + input_length)])

        # output window
        train_y[-1].append(values[channel_index][sequence_start + input_length:sequence_start + input_length + output_length])

    sequence_start += 1

train_x = np.array(train_x)
train_y = np.array(train_y)

# create loader
dataset = GraphWavenetDataset(train_x, train_y)
```

```
dataset_loader = DataLoader(dataset, batch_size, shuffle)
```

Listing 5.11: RGW Sliding Window

Finally, as it can be seen at the end of the provided code snippet from 5.11, after creating all of the examples, the example arrays are converted into the actual loaders which will be further used by the Graph Wavenet model.

• **GraphWavenetModel class**

This class has been developed by the authors of the original paper [9] and has been integrated in our solution because no library was provided for it.

• **TrainEngine class**

This class represents the core of the *Recurrent Graph WaveNet* module. Inside this class, the model is instantiated, the training takes place and the prediction of the model is performed.

• **TrainEngine.create_model()**

This function has been implemented by my colleague and serves the purpose of initializing the Graph WaveNet model.

• **TrainEngine.create_optimizer()**

This function has been implemented by my colleague and has the goal of defining the optimizer that will be used by the model during training.

• **TrainEngine.mean_absolute_scaled_error()**

This function has the purpose of implementing the *Mean Absolute Scaled Error* defined in equation 4.16. The implementation can be seen in 5.12. The actual implementation of the MASE function is defined example wise, meaning that, for each example, the MAE value component of the function is computed with respect to the prediction from that example and the naive forecast component is computed with respect to the values from the input example.

```
for node in range(self.number_of_nodes):

    # compute naive forecast
    aux = []
    for index in range(1, len(input[0][0][node])):
        aux.append(float(abs(input[0][0][node][index] - input[0][0][node][
    index - 1])))

    # compute scale factor
    mean = np.array(aux).mean()

    # compute MAE
    mase[node] += abs((real[0][0][node] - predicted[0][0][node]).detach()
    .cpu().numpy().mean())

    # compute MASE
    if mean != 0:
        mase[node] /= mean
```

Listing 5.12: Implementation of the MASE function.

• **TrainEngine.train()**

This function represents the core of the *TrainEngine* class. In this function, for each of the splits defined using the *rolling origin forward validation* approach, an entire training of the Graph Wavenet model takes place. For each of this trainings, the best model is kept track off with respect to its performance on the cross-validation set in order for the *early stopping* mechanism to take place. This mechanism can be viewed in 5.13.

```
# if a new best model has been found
if cross_epoch_loss[-1] <= min_cross_error:
    self.best_model[-1] = copy.deepcopy(self.model)
    last_update = 0
    min_cross_error = cross_epoch_loss[-1]

    log('New best model!', self.log_file, self.widget)

# otherwise increase the time since the last best model has been found
else:
    last_update += 1

# early stopping if more than then epochs have passed
if last_update >= 10:
    log("EARLY STOP", self.log_file, self.widget)
    break
```

Listing 5.13: Implementation of the MASE function.

• **TrainEngine.full_train()**

This function has the goal of implementing the concept of the full training specified in 4.8.3. It provides almost the same functionality as the **train()** function, with the difference that the concept of the *rolling origin forward validation* is not present anymore together with the validation on the cross-validation set.

After the execution of the module has finished, at the specified output path, multiple files and plots are generated for each of the splits defined by the *rolling origin forward validation* approach. In these files, information such as the average MASE score, the prediction of the model or the learning curves can be seen. The structure can be viewed in 5.9.

5.3.9 Graph Visualization

The purpose of this module is to implement the concept of the graph visualization presented in 4.9. Multiple plots are to be generated, having the channels aggregated in the corresponding brain regions and their values normalized in order to be able to display them properly.

The input of this module is the result of the *Recurrent Graph Wavenet* module. More specifically, it needs to have access to the self-adaptive learnt adjacency matrices for each window of each trial that the model has been trained on.

The following functions are defined:

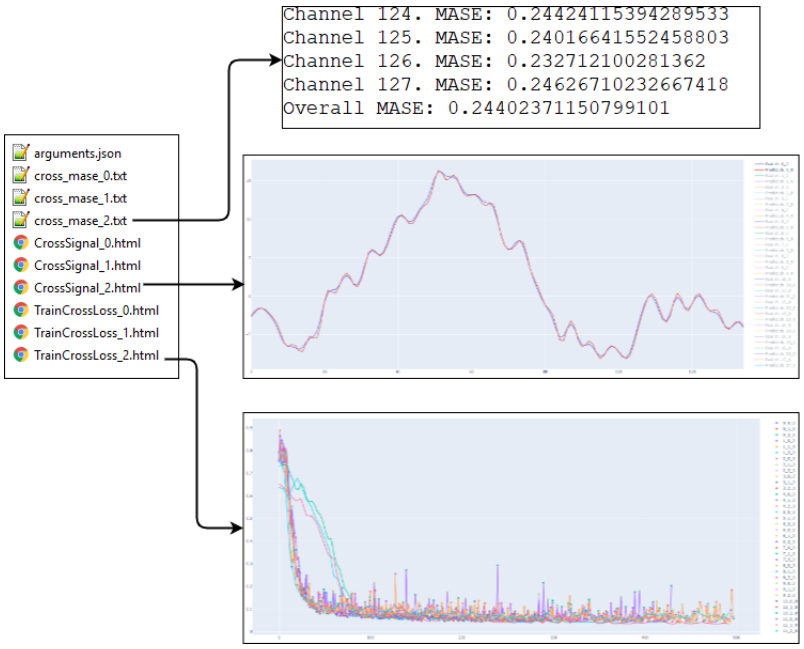


Figure 5.9: The output structure of the Recurrent Graph Wavenet module.

• **graph_regions_plot_individual()**

This function has the purpose of displaying in a graph form the matrix of a window from a trial. After defining the necessary parameters such as the path from where the matrix can be read and the window and trial from where it comes, the next step is to actually read the matrix and aggregate it using the **aggregate_channels** function. After the aggregated matrix has been defined, the **plot_graph** function is called in order to display the matrix.

• **aggregate_channels()**

This function has been developed by my colleague and has the purpose of aggregating the channels into the corresponding brain regions as specified in 4.9.2.

• **plot_graph()**

This function has the goal of actually displaying a graph. It starts but sorting the edges in an ascending manner in order for them to be plotted this way. By doing this, the most important edges will be displayed on the top of the others. Afterwards, it defines the layout of the graph in order to resemble the same layout as the one presented in the figure 4.17. Finally, when plotting the graph, not only was a colormap provided in order to display the value of the weights, but also the elements from the drawing have a variable width with respect to the weight. It is important to note here that even though node weights are considered to be self-loops, they are actually displayed on the node itself for aesthetic purposes.

At the end of the execution of this module, at the specified output path, a complex folder structure is created. In this folder structure, for each trial, multiple folders are

created where the graph plots are placed either by themselves, or concatenated with the other graphs from the other windows of the trial. This structure can be viewed in 5.10.

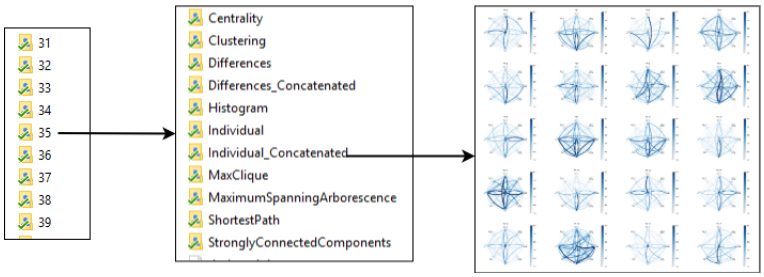


Figure 5.10: The output structure of the Graph Visualization module.

5.3.10 Graph Metrics

The goal of this module is the implementation of different graph metrics that are studied on the generated graphs, as described in 4.10.1. More specifically, for each matrix that has been generated on a window of a trial, a set of metrices is applied and their results are saved for further use.

The input of this module is the same one as for the module *Graph Visualization*. More specifically, this module needs to have access on the self-adaptive learnt matrices that have been generated by the *Recurrent Graph Wavenet* module.

The following functions are defined in the module:

• **graph_metrics()**

This function represents the wrapper function of the module which needs to be called in order to access the functionality provided. In order to use it, some parameters must be defined. First of all, the path to where the matrices are stored needs to be given toghether with the output path where the module should save the results. Moreover, it also needs to receive the trial index on which the metric computation is performed. After reading the graphs, they follow the same processing pipeline as the one from *Graph Visualization*, with the difference that the channels from the graphs are not aggregated anymore.

• **graph_clique()**

This function has the purpose of computing the size of the maximum clique for each matrix provided. As defined in 4.10.1.2, the clique is computed on the undirected version of the graph. After converting the graph to an undirected one, the clique is computed using the **networkx** package.

• **graph_minimum_spanning_arborescence()**

This function represents the computation of the weight of the *Minimum Spanning Arboresence* on the provided matrices as noted in 4.10.1.1. However, considering that we are actually interested in using the edges with maximum weights, the graph edges needed to be inversed following the equation 4.18. Having the graphs transformed, the MSA is

computed on each one of them and, following the edges kept by the MSA algorithm, the final value is computed as the sum of the original, untransformed weights.

• **graph_shortest_path()**

This function has the purpose of computing the average length of the *shortest path* between any two nodes in the graph as specified in 4.10.1.3. It is important to note here that the algorithm is applied on the undirected version of the graph.

• **clustering()**

This function has been developed by my colleague and has the purpose of applying the clustering metrics on the graphs.

• **centrality()**

This function has also been developed by my colleague and has the goal of applying multiple centrality metrics on the graphs.

During the execution of the module, a metric dictionary is computed where, for each window and metric, the final value computed from the corresponding metric is saved. The structure of the dictionary and how it is used can be seen in 5.14.

```
# save results for clique metric
properties_dict[window][MAX_CLIQUE_LENGTH] = max_length
properties_dict[window][NUMBER_OF_MAX_CLIQUES] = number_of_cliques
```

Listing 5.14: Metric dictionary use.

At the end of the execution of the module, in the same output structure which was also used by the *Graph Visualization* module, for each trial, multiple folders are defined, one for each metric studied where different analyses with respect to the metric have been saved, such as the individual metric value on each graph. Moreover, the metric dictionary is also saved inside the trial folder for further use. This structure can be seen in figure 5.11.

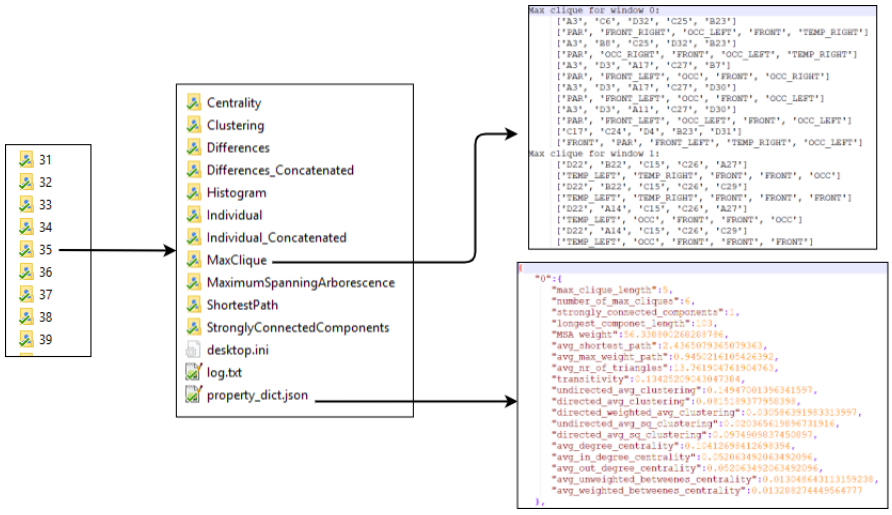


Figure 5.11: The output structure of the Graph Metrics module.

5.3.11 Graph Analysis

This module is the final component of the system. It's purpose is to analyse how the values of the metrics defined in *Graph Metrics* evolve in time with respect to different factors, such as trend or similarity.

The input of this module is represented by the results of the previous model. More concretely, for each trial, it needs the defined metric dictionary where, for each window, the value of each metric is defined.

The following functions have been defined:

• **graph_analysis()**

This function represents the wrapper function of the module. In order to use it, some parameters must be defined such as where are the metric dictionaries defined together with the output path to where the analysis should be saved and, finally. Moreover, a list of metrics to be analysed needs to be provided in case the user does not want to study all the metrics. After the function is called, a single central dictionary of metrics is defined where the dictionaries from all the trials are concatenated.

• **dtw_matching()**

In this function, the concept of the *Dynamic Time Warping* similarity function and warping path is implemented as specified in 4.10.2. More concretely, for each metric, the stimulus metric evolution is defined by concatenating the metric values from all the windows of the trials based on that stimulus in a time wise manner. This process can be viewed in 5.15.

```
# metric evolution for the first stimulus
for trial in TRIALS_FOR_STIMULUS[first_stimulus][1:]:
    for window in trial_dictionary[trial]:
        first_values.append(float(trial_dictionary[trial][window][metric])
    )
        first_text.append(f'{trial}_{window}')

# metric evolution for the second stimulus
for trial in TRIALS_FOR_STIMULUS[second_stimulus][1:]:
    for window in trial_dictionary[trial]:
        second_values.append(float(trial_dictionary[trial][window][metric
    ]))
        second_text.append(f'{trial}_{window}')
```

Listing 5.15: Stimulus metric evolution.

After defining the evolution of a metric for each stimulus, the stimuli are compared in pairs of two by applying the *Dynamic Time Warping* algorithm on them. This step was actually done by using the **DTW-Python** package. Finally, after applying the algorithm, the resulting warping path is plotted and the similarity measure is saved.

• **trend_analysis()**

This function has been developed by my colleague and serves the purpose of making an analysis of the metric evolution in time during the experiment with respect to the trend.

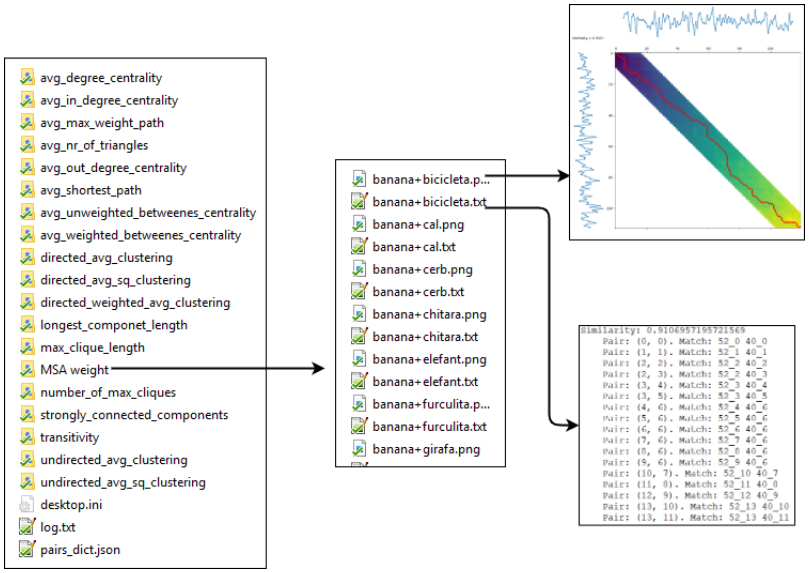


Figure 5.12: The output structure of the Graph Analysis module.

At the end of the execution of the module, a complex file structure is created at the specified output path. Inside this structure, for each metric, a folder is defined where the DTW analysis for the corresponding metric is saved. Inside a metric folder, for each possible pair of stimuli, two files are defined, one representing the plot of the matching cost matrix, and the other representing the detailed warping path where, for each match, the trials and windows making up the match are presented alongisde the overall similarity measure score.

5.4 Graphical User Interface

Considering the fact that the application consists of multiple modules, each with its own set of parameters that need to be defined, we have made the decision of implementing a *Graphical User Interface* that hides the complexity of the solution and allows the user to easily interact with the application.

For this, we have made use of the **tkinter** package from python. We have created a simple window, having multiple tabs, with one tab for each of the modules. Inside each tab, the user can specify the running parameters in the provided fields and can start the execution of the corresponding module. Because some executions take quite a long time, we have also integrated, inside each tab, a textarea where the progress of the application is displayed for the user to see. More concretely, during the execution of the module, at each intermediary state it passes through, it notifies the user using a **widget** object that gives it access to the GUI application.

As one last feature integrated, in order to avoid issues regarding incompatibilities

between datatypes at running time, we have actually integrated a validation component inside the GUI module. More specifically, for each possible type of data which a module could receive, we have defined multiple validator functions that test if the corresponding string provided is valid or not. These functions are the following ones:

- Check if the provided string is an integer or not. In order to make this check, we have made use of the python built-in exception mechanism. More concretely, we try to convert the string to an integer and, if it fails, we catch the thrown exception and invalidate the corresponding string. This function can be observed in 5.16.

```
def check_if_int(value):
    # try to convert value
    try:
        int(value)
        return True
    # if the value is not an integer
    except ValueError:
        return False
```

Listing 5.16: Check integer validation.

- Check if the provided string is a float or not. This function works exactly as the function displayed in 5.16, with the difference that it makes the conversion to **float**.
- Check if the provided string represents a boolean value or not. A boolean value is defined as **T** for true and **F** for false. In order to make the validation, a simple equality check sufficed.
- Check if the provided string represents a valid path. Considering that, for each module, multiple paths need to be specified, we need to make sure that the paths we provide to the modules are actually valid and they exist. For validating this fact, we have made use of the **os** package from python which has built-in functions for these types of checks.

Having the GUI module defined, a user can easily interact with our application by specifying different running parameters as he sees fit, executing different modules and being notified by their progress. Moreover, the GUI also has the ability of running in parallel different modules, since each execution takes place on a different thread created by the GUI main thread. Therefore, the sequence diagram of the interaction between the user and a generic module of the application facilitated through the GUI interface can be viewed in figure 5.13.

5.5 Command Line Interface

Because of the nature of both the solution and the data, we faced the problem that our own computers were not powerful enough to execute the *Recurrent Graph WaveNet*

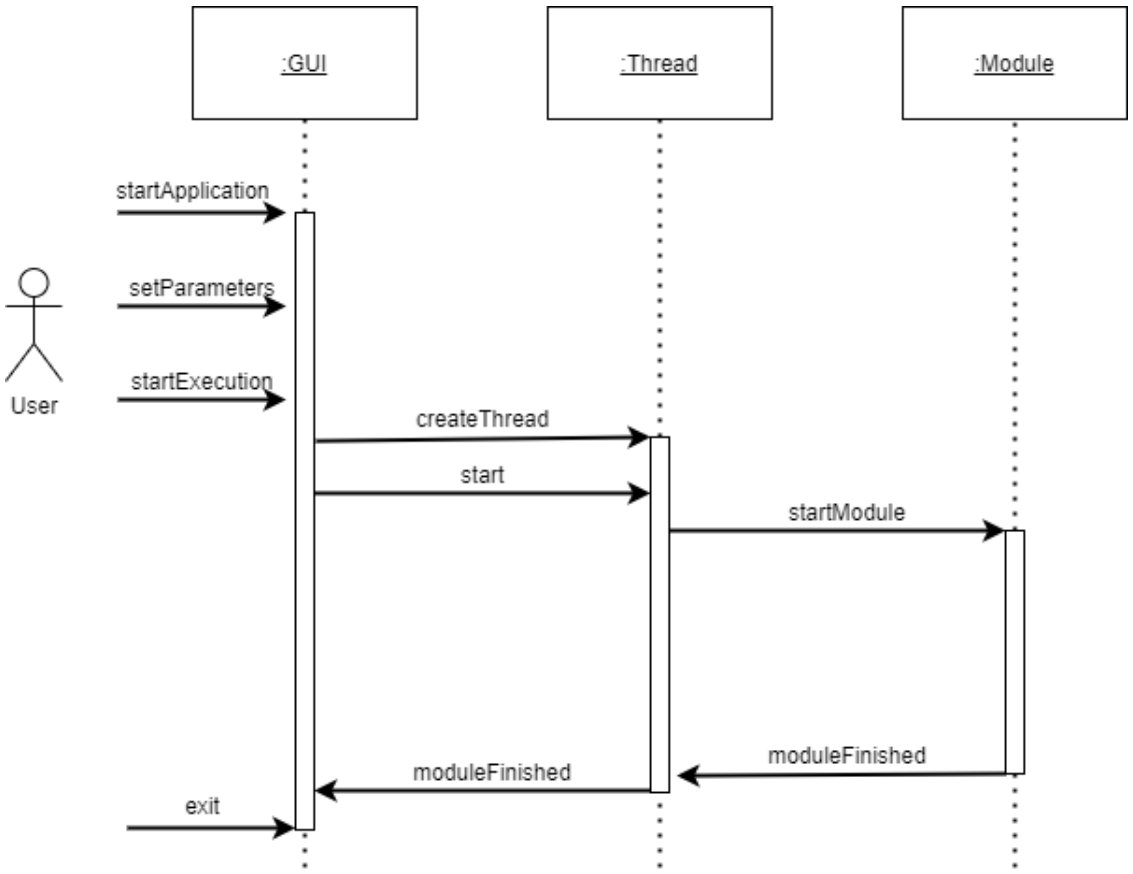


Figure 5.13: Sequence diagram of the GUI user interaction.

module in a feasible amount of time. Because of this, we have decided to also deploy our solution on the **Google Cloud Platform**.

More concretely, what we have actually done was to create a Virtual Machine on the GCP where we could execute our solution. In order for the machine to have access to the code, we have connected the VM to our own private **Github** repository where the code of the solution is saved. Moreover, in order to enable the access to the machine, what we have done was to set up a **SSH** connection between our computers and the virtual machine by creating a secure channel of communication with which we were able to access the machine using the command line. Having this connection, we were able to remotely execute our solution.

Finally, in order to have access to the file structure, we have also set up a **SFTP** connection using the pre-established **SSH** connection. Using the SFTP connection, we were able to upload and download data to and from the VM.

However, since we interacted with the machine only from the command line, we could not use the GUI module we have defined. Therefore, in order to facilitate an interface to our solution for scenarios when the GUI can not be used, we have also developed

a command line interface from which the entire functionality of all the modules of the application can be accessed. For using it, a user needs to call a special python script with the name of the module and the parameters of the module as shown in 5.17.

```
./main_cmd.py module_name [param_list]
```

Listing 5.17: Command Line Interface calling.

If one is not sure how to call a certain module, we have also included an additional option to the CMD module. By calling it with the argument **"help"**, a user manual will be printed where, for each module, each parameter is described. This option can be seen in figure 5.14.

```
If option is: DatasetStatistics
     1. Path to the Dots folders
     2. The output path

If option is: RawDataFilter
     1. Path to the Dots folders
     2. The output path
     3. The degree of parallelism
     4. Trial filter length - a choice would be 627
```

Figure 5.14: Result of running the **help** command.

Chapter 6

Testing and Validation

In this chapter, the results of the solution are presented. The sections inside this chapter follow the same logical structure as the one presented in the conceptual architecture that can be seen in figure 4.1, with the focus only on the meaningful results that have been generated.

6.1 Dual Ensemble Classifier Optimal Hyperparameters

6.1.1 DEC Performance Tuning

As specified in 4.5.2, the first step in choosing the DEC model was to perform an initial tuning with respect to a large set of hyperparameter choices. More concretely, for each of the two possible hyperparameters, namely *window size* and *window offset*, the following values have been studied:

- Window size: 100, 125, 150, 175, 200, 225, 250, 275, 300
- Window offset: 50, 75, 100

Having the values of the hyperparameters defined, the total number of possible sets of hyperparameters is equal to the cartesian product of these two sets, equal to 27. After training each model, based on the confusion matrix for each of the two classification problems solved, the average f1-score was computed over all the possible classes, resulting in a set of two values. These values specify the performance of the model with respect to the classification problem being solved.

The distribution of the performances of these 27 models can be viewed in figure 6.1. As it can be seen from the figure, the performance of the DEC model varies with respect to the choice of hyper-parameters. If we were to look only at the maximum values of the f1-score on each of the two problems, it can be seen that, for the stimulus problem, the

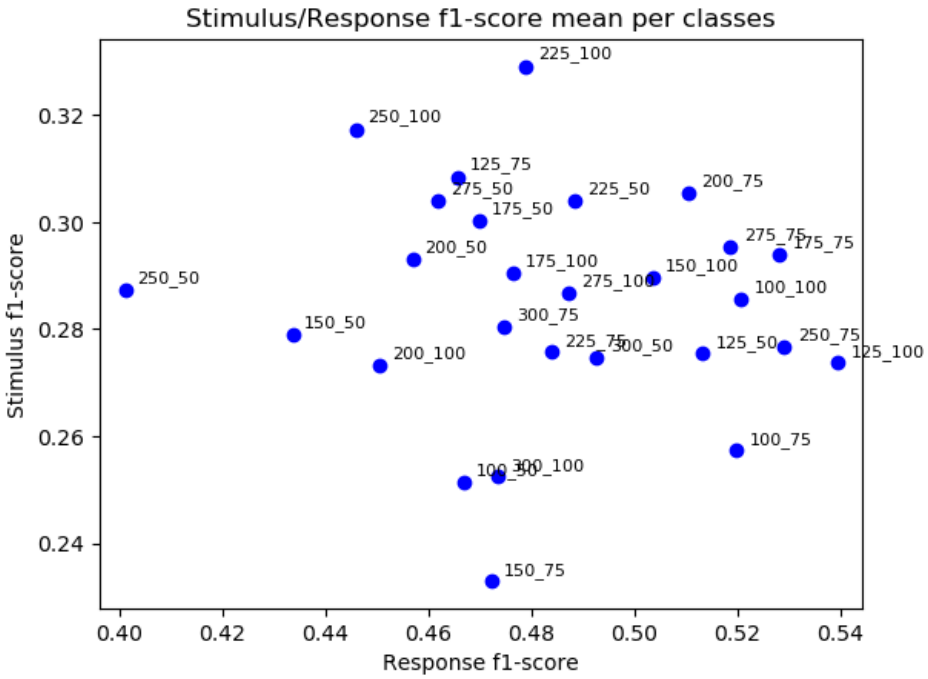


Figure 6.1: Initial distribution performance of the DEC model.

best choice would be **225_100**, whilst for the response problem, the best choice would be **125_100**.

However, the problem with these two choices is that, even if they are the best on one of the problems, they perform rather badly on the other problem compared to the best option from there. For example, on stimulus, there is a difference of 0.04 and, on the response problem, there is actually a difference of 0.06. It is important to take into consideration that we are looking for a choice of hyperparameters that is capable of making the DEC model perform at its best on both classification problems. Therefore, in order to ease the analysis of the hyperparameters, a filtering has been done onto the plot from figure 6.1, where only the models that had the performance above the average on **both** classification problems are kept.

The result of the filtering can be seen in figure 6.2. As it can be seen from the figure, out of the 27 initial models, only 7 are kept after the filtering. Now, following the same reasoning of choice that was applied on the initial plot, if we were to choose the best two models with respect to the classification problems, we would choose **200_75** and **175_75**. Studying the performance of these two models relative to each other, we can see that the problem previously encountered was almost solved, with the two models having a difference of only 0.02 between the performances on both classification problems.

However, we still do not have a model that performs the best on both classification problems at once. The only model that is close to achieving that type of performance is

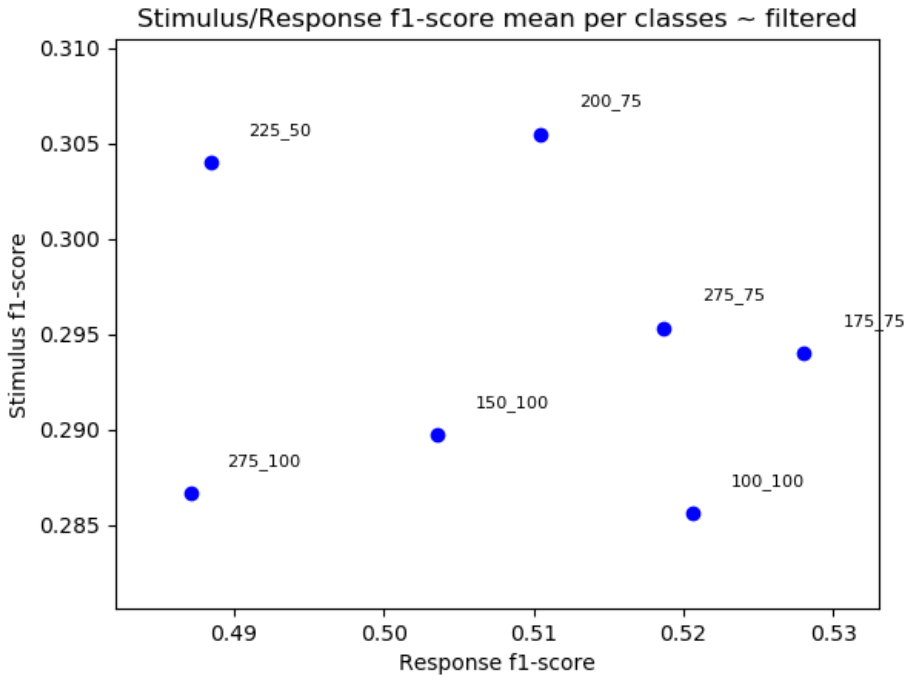


Figure 6.2: Filtered distribution performance of the DEC model.

the **275_75**, which performs rather in the middle between the previous two choices.

Taking everything into consideration, only this analysis is not enough in order to make a pragmatic choice of optimal hyperparameters. There are 3 viable options here, namely **200_75**, **175_75** and **275_75**, each with a different performance that is the best with respect to a certain perspective. In order to make a concrete choice between the three models, my colleague [12] has implemented another type of analysis that is capable of choosing the best set of optimal hyperparameters starting from these 3 options presented here.

His analysis concluded that the best choice of hyperparameters that make the DEC model perform at its best are the following values:

- A window size of 275 milliseconds.
- A window offset of 75 milliseconds.

6.1.2 Optimal Hyperparameters Interpretation

From a Neuroscience perspective, there are 4 main brainwaves that are recorded and studied using an EEG headset. The brainwaves are the following ones [21]:

- **Delta**: The Delta brainwave has a frequency of 3Hz or under, with a period of around 333 milliseconds or higher. It usually occurs during deep sleep.

- **Theta**: The Theta brainwave has a frequency of 3.5 to 7.5Hz, with a period between 133 and 285 milliseconds. It usually occurs during sleep or when in deep meditation, such as learning.

- **Alpha**: The Alpha brainwave has a frequency of 7.5 to 13Hz, with a period between 77 and 133 milliseconds. It usually occurs during mundane activities that do not require a lot of information processing.

- **Beta**: The Beta brainwave has a frequency of 14Hz or above, with a period below 71 milliseconds. It usually occurs during activities where a lot of information processing is needed, especially during attention is needed during cognitive tasks.

Considering the context of the experiment, it is safe to assume that no **Delta** brainwaves would be generated during the experiment. However, regarding the other 3 brainwaves, they might be encountered during the experiment, with the **Beta** brainwave being the most predominant one since the activity that the subjects are exposed to is actually image recognition, which is a cognitive task.

What is interesting about the discovered division hyperparameters is the fact that, for a window of size 275 milliseconds, the information that is capable to be captured inside this time frame may contain information from all the 3 brainwaves. More concretely, looking at each wave's period, we can see that, for this choice of window size, we could capture 1 full Theta cycle, $2-3$ full Alpha cycles and, most notably, 4 or more full Beta cycles.

Moreover, with respect to the window offset, a correlation between its value and the **Beta** brainwave can be discovered. More specifically, our trial segmentation process uses a window offset of 75 milliseconds, which means that the next window will be shifted in time with this amount. What is interesting here is that this value is also almost equal to the upperbound of the period of the **Beta** wave, suggesting that the window division process follows the periodicity of this wave. However, correlation does not imply causation and this fact should be further studied in order to prove its validity.

Nevertheless, as it can be seen from this analysis, our choice of optimal hyperparameters allows the capturing of multiple types of brainwaves during the timeframe chosen, most notably a large number of **Beta** waves.

6.2 Dual Ensemble Classifier Analysis

6.2.1 DEC Classification Performance

Based on the choice of optimal hyperparameters discovered, in this subsection the performance of the classifier will be analysed. For this, we have chosen the model with the best average f1-score on both classification problems from the pool of multiple runs done on this set of parameters.

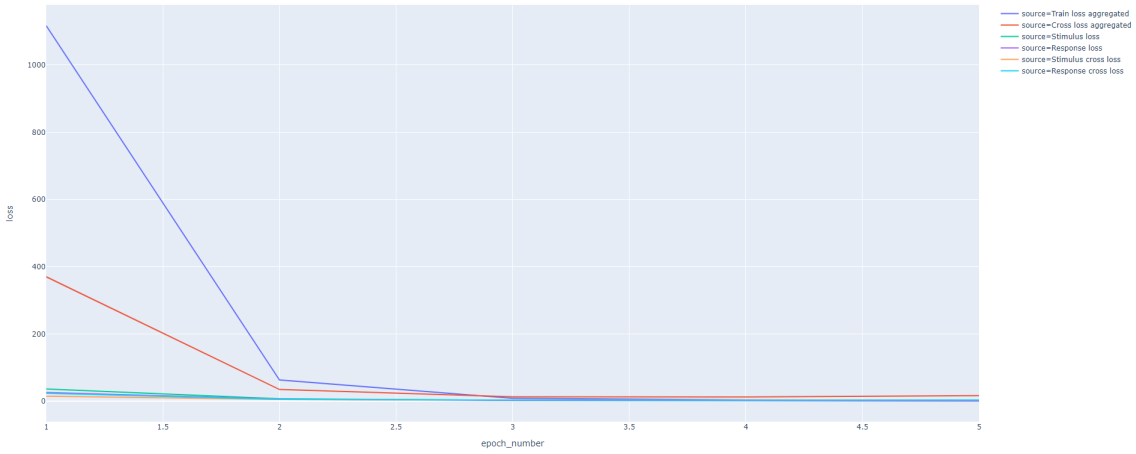


Figure 6.3: The learning curves of the DEC model.

First of all, the learning curves of the model can be seen in 6.3. There are two sets of curves that are to be followed here: the train and cross-validation loss on the DEC model as a whole, and the train and cross-validation loss on each of the two classification problems being solved. As it can be seen from the figure, the train and cross-validation loss of the DEC model as a whole decrease together, with the model not overfitting during training.

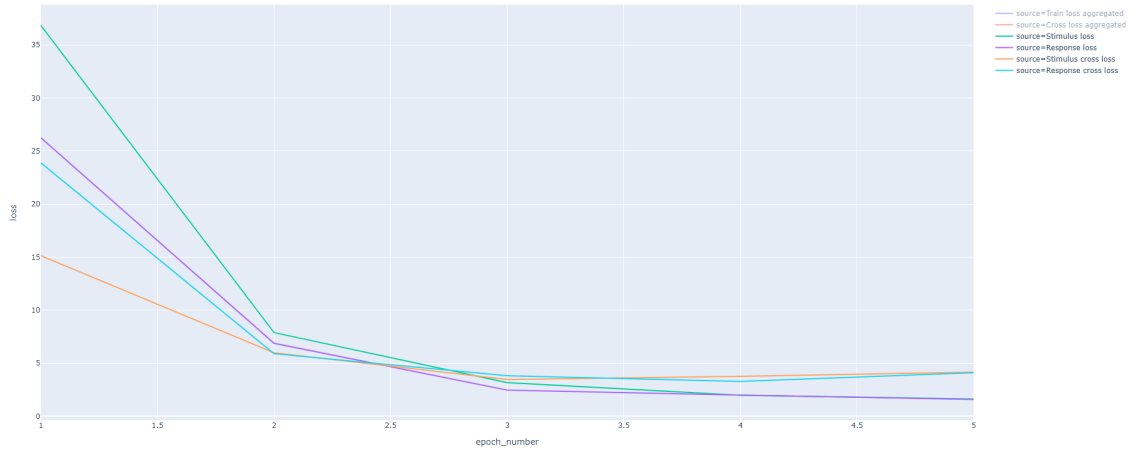


Figure 6.4: The learning curves of the two classification problems.

Regarding the other set of curves, a closer inspection is needed in order to actually see them. For this, in the figure 6.4, only this set of curves is displayed. As it can be seen from the figure, both models are optimized together, with neither of them being stuck in a non-optimal local minima. Moreover, it can be observed that the difference between the two losses on the two models decreases as the training takes place. Both of these facts

apply to both the training set and the cross-validation set, and therefore is a proof that the **Dual Aggregation Loss** defined in 4.2 actually works.

Table 6.1: Response classification performance.

Class label	Precision	Recall	F1-Score
seen	0.71	0.63	0.67
uncertain	0.39	0.33	0.36
not seen	0.52	0.65	0.58

Table 6.2: Stimulus classification performance.

Class label	Precision	Recall	F1-Score
0.05	0.33	0.43	0.38
0.1	0.41	0.24	0.30
0.15	0.26	0.40	0.32
0.2	0.31	0.22	0.25
0.25	0.31	0.48	0.38
0.3	0.40	0.16	0.23

Moreover, in tables 6.1 and 6.2, the classification performance on each of the two classification problems can be seen. Regarding the stimulus classification problem, the classifier can be seen that it does not perform very well. With an average f1-score of 0.31 and an accuracy of 32%, it can be seen that the classifier has quite some problems distinguishing between the 6 classes. Unfortunately, we are not aware of any other approach that tried to classify this experiment using the stimulus label and, therefore, we can not make a pragmatic comparison and have an answer to the question *"does this model perform well or worse than another model?"*. All we can say here is that the problem that might stop the stimulus classifier to perform better is the choice of the stimulus labels. What is meant by this is the fact that, since the task of each trial is the same, namely image recognition, it might be possible that, from the way we have built the examples, the factor that influences how much is the image visible could not be present amongst them. Our intuition dictates that maybe some feature enhancing could help the model improve better, like introducing as a feature the length of the trial.

However, regarding the performance on the response classification problem, thing are looking a lot better. More concretely, as seen in table 6.1, the classifier performed quite well, having an average f1-score of 0.53 and an accuracy of 58%. Moreover, as it can be seen from the table, the most problematic class is the *uncertain* one, which proves to be hard to actually classify.

Comparing the response classifier to the other approaches available, we first need to specify that the other classifiers have actually been binary classifier, ignoring completely the *uncertain* class and keeping only the other two classes. Moreover, in some cases, the

other studied classifiers may have also performed the training either on one subject, or only on one channel. However, our DEC model actually makes a multi-class classification, learning on all the subjects and all the channels available.

Considering all of these facts, looking at the performance of the DEC model, it is clear to see this model can actually be a worthy adversary of the other models, having a comparable performance with the other classifiers even in the most possible general case which has not been covered previously.

6.2.2 DEC Classification statistics

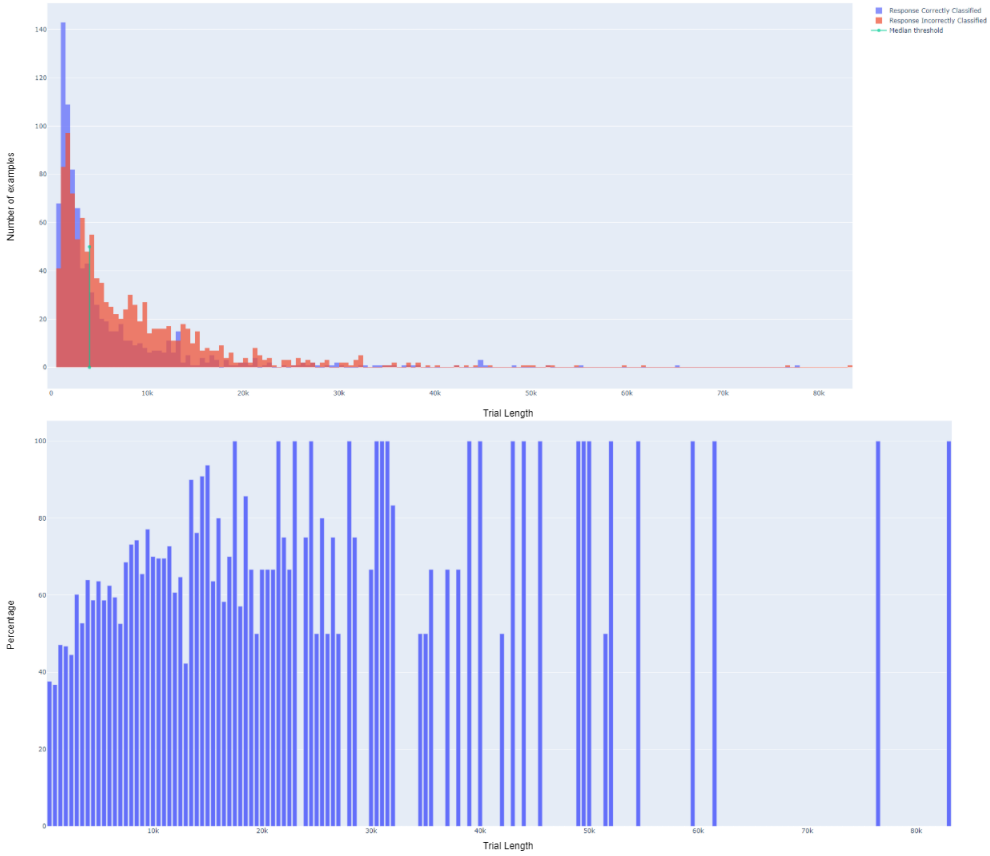


Figure 6.5: Number of examples correctly and incorrectly classified (top). The misclassification accuracy (bottom).

As an addition to the previous subsection, another analysis can be made on the best DEC model trained on the set of optimal hyperparameters. More concretely, as specified in 4.6, the misclassification accuracy of the DEC model is plotted with respect to the trial's length under the form of a histogram, as it can be seen in figure 6.5. However, only

the response classifier is studied because, for the stimulus one, because of its results, no relevant information was extracted.

As it can be seen from the plot on the top, what can be observed is that, for the trials that are under 3 seconds, the number of examples correctly classified (blue) is far larger than the number of examples incorrectly classified (red). However, after the mark of 3 seconds, the number of examples incorrectly classified is larger than the one of the correctly classified. Moreover, from the plot, it is easily notable that there is a long tail distribution of the trial's lengths, having quite a significant number of examples in each bin until the mark of 20 seconds. However, after the mark of 20 seconds, only outliers are present until the end of the distribution. Taking these two into consideration, the observation that could be extracted from here is that, the larger the trial's length is, the larger is the misclassification accuracy.

However, because of the distribution from the plot on top, this observation might actually be misleading. Therefore, the second plot from the figure has been created, having the actual misclassification accuracy plotted for each bin. From this figure, it can be seen that our observation is actually valid. Until the mark of 20 seconds, with each bin representing trials that are larger and larger, the misclassification accuracy keeps getting worse as long as the trial increases. However, after the mark of 20 seconds, the plot can no longer be interpreted properly because of the outliers present in the distribution of the trial's length.

Nevertheless, the observation that can be extracted from here is that one of the limitation factors of the DEC model is the fact that trials have different lengths and this feature actually impacts the performance of the model. Considering that, based on the trial segmentation process presented in 4.4.2, all the examples have the same amount of information from the trial, it follows that, for the trials incorrectly classified, the problem might be the fact that, after filtering, the actual meaningful information from the trial is removed.

With this observation in mind, as specified in 4.7.3, we have decided to increase the threshold filter value for a trial from 1254 milliseconds all the way up to 2000 milliseconds in the hope that the discriminatory information from the trial would be included.

6.3 Recurrent Graph WaveNet Optimal Hyperparameters

6.3.1 RGW Performance Tuning

Before generating the correlation graphs from the windows, we first need to identify which are the optimal hyperparameters for the RGW model. More concretely, the following values of hyperparameters are defined:

- Blocks and layers: (2,4) and (10,2). This parameters are specified together because

they are dependent on eachother.

- Use functional networks: True or False.
- Use learnt matrix from previous window: True or False.

Because we need to use the learnt matrix during the validation and we do not posses any, there actually needs to be 2 different validations, with the first one not using the learnt matrix. Therefore, in the table 6.3, the first validation can be seen, which was done on the first window from trial 210.

Table 6.3: RGW performance on the first window of trial 210.

Blocks	Layers	Functional Network	Previous Learnt Matrix	MAE	MASE
2	4	False	False	0.0354	0.253
2	4	True	False	0.033	0.244
10	2	False	False	0.024	0.171
10	**2**	**True**	**False**	**0.023**	**0.169**

As it can be observed from the table, the best choice of hyperparameters is when the architecture is as large as possible and the functional network of the corresponding trial is used. The results of the validation prove that, for both the **MAE** and **MASE** functions, this choice of parameters is the best.

Therefore, in order to make the second validation, the learnt matrix from the best model on the first validation has been used. It is important to note here that, because we need to use this matrix, the validation was not performed on the first window of trial 210, but rather on the second one. Moreover, the observation regarding the large architecture was also kept, therefore the second validation using only the **(10,2)** configuration for the number of blocks and layers.

Table 6.4: RGW performance on the second window of trial 210.

Blocks	Layers	Functional Network	Previous Learnt Matrix	MAE	MASE
10	2	False	False	0.032	0.246
10	2	True	False	0.041	0.322
10	2	False	True	0.037	0.297
10	**2**	**True**	**True**	**0.028**	**0.222**

In the table 6.4, the validation on the second window from the trial 210 can be observed. In this validation, the hyperparameter *Previous Learnt Matrix* was also tested.

As it can be seen from the table, the best choice of hyperparameters is the one where both the functional network and the learnt matrix from the previous window are used with the model having this configuration presenting the best performance on both cost functions.

Therefore, the choice of optimal hyperparameters is rather straightforward. The Graph WaveNet modules will have a large architecture consisting of **10** blocks, **2** layers and using as an input specification of the dependencies in the graph two matrices: the **functional network** of the corresponding trial together with the **learnt matrix** from the previous window.

Moreover, this choice of hyperparameters is backed up not only by the original authors of the Graph WaveNet paper [9], but also by the authors that improved the Graph WaveNet module to reach state-of-the-art performance [17]. More concretely, their observation is that, based on their tests, the best performance that can be reached with the Graph WaveNet model is when:

- The architecture is as large and deep as possible (hence the number of blocks).
- The model has access to some pre-established structure of the graph (hence the two matrices used as an input).

6.3.2 RGW Performance Analysis

The learning curves of the best trained RGW model can be seen in figure 6.6, with the red line representing the loss on the cross-validation set and the blue line representing the loss on the train set.

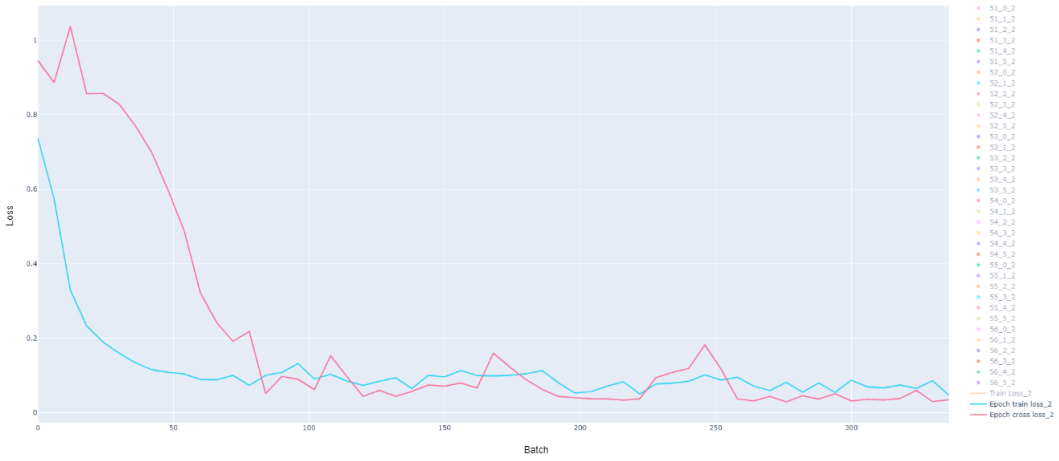


Figure 6.6: RGW Learning Curves.

As it can be seen from the figure, the model actually manages to reach a performance on the cross-validation set that is as good as the one on the train set which indicated that the model has trained properly, with neither overfitting or underfitting occuring. Moreover,

because of the early stopping mechanism, the training stops after 336 batches, meaning that the model has not improved his performance on the cross-validation set anymore. This stopping value is rather constant with respect to multiple runs done on the same set of hyperparameters and, therefore, when the full training will occur as specified in 4.8.3, the training will be stopped at 300 batches.

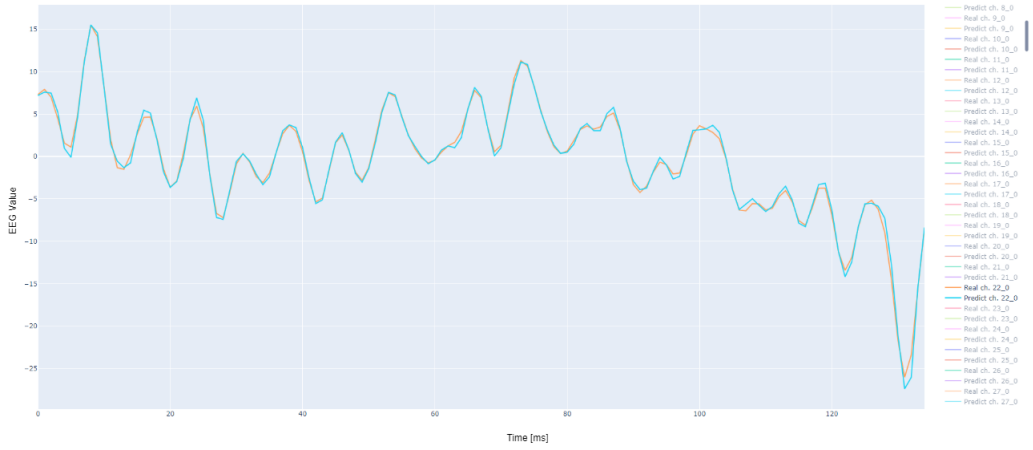


Figure 6.7: RGW one step-ahead prediction on channel A23.

Moreover, in figure 6.7, the one step-ahead prediction of the RGW model on channel A23 from the cross-validation set is displayed. In the figure, the orange line represents the true value of the signal, whilst the light blue line represents the predicted value of the signal. As it can be observed, the model manages to make quite an accurate one step-ahead prediction of the evolution of the signal, managing to respect almost all the patterns encountered, following almost all the time the trend and turns of the signal. This fact is also proved in table 6.4, where the MASE score displayed indicates that this model has a prediction net superior than the one of the naive forecast.

6.4 Graph Analysis

6.4.1 Graph Visualization

As specified in the 4.9, before being able to look at the graphs, we first need to preprocess and aggregate the nodes with respect to the brain regions, as it can be seen in figure 4.17.

The result of the aggregation can be seen in figure 6.8 where the second window of the trial 207 is represented. The trial from which the window was extracted was a *seen* trial. As it can be observed from the figure, there are a total of 9 nodes corresponding to the brain regions, with edges going both to and from a node. Moreover, not only are the width and the color of the edges proportional to the value of the weight, but also the node

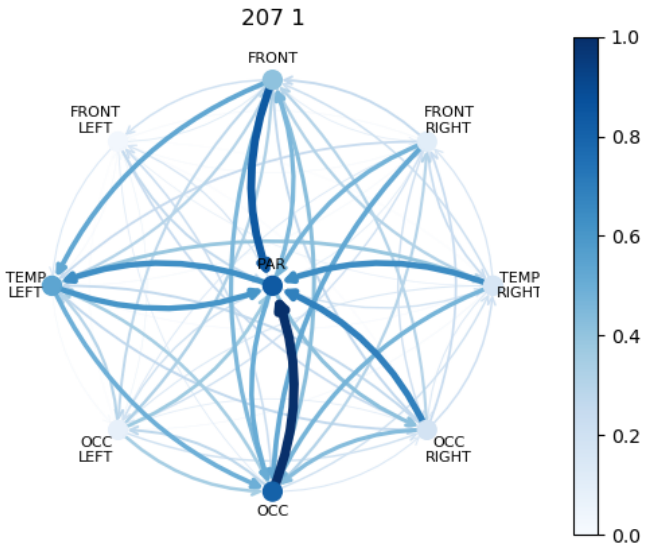


Figure 6.8: The second window visualization from the trial 207.

weight is represented with the color placed on it. The weight values can be figured out by looking at the colormap displayed on the right of the figure.

However, if we want to actually make an analysis based on the evolution of these graph snapshots in time for a trial, we actually need to look at the graphs from all the windows in the trial. With this in mind, the graphs from trial 207 are displayed in figure 6.9, with the windows being placed in the picture in a **row-major order**, having the first window laying in the top left corner.

Having the evolution displayed in this manner, we can actually see the entire brain state and how it evolves during the image recognition task. In this context, in the following paragraphs, our observations regarding this evolution will be presented with respect to the structure and functionality of the brain. These observations are the result of the visual analysis on multiple trials and the correlation between these observations and the brain's known functionality.

The first thing to note here is that, when looking at the evolution over a trial, what is always a constant fact that occurs is the fact that these networks of correlations actually **fluctuate**. More concretely, what is meant by this is the fact the network keeps reconfiguring itself, with new connections appearing and old connections being removed, fluctuating between states where a high correlation is present to states where little correlation is present at all.

After discussing this observation with the researchers from the TINS institute, their intuition is that this fluctuation is actually a result of the image recognition process. Since a subject needs to figure out what is presented in the image, he constantly tries to see

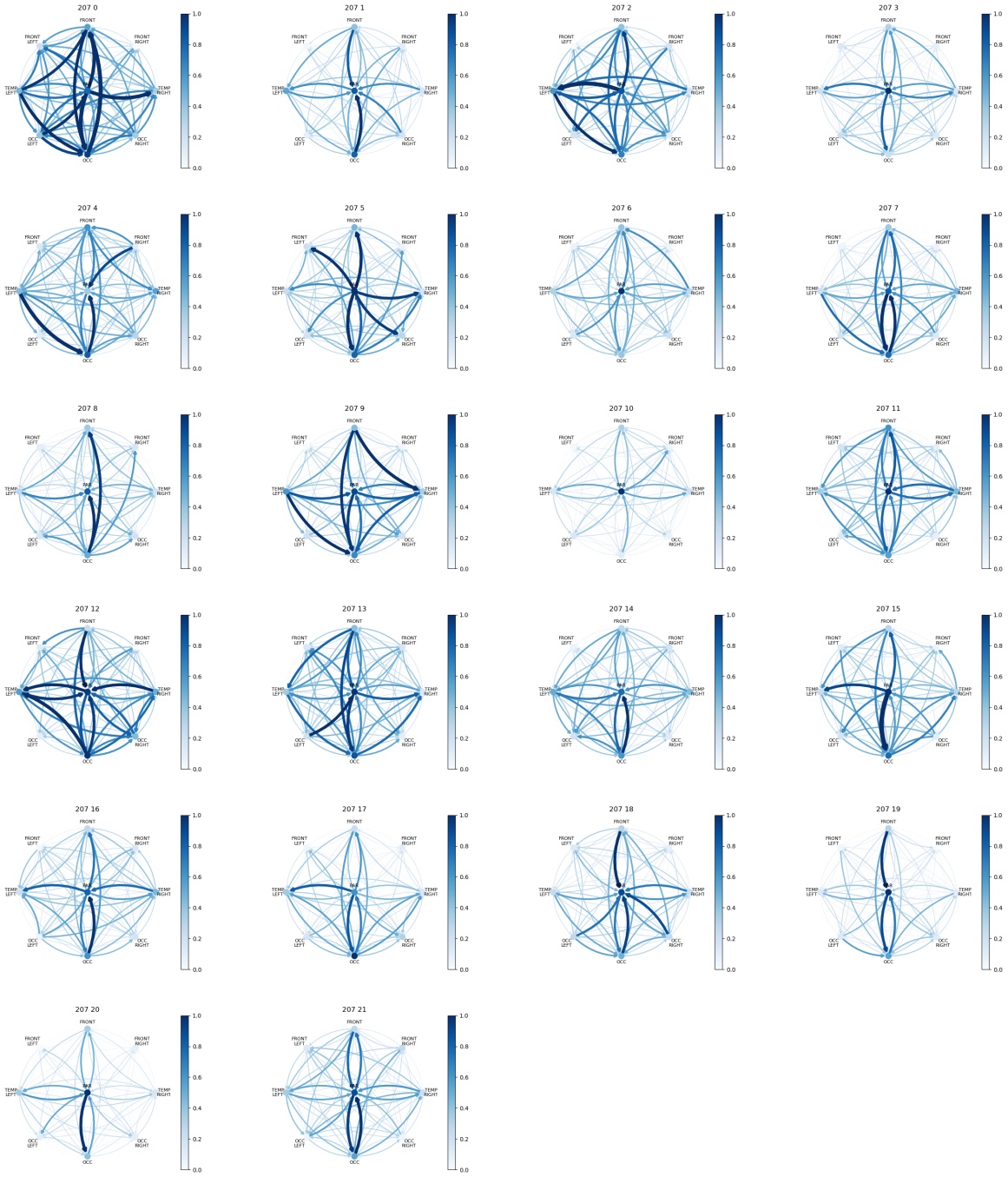


Figure 6.9: The evolution of the graphs for trial 207 in a row-major order.

patterns and identify the depicted object. Moreover, most of them are not capable of identifying the object instantly. Therefore, they keep trying and failing until they either

give up or they figure out what the stimulus is. Considering this, the fluctuation of the network can actually be a result of this *try and fail* mechanism, with the network reconfiguring itself into a highly connected graph for each new try.

The next interesting fact that can be observed from these networks is that, almost in all of them, the occipital region or its 2 neighbours are present as being highly connected either inside the region or connected to another region. Considering that the EEG data was recorded for subjects that had the task of cognitive recognition, it is rather obvious that the occipital lobe (which is the center of the visual processing component of the brain) would play a major role during the trial.

Moreover, in the same context, the previous pattern can also be observed for the frontal region and its 2 neighbours, which are also present rather often as being highly connected. With respect to the experiment's task, this pattern can also be explained by taking into consideration the fact that a subject was not actually capable of instantly figuring out what the depicted object was. Therefore, he needed to actually make an effort to analyse the photo and try to see if he could identify the stimulus. For tasks like this where a high level of attention and problem-solving skills are needed, the frontal lobe plays the most important role. Considering this, it is rather safe to assume that this is the reason why the frontal lobe is highly connected with the other brain regions.

Furthermore, the next observation that can be extracted from the graphs is the fact that there appears to be quite a large correlation between the temporal lobes and the occipital lobe. This correlation between these regions can actually be justified by taking into consideration two important aspects: the visual stimulus and the memory of the subject. Considering that the subject is not unfamiliar with the stimulus (they represent object from everyday life), its shape should be familiar to the subject with respect to his long-term memory and he should not have any problem identifying the object under normal circumstances. The second aspect that needs to be considered is the short-memory of the subject. Since the trials are one after the other, as the experiment progresses, the subject not only get accustomed to the experiment's layout, but also remembers the shape of the stimuli presented until then. Considering there are 30 stimulus at 7 different deformation levels, a subject is exposed to the same stimuli a total of 7 times and, after identifying correctly the object for the first time, it has no problem identifying it from that moment on.

Taking these two aspects into consideration, the correlation links between the temporal lobes and the occipital lobe can be justified by functions that the temporal lobe performs, specifically the aid of making sense of complex visual information such as abstract shapes, remembering people's faces or even figuring out what stimulus is presented.

Finally, the last observation we can make here is actually the most interesting one. Over most of the trials, one consistent pattern appears that is present in the same exact moment for each trial. More concretely, what can be observed is that, usually, the last 2 windows from the trial have rather less overall correlations, but the edges or weights that are kept usually involve both the parietal and frontal lobe.

This observation is interesting because of the way a trial ended. In order for the

subject to give his response, he needed to manually press a button from a keyboard. For a *not seen* trial, he needed to press the letter **A**. For an *uncertain trial*, he needed to press letter **S**. However, for a *seen* trial, he needed to press **L**. The interesting thing about the possible answers is the hand positioning on the keyboard. These keys have been chosen in order for the subject to have a positioning that he was not familiar with. Therefore, when giving the response to a trial, he can not actually press the key instantly. He firstly needs to remember which key is which and only then answer. This type of behavior usually appears in humans that are given a task which, for example, they are accustomed to do with their right hand, but they are asked to do it with their left hand. Because they are put in a situation in which they are not familiar, a clumsy behavior appears that the person can fix only by focusing a lot of attention in performing the task.

What is interesting about this behavior is that its actual seenable in our results. First of all, the parietal lobe's presence as almost the solely important node here suggests a lot of activity inside the region. Considering that the purpose of this lobe is to actually control the motor functions of the body, it could be correlated with the subject wanting to press the button. Moreover, since the subject makes an extra effort to correctly press the key that indicates his response, he could also make use of the frontal lobe in order to figure out which is the keystroke he needs to do.

Multiple observations of these kind can be extracted from the data, justifying seenable patterns by correlating them either to the experiment, the human behavior or the purpose of each of the brain lobes. However, correlation does not imply causation and these results and observations should be further studied in order to prove their validity.

6.4.2 Dynamic Time Warping matching

As specified in 4.10.2, an analysis has been made on these graphs with respect to certain statistical properties they might have. A set of metrics have been defined by me and my colleague as follows:

- Maximum Spanning Arboresence weight.
- The size of the maximum clique.
- The average length of the shortest path between any two nodes.
- The average unweighted betweenness centrality.
- The average weighted betweenness centrality.
- The average number of triangles.
- The transitivity measure.
- The directed average clustering score.

After computing the metrics on each graph, what was actually analysed was how similar were the evolution of the values of a metric for one stimulus with respect to another stimulus. Considering that, for each stimulus, 6 trials from the dataset we have used were based on it, in order to see the evolution on a stimulus, all the windows from these trials needed to be concatenated with respect to the time they occurred.

In this context, the DTW matching algorithm was applied on each pair of stimulus for each metric to see how similar the metric evolution was between the two. In figure 6.10, the result for the DTW matching applied on the MSA weight evolution for the *banana* and *elephant* stimulus is presented.

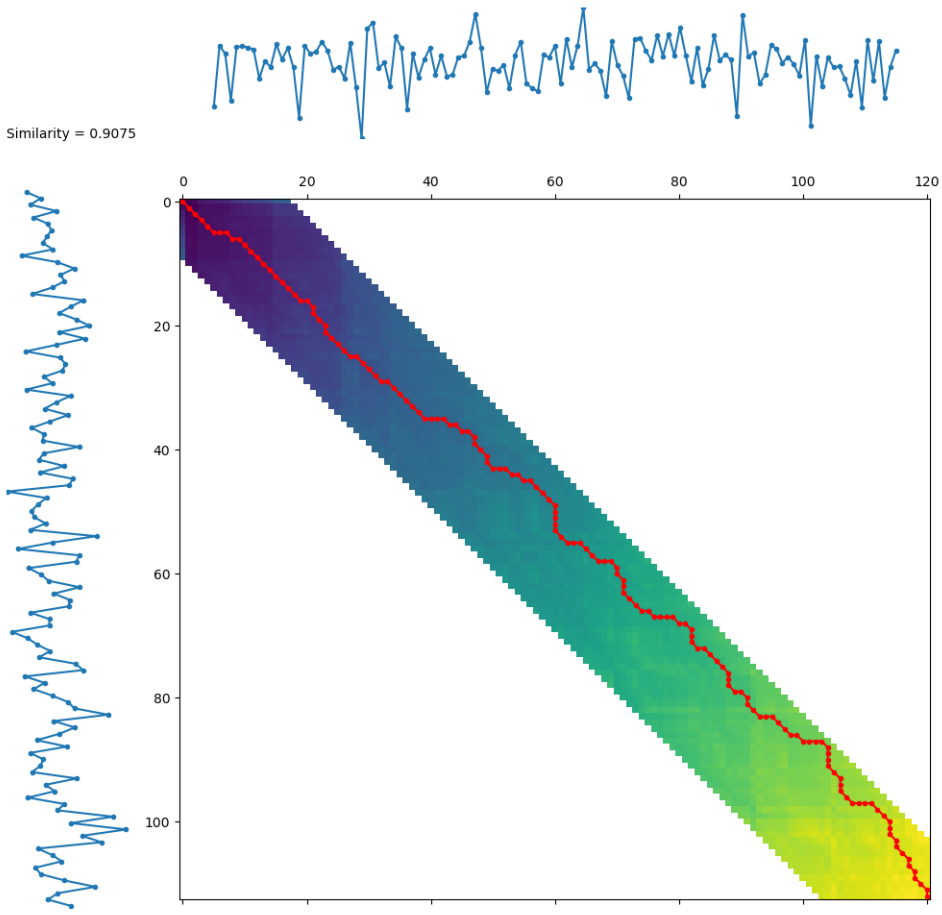


Figure 6.10: DTW Matching on the MSA weight metric for the banana and elephant stimuli with banana being on the vertical axis and the elephant on the horizontal axis.

The result of the DTW matching is actually a plot, containing the warping path between the two stimuli. The algorithm also had a locality constraint, imposing that the warping path should be generated with an offset of maximum ± 10 windows. This

constraint implies that, if the path were to not be optimal in this radius, then the path would be displayed as close to the optimum as possible, meaning that it would actually be exactly on the edge of the constraint band which is colored in the picture. Moreover, the similarity measure is also computed and displayed in the upper left corner of the photo, with a value of 1 representing identical and a value of 0 representing completely unrelated.

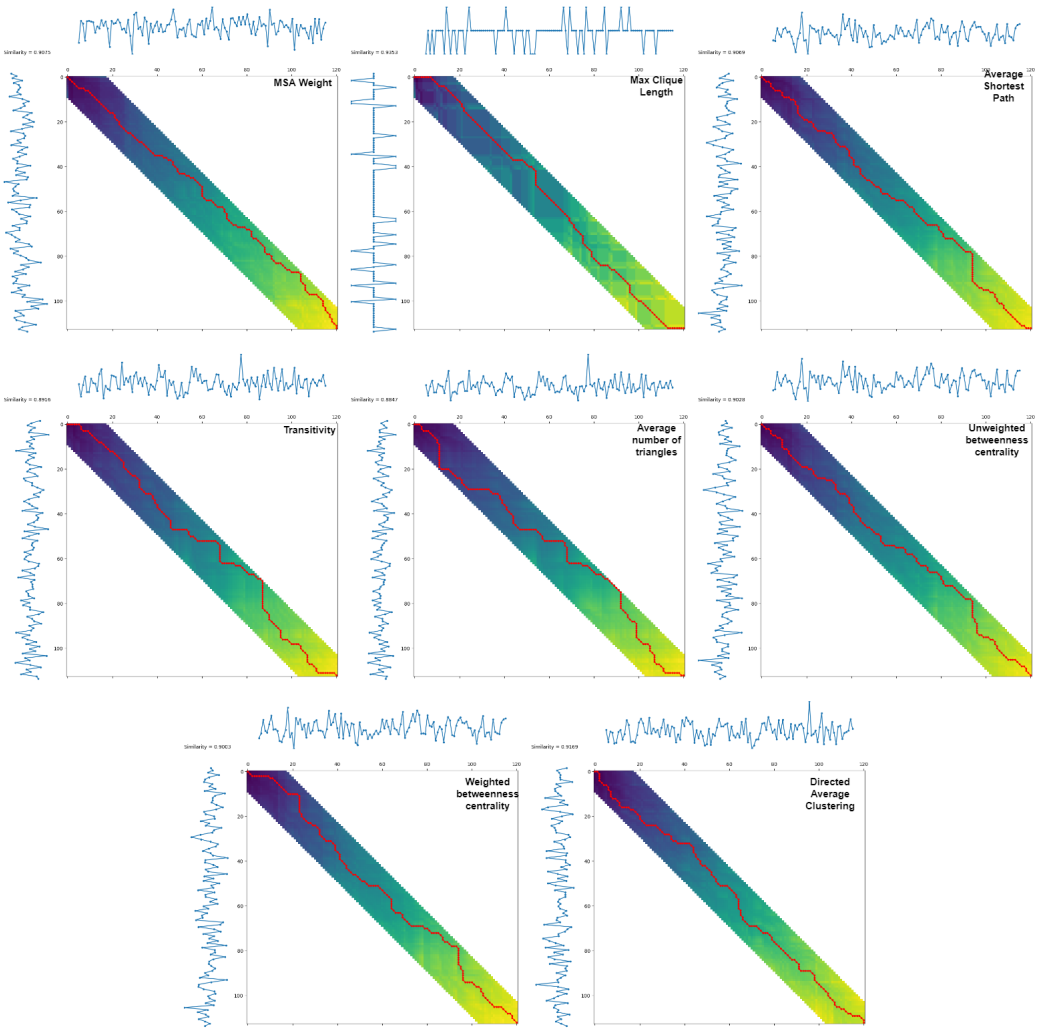


Figure 6.11: DTW Matching on all the metrics (name specified in the top right corner of each photo) for the banana and elephant stimuli with banana being on the vertical axis and the elephant on the horizontal axis.

Moreover, in the figure 6.11, the same concept as before is presented, but this time, the evolution on all the metrics is displayed for the banana and elephant stimuli, with the name of the corresponding metric being displayed on top of the photo.

What is interesting about the warping path for each of these metrics is that, almost

all the time, the optimal path is contained almost entirely inside the constraint band. What this fact suggests is that the evolution of a metric on two different stimuli is rather quite similar, with the DTW algorithm being able to construct an optimal match between the two evolutions with a locality constraint of ± 10 windows.

Furthermore, for the spots where the optimal path would not have been fully contained inside the constraint band for the unconstrained version of the algorithm, there is actually a logical explanation as to why this happens. Considering the process of the trial segmentation, where, based on the length of the trial, a filtering was done, it is quite common for two trials to not have the same number of windows. Because of this, as it can be seen from the figure, the two signals do not have the same length.

In this concrete case, the elephant had a number of windows greater than the banana stimuli and, therefore, the matching is actually shifted to the right for each individual metric. What this suggests is that, if the two trials were to have the same length, then the optimal path could have been built with an even smaller locality constraint.

Nevertheless, what the matching proves is actually a fact that is intuitively correct. More concretely, this analysis using the DTW algorithm actually proves that the statistical properties of the graphs generated by our method actually follow the same evolution with respect to a pair of two stimuli. This is rather expected because of the way the experiment was conducted. Since, no matter the stimulus, the experiment's task was the same, namely object recognition, it logically follows that the same processes would have taken part in this cognitive endeavour and, therefore, the captured time snapshots of brain activity would be relatively the same across all studied stimuli.

6.5 Additional Results

In order to see if the generated matrices hold any discriminatory information with respect to the trials, we have made use of the DGCNN classifier model developed by our colleague [14]. This model is a binary classifier, capable of classifying examples that have a graph structure.

Initially, our colleague applied this classifier on the functional networks generated in [2] [3], over a dataset that contained the networks for all the subjects, trying to classify the graph either into the *seen* class or the *not seen* class, but the performance obtained was not better than random guessing. Afterwards, our colleague tried applying it at a subject level, meaning one classifier per subject, but the results were the same. Her conclusion was that the functional networks generated do not hold any discriminatory information that can make the DGCNN model be able to distinguish between the two types of graphs.

In this context, we wanted to see how well could the graphs generated with our approach be classified. However, we were not able to use the generated graphs we have discussed up until this point because the DGCNN model is able to classify only one graph at a time. Therefore, we needed to modify the RGW model to generate a matrix per trial.

Following the same reasoning as in 6.3.1, the model needed to be finetuned at a trial

Table 6.5: RGW performance on trial 209.

Blocks	Layers	Functional Network	Previous Learnt Matrix	MAE	MASE
2	4	False	False	0.042	0.481
2	4	True	False	0.046	0.533
10	2	False	False	0.052	0.654
10	**2**	**True**	**False**	**0.028**	**0.327**

Table 6.6: RGW performance on trial 210.

Blocks	Layers	Functional Network	Previous Learnt Matrix	MAE	MASE
10	2	False	False	0.039	0.350
10	2	True	False	0.038	0.346
10	2	False	True	0.037	0.343
10	**2**	**True**	**True**	**0.035**	**0.329**

level. Therefore, we have first tuned it on trial 209 without using the previous learnt matrix and, with the best model from here, we have fine tuned it on trial 210. As it can be seen from the tabels 6.5 and 6.6, the optimal set of hyperparameters is the same one as before, meaning an architecture of 10 blocks, 2 layers, using as an input the functional network of the trial and the learnt matrix of the previous trial. Moreover, what is interesting to note here is that the performance of the RGW model in the optimal use of hyperparameters is actually almost the same in both types of generations, namely at trial level and at window level.

Having the graphs defined, we have trained the DGCNN model on two scenarios, namely at subject level and over all the subjects as a binary classification problem, keeping only the *seen* and *not seen* classes. The classifier's accuracy can be seen in table 6.7.

As it can be observed from the table, in almost all the subject, the performance of the DGCNN model is far better than random guessing, with some subjects having a classification accuracy as large as **72%**. Moreover, what is even more important to note here is that, in the case of all the subjects, the classification accuracy of the DGCNN model is almost 60% which means that, even in the case where the dataset consists of the entire set of subjects, the classifier's performance is better than random guessing.

What this analysis proves is that the information contained in the generated matrices by the RGW model were able to capture discriminatory information that could be used by the DGCNN model in order to distinguish between two classes, a problem that previously could not be solved using functional networks. Moreover, it is important to note here that this matrices were generated and classified at a trial level. However, considering the fact that the RGW model is able to extract multiple snapshots from a trial, it follows

Table 6.7: DGCNN accuracy.

Subject	Accuracy
1	68.43%
2	72.82%
3	60.00%
4	52.56%
5	64.00%
6	59.69%
7	56.92%
8	47.89%
9	52.75%
10	48.75%
11	70.51%
all	59.95%

logically that a classifier model having the ability to classify a set of graphs evolving in time should reach an even better performance, but this fact needs to be further studied in order to be validated.

As one last mention here, what is also interesting to take note off is that the accuracy of the DGCNN model for the dataset containing all the subjects in a binary classification problem proves to be equal to the accuracy of the DEC model in a multiclass classification problem. This fact indicates that our novel architecture is able to reach the performance of a Deep Learning model by using only the RAW EEG signal.

Chapter 7

User's manual

In this chapter, the project's prerequisites are presented alongside an installation guide for a user to be able to run this solution on his local machine.

7.1 Prerequisites

The software requirements needed in order to run the application are:

- Operating system: Window, Linux or MacOS
- Python 3.7
- Git
- All the libraries and packages presented in chapter 5 need to be installed. The name of the library and the version can be found in the file *project_requirements.txt* which is placed inside the project. In order to install it, one needs to run the command: **pip install -r project_requirements.txt**

With respect to the hardware limitations, the minimal hardware requirements for executing the solution are:

- 4 cores CPU
- 12GB RAM
- 50GB Storage
- 8GB GPU VRAM (optional; the execution can be performed without a GPU)

7.2 Running the application

As specified in chapter 5, a wrapper GUI for the entire application has been developed in order to aid the user in easily using this application. In order for an user to start the GUI, it just needs to run the file *main_gui.py* which resides in the project's folder as follows:
./main_gui.py.

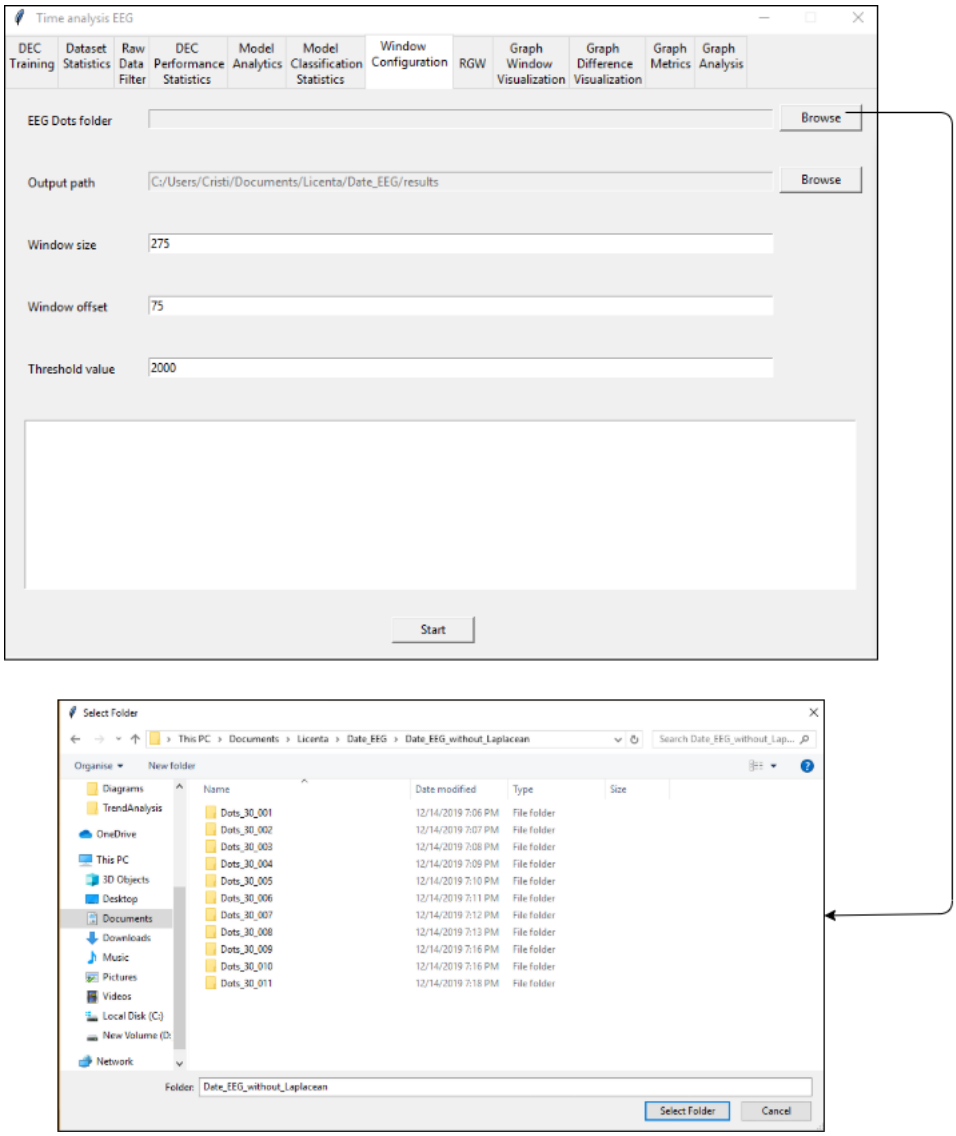


Figure 7.1: The Graphical User Interface of the application.

After running the script, a window is opened where the GUI can be seen by the user. The layout of the GUI is presented in 7.1. The user can choose from the tabs on the

top the module he wants to execute. For that module, it needs to fill the corresponding fields that are present on the interface. He can either browse for folders or manually fill the single valued parameters. After completing all of them, the user can run the application by pressing the **Start** button and can observe the progress of the application in the log area present at the bottom of the interface.

Chapter 8

Conclusions

The purpose of this thesis was the development of an architecture capable of extracting dynamic correlations from multivariate timeseries data, at a self-identifiable granularity level, able to capture meaningful processes that occur during the evolution of the studied signals. As an additional toolkit, the means of visualizing and analysing the discovered networks were provided. Moreover, from a Neuroscience perspective, this solution represents an insight into identifying the cognitive processes that arise in the brain during visual recognition, providing the method from which the evolutionary functional states of the brain can be studied.

In the following sections, the contributions brought to both the Computer Science field and the Neuroscience field are presented, together with an intuition regarding further improvements that could be made to the current solution. However, it is important to note here that the development of this solution has been done as a joint intellectual effort between my colleague [12] and I, with the individual contribution being present only at the implementation level. Because of the sequential nature of the solution, each conceptual component presented in chapter 4 was the result of a mutually intensive collaboration without which the final solution would not have existed if it were to be developed by only one of us. Therefore, the contributions will be discussed from the perspective of the solution as a whole and the value that it brings to the corresponding fields.

8.1 Contributions

8.1.1 Computer Science

From the perspective of Computer Science, it must be pointed out that the current solution is applicable in a wide range of domains. Even if, in this thesis, the subject of visual recognition was studied, the potential of the architecture is not constrained nor restricted to this particular application. In fact, the developed solution can be used for any multivariate timeseries data having a set of dependencies that change in time. In this

context, some use cases in which the current solution may be applied include, but are not limited to, sales analysis or stock analysis.

Moreover, the contribution brought to this field consists of not only an application encapsulating the functionality presented in this thesis, but also a set of novel conceptual tools in the context of Machine Learning. More concretely, two Machine Learning approaches have been defined, namely a multilabel classifier and a timeseries analysis pipeline, both of them having the means of validating and interpreting their results.

Therefore, in this context, the contributions to the Computer Science field are presented with respect to the components of the conceptual architecture presented in figure 4.1, focusing on both implementation and theoretical aspects. Thus, the contributions are:

- Development of a tool capable of analysing and visualizing a set of signals with respect to their length, consisting of multiple possible aggregations between them.

- Design of a machine learning model capable of performing multilabel classification on multivariate timeseries data, able of reaching performances comparable to other studied methods. Moreover, even if, in the current solution, the model allowed only 2 sets of labels, the architecture of the model can easily be extended to accommodate multiple label sets.

- Development of a toolkit capable of performing hyperparameter tuning and an analysis with respect to the model's interpretability on the previously mentioned classifier.

- Modelling a pipeline of multiple Spatial-Temporal Graph Convolutional Networks (STGCNs) which, connected sequentially by using the previous STGCNs' latent information, are able to extract the hidden spatial dependencies from a set of multivariate timeseries at the specified timeframe in the signals.

- Development of a visualization tool able to aggregate and display the previously learnt correlation matrices.

- Development of an analysis tool capable of generating and displaying the evolution of statistical properties computed over the generated networks. Even if a certain pool of graph metrics is currently available, the implementation of the tool actually allows easy integration of new metrics into the module.

8.1.2 Neuroscience

From the perspective of Neuroscience, the current solution can be used to study any set of brain signals that the person using this solution wants to analyse. However, in the specific use case studied in this thesis, the set of signals represented the EEG recordings captured during the visual recognition process on a human subject. In this context, the following contributions can be stated regarding this field:

- Generation of a set of Dynamic Networks representing the functional dependencies occurring in the brain during the image recognition process. Because each of the generated networks represent a particular state of the brain at a particular snapshot in time, a neuroscientist can actually study the evolution of these correlation matrices and gain insight into how certain cognitive tasks are performed by identifying the underlying processes that control the dynamicity of the functional correlations.

- Visualization of the Dynamic Networks with respect to the brain's structure. More concretely, using the relative position of the electrodes on the EEG headset, an aggregation over the central brain regions can be defined which can provide an overview over the functional dependencies captured.

- Extraction of structural properties from the Dynamic Networks, together with an analysis of the evolution of these properties in time. More specifically, by analysing network properties such as clustering or centrality, a neuroscientist can gain insight into how the functional dependencies of the brain are structured and how does this structure change with respect to time.

8.2 Further Improvements

As for any piece of software, there is always room for improvement, whether it is design wise or architecture wise. In this context, some further improvements regarding the current solution include, but are not limited to, the following:

- Based on the observations from the analysis of the DEC model, the *Dual Ensemble Classifier* model performance appears to be limited by the loss of relevant information from long trials. Therefore, in order to improve the classification performance of the DEC model, a study regarding taking into account larger portions from the trial may be conducted.

- In the context of the *Recurrent Graph WaveNet* pipeline, the modelling of the evolution of brain states with respect to the states that previously ocurrred has been done in a sequential manner, with each Graph WaveNet model using only the Dynamic Network of the model placed before it in the pipeline. However, it could be the case that the state evolution consists of a large set of temporal dependencies between different states. In this context, a possibility of enhancing the Recurrent Graph WaveNet approach would represent the integration of multiple previously generated Dynamic Networks, either aggregated into only one network or given as multiple input matrices to the model.

- The analysis toolkit of the Dynamic Networks can also be further enhanced. Firstly, new metrics may be added to the set of metrics already defined that study other properties of the Dynamic Networks, such as page ranking or communities. Moreover,

the networks could also be studied by firstly performing an operation to their weight matrix, such as computing the Laplacian.

- Moreover, one new module could be included that is capable of analysing the Dynamic Networks from a Machine Learning perspective. More concretely, starting from the *DGCNN* [14] model, an enhanced graph-based classifier could be developed capable of analysing not one, but rather a set of multiple graphs which, in this case, would be the Dynamic Networks. Considering this, the classifier could be trained on the *response* type and, if a good performance is obtained, the latent information of the model could be further analysed in order to provide an insight in how the functional dependencies encapsulated by the Dynamic Networks can actually be modelled as a deterministic mathematical function.

Moreover, it is important to note here that both the Dynamic Networks extraction methodology and the obtained results are to be published in a scientific journal, namely the *Journal of Computational Neuroscience*. The reason for which this was not already done is because the results to be presented in the article will be performed on a different dataset than the one used by this thesis and we are currently waiting for its generation.

Bibliography

[1] H.-J. Park and K. J. Friston, "Structural and functional brain networks: From connections to cognition," *Science*, vol. 342, 2013.

[2] D.-A. Dumitru, "Studiul modalităților de extragere a retelelor din creier în contextul percepției vizuale," Bachelor Thesis, Technical University of Cluj-Napoca, 2018.

[3] E.-B. Ceuță, "Analiza metodelor de obținere a rețelelor funcționale din creier folosing semnale eeg," Bachelor Thesis, Technical University of Cluj-Napoca, 2018.

[4] S. Yan, Y. Xiong, and D. Lin, "Spatial temporal graph convolutional networks for skeleton-based action recognition," 01 2018.

[5] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Graph convolutional recurrent neural network: Data-driven traffic forecasting," 07 2017.

[6] M. Schnaubelt, "A comparison of machine learning model validation schemes for non-stationary time series data," 11 2019.

[7] V. V. Moca, I. Tincas, L. Melloni, and R. C. Muresan, "Visual exploration and object recognition by lattice deformation," *PLoS ONE*, vol. 6, 2011.

[8] Headcaps. [Online]. Available: https://www.biosemi.com/headcap.htm

[9] Z. Wu, S. Pan, G. Long, J. Jiang, and C. Zhang, "Graph wavenet for deep spatial-temporal graph modeling," 08 2019, pp. 1907–1913.

[10] J. Dicarlo, D. Zoccolan, and N. Rust, "How does the brain solve visual object recognition?" *Neuron*, vol. 73, pp. 415–34, 02 2012.

[11] R.-G. Pașca, "Extracting neural information from the visual cortex using deep learning," Bachelor Thesis, Technical University of Cluj-Napoca, 2019.

[12] V.-C. Buda, "Spatial-temporal graph convolution for dynamic network extraction in the context of visual perception," Bachelor Thesis, Technical University of Cluj-Napoca, 2020.

[13] D. Nikolić, R. Mureşan, W. Feng, and W. Singer, "Scaled correlation analysis: A better way to compute a cross-correlogram," *The European journal of neuroscience*, vol. 35, pp. 742–62, 03 2012.

[14] L.-D. Palcu, "Brain functional networks - an interpretable graph classification solution," Bachelor Thesis, Technical University of Cluj-Napoca, 2019.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. Yu, "A comprehensive survey on graph neural networks," 01 2019.

[16] A. oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 09 2016.

[17] S. Shleifer, C. McCreery, and V. Chitters, "Incrementally improving graph wavenet performance on traffic prediction," 12 2019.

[18] R. Hyndman and A. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, pp. 679–688, 02 2006.

[19] L. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, pp. 35–41, 03 1977.

[20] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978.

[21] Biomedical signals acquisition. [Online]. Available: https://www.medicine.mcgill.ca/physio/vlab/biomed_signals/eeg_n.htm?fbclid=IwAR1QUjcbWdJ0lRMjyO051qJWeaRLlah9RluvuCxpO-XE89X_KRFS0Xbggcs