UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Modularization of reservoir computing networks for the recognition of brainstates

Xènia Domènech Gutiérrez

Director:	**Ignasi Cos**
Tutor:	**Gorka Zamora**
Realitzat a:	**Departament de Matemàtiques i Informàtica**
Barcelona,	**24 de gener del 2022**

Contents

List of Figures

List of Tables

Abstract

This project consisted of the research, study and modularization of a complex neural network, consisting of one or several reservoir computing modules. This is performed by means of Python scripts, aiming at learning in a way inspired on how the human brain does. The manucript starts with a theoretical introduction describing basic concepts of complex networks, listing their types and applications, so that the reader can understand how we have used the idea for the development of the script. The second part describes the process and impact of the modularization modules first described, and the comparison with the non-modularized program.

2022 Mathematics Subject Classification. 68T07

Introduction

The human brain, the social media and the World Wide Web are examples of highly complex network, which are systems composed of a large number of units interconnected through non-trivial and complex pattens of interaction. With this idea in mind, the project here presented aims at the study of modular complex neural networks, specifically on Reservoir Recurrent neural networks, to extrapolate the concepts and design a program, capable of producig a brain state representation as described by the temporal series datasets we described.

My contribution is a characterization of a modular reservoir network to capture high-dimensional statistics from temporal series, by contrast to the original single module reservoir [5]. To this end, I developed ad-hoc python scripts, as to allow for a performance comparison between the different architectures. In brief, I extended from a single reservoir to a two-module architecture, to be consistent with the brain distribution of areas from which neural recordings were performed [1].

The performance of this modular architecture was consistently compared with respect to a baseline, single module architecture, and was tested with a dataset consisting of temporal series recorded from the motor and pre-motor cortices of non-human primates [1] during performance of a five-state motor reaching task. In summary, the main goal was to assess the effect of brain-like modularity onto the final neural representation, assessed through its clustering accuracy.

The main result of the work is that as one would expect, modularized network yields better accuracy than the unmodularized one. This makes sense because the dataset used to train and test the network comes from different parts of the cortex, and therefore from different modules of the brain.

Previous work

This project extends on a preliminary study on reservoir computing for the representation of brain temporal series [5]. He created a preliminary version of the script, which we have modularized with the intention of creating a more faithful neuro-inspired architecture and to study the influence of modularity.

Addicionaly, the dataset used to create and operate the network originates from a dataset previously used by Michael A. Depass in his master thesis [1]. Moreover, most of the matematical operations responsible for the network's training are originated in the work of Núria Sánchez's MSc Thesis [7].

However, the first piece of relevant work was pursued by Arnau Naval in his thesis [5]. In this work, the focus was on building a reserve computing model capable of identifying different states related to movement from a set of neural data recorded from a non-human primate. To this end, two types of analysis were performed: on the one hand, he tried to make the program learn to classify five states that the macaque brain went through and with which we trained the program. On the other hand, he did a similar process to study whether the program could distinguish when the data originated from the primate using the left or the right arm.

Furthermore, it was also interesting to see that indeed reservoir computing was not only able to capture the features of temporal series recorded from specific brain regions and that the accuracy of the data after passing through the model helped to classify the states better than without using the neural network.

In our work, we will build on the five-state classification part, and by modularizing the network we aim to improve on previous results so that we can obtain more optimal accuracy for both the linear and logistic classifier.

Hypothesis and objectives

The goal of this work is to study and characterize the influence of modularity on the neural dynamics of reservoir computing networks. Specifically, our interest is to study the impact of brain-like modular reservoirs in the representation of brain states, characterized by temporal series recorded from separate cortical areas. Can a modular reservoir network outperform a single reservoir network?

We formulate the next operational hypotheses:

- A modularized reservoir network yields a more faithful (accurate) representation of brain states than an unmodularized reservoir.
- An increasing number of nodes improves the accuracy of any state representation. We predict that the accuracy would have a logarithmic growth as a function of the number of nodes.
- In a modularized reservoir, the use of a linear and/or logistic classifier would not yield fundamentally different accuracies.

Our analysis of performance, in addition to accuracy, will be based on a newly designed metric: the join_probability, which we deem provide a more complete characterization of network performance, than the simpler accuracy metric.

Chapter 1

Complex network

In this section you will find a short explanation of how complex networks work and their most important utilities, in order to be aware of their relevance. [3]

1.1 Artificial neural network

Artificial neural networks (ANN) are the basis of deep learning, a subfield of machine learning in which algorithms are based on a trial an error method. This process of learning is inspired by the structure of human brain which is made up of layers of neurons, the central processing units of the network.

To understand the brain's learning process, let's assume that we have an image that consists of 28 by 28 pixels, in total 784 pixels. Each pixel is fed as an input value to the neurons in the first layer, and then the neurons in this layer are connected through channels to the neurons in the next layer. It is important to note that all these neurons are associated with a numerical value called bias, which is added to the sum of the input values. The new value of each neuron is passed through a threshold function called the activation function, which transmits the data to the neurons of the next layer through the channels. This forward propagation process is repeated through all the layers of the network until it reaches the output layer, where the neuron with the highest value determines the output. Below, you can see a schematic of the brain learning process.

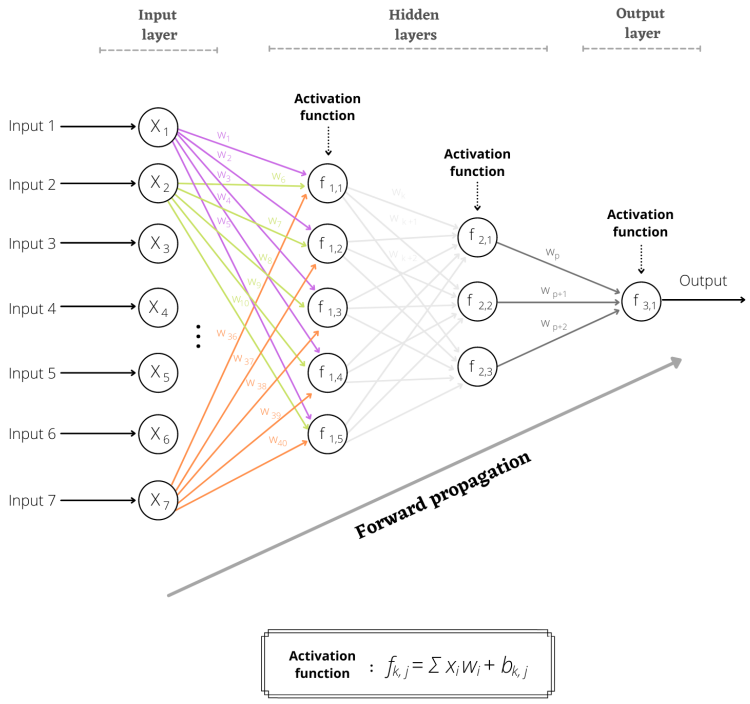

Figure 1.1: Fully connected feedforward neural network with two hidden layers and only one node in the output layer

Once we understand the process, we must comment on a drawback that we encounter, and that is that there is the possibility that our neural network makes an erroneous prediction, but how does the network find out? We must bear in mind that our network still has to be trained by means of several different inputs of what we want to learn by repeating this neural process. The learning process is linked to the magnitude of the error that tells us how wrong we are, and it decreases as we pass more information to it.

Thus it is crucial to understand that this cycle of backward and forward propagation is done iteratively with multiple inputs, and that this process continues until the values are assigned in such a way that the network can correctly predict what has been learned. In most cases, the time it takes to train neural networks can take hours or even months. But this time is a reasonable trade-off, as for example, thanks to these complex systems we are able to enable facial recognition cameras in phones to estimate a person's age based on their facial features.

Neural networks are trained to understand patterns and have great application in our daily lives. Among other things, they can detect the possibility of rain, predict stock prices with high accuracy, and can even learn musical patterns to the point of being able to compose a new melody.

1.2 Types of ANN

There are many kinds of artificial neural networks used for the computational model. The set of parameters and operations of mathematics determines the type of neural networks to be used to get the result. Here we will discuss some of the critical Neural Network types in Machine Learning:

- **Modular Neural Networks:** In this type of neural network, many independent networks contribute to the results collectively. There are many sub-tasks performed and constructed by each of these neural networks. This provides a set of inputs that are unique when compared with other neural networks. There is no signal exchange or interaction between these neural networks to accomplish any task.

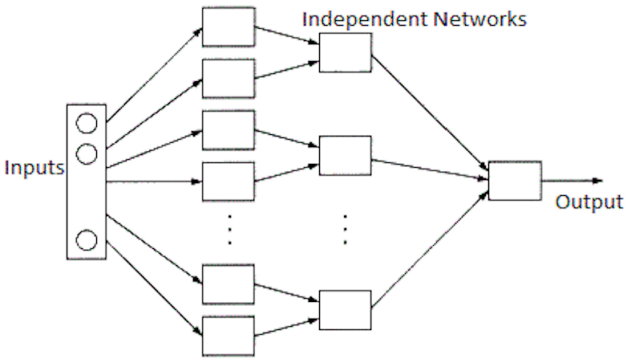

Figure 1.2: Modular Neural Networks example

- **Feedforward Neural Network:** The information in the neural network travels in one direction and is the purest form of an Artificial Neural Network. This kind of neural network can have hidden layers and data enter through input nodes and exit through output nodes. Classifying activation function is used in this neural network. There is no backpropagation, and only the front propagated wave is allowed. This type of neural newtork is the one that can be seen in the figure 1.1.

- **Kohonen Self Organizing Neural Network:** In this neural network, vectors are input to a discrete map from an arbitrary dimension. Training data of an organization is created by training the map. There might be one or two dimensions on the map. The weight of the neurons may change that depends on the value. The neuron's location will not change while training the map and will stay constant. Input vector and small weight are given to every neuron value in the first phase of the self-organization process. A winning neuron is a neuron that is closest to the point. Other neurons will also start to move towards the point along with the winning neuron in the second phase.

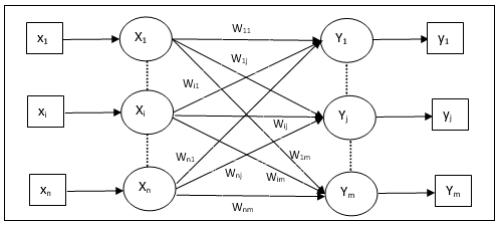

Figure 1.3: Kohonen Self Organizing Neural Network example

- **Recurrent Neural Network:** A RNN is an Artificial Neural Network (ANN) that, in addition to feed-forward connections, also has recurrent connections. They are distinguished by their "memory", as they can utilize the information of previous inputs to influence the present inputs and/or outputs.

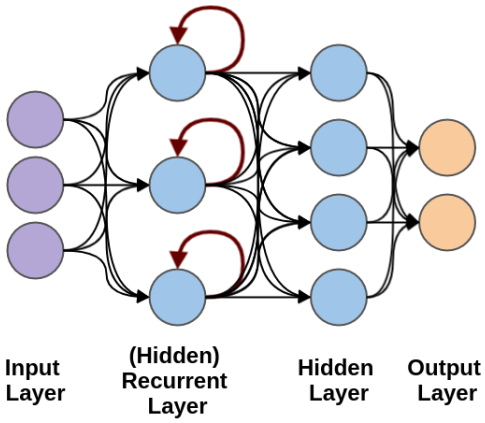

Figure 1.4: Recurrent Neural Newtork example

RNN are a very powerful tool for solving complicated temporal automated learning problems, and since they can take variable length inputs we will see that they are good candidates to use it in our project. That is why the whole practical part is based on the modularization of a recurrent neural network-based program using reservoir computing. Therefore, we will proceed to explain a bit more in depth how recurrent neural networks work with the use of reservoir computing.

1.3 Recurrent neural network with reservoir computing

Reservoir computing is an algorithm process information generated by dynamical systems using observed time-series data. Importantly, it requires very small training data sets and thus requires minimal computing resources. Promising to be both theoretically sound and computationally fast, reservoir computing has already been applied successfully to numerous fields: natural language processing, computational biology and neuroscience, robotics, even physics. The use of reservoir computing in our RNN will solve the basic problem of this kind of network, it is hard to train. Reservoir Computing is a framework that aims to make this process easier without a significant loss of performance.

The difference between a classical RNN and one working under the Reservoir Computing paradigm is that with RC all the network's connections are fixed except for the ones at the readout layer, making it computationally cheaper than other RNN approaches [5], [2], [4], [8].

The structure of the RC network that was used in the present work can be described in the following way:

- An input layer of size K denoted by u. In our case $K = 128$ since our input will be the number of electrodes.
- A hidden layer of size N denoted by x. N is given by the number of nodes of our network.
- An output layer of size L denoted by y. In our case, the size of the output layer is given by the number of states (5) that we want to be able to classify.

The following connectivity matrices will tell us how the different layers connect between themselves:

- The input connectivity matrix $W^{in} \in M(\mathbb{R})_{N\times K}$ holds the connection weights that relate the input units of the network to its internal units. Therefore, $W^{in}_{i,j}$

represents the connection weight of the input unit with index j to the internal unit with index i.

- The internal connectivity matrix $W \in M(\mathbb{R})_{N\times N}$ holds the connection weights between the different internal units of our network.
- The output connectivity matrix $W^{out} \in M(\mathbb{R})_{L\times N}$ gives us the weights from the internal units of our network to the output layer.
- Finally, the feedback connectivity matrix $W^{back} \in M(\mathbb{R})_{N\times L}$ holds the connection weights from the output layer to the internal units of the network, but for the present work these weights were set to 0.

Therefore, the internal state of the network will be updated at every timestep as indicated by the following formula:

$$x(n+1) = f(Wx(n) + W^{in}u(n+1) + W^{back}y(n))$$

where f is an activation function.

It can be observed that the internal state of the network at time $n+1$ is computed using its previous state at time n. Since we are dealing with a RC network, we will set fixed values for both W and W^{in} and we will only train the values of W^{out}. The weights of W^{out} are obtained with a regression method of our own choosing and, for the present work we chose to use a logistic regressor.

The values of the output y are obtained with the next equation:

$$y(n+1) = f^{out}(W^{out}x(n+1))$$

where f^{out} is the activation function. In our case these activation functions are hyperbolic tangent functions, which are one of the most commonly used activation functions for RNNs and are defined as:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Once the basic structure of the network is defined, we procceed to explain in detail all the modularization and how we use this powerful tool, such as reservoir computing, for our particular case.

Chapter 2

Modularization of the network

This section will be dedicated to the practical part of the work, that is, to the development of the script and the results obtained. To begin with, we will talk about key aspects of the program, then we will give a brief explanation of the script and finally we will test it.

2.1 Data

In order to train this model, we will be using a Local Field Potential (LFP) dataset collected by the Dancause Laboratory at the University of Montreal. The intracerebral local field potential (LFP) is a measure of brain activity that reflects the highly dynamic flow of information across neural networks. This is a composite signal that receives contributions from multiple neural sources, yet interpreting its nature and significance may be hindered by several confounding factors and technical limitations. This measure of brain activity constitutes an increasingly important tool both in neurophysiology and medicine. Specifically, these neural data were collected while Non-Human Primates (NHP's) performedreward retrieval tasks involving reaching and grasping.

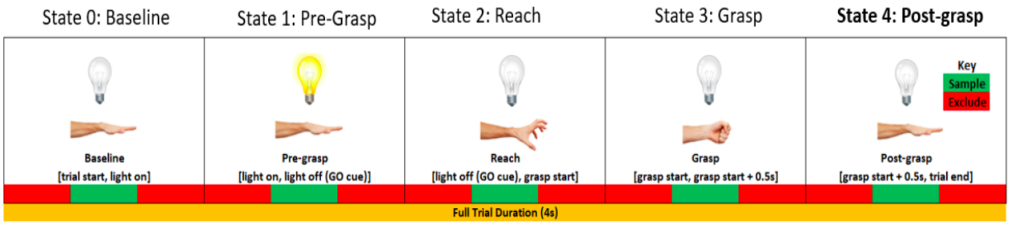

Figure 2.1: Timeline of a single trial

The data set is divided into trials, and the figure above shows the structure of each trial. Initially, it starts in a baseline condition, the second state is when the visual signal is active, the third is begin the movement towards the handle. The next is reache the object and the last state is to grab it. Finally, the primate drops the handle expecting a reward.

The data we will be using has been separated by motor state (5 states in total, as we have explain above) and it includes data in which the handle to reach for was positioned in diferent orientations (0, 45, 90 and 135 degrees). The dataset consists of 140 trials, each one proceed form an LFP collected from 128 electrodes, sometimes referred to as channels. Each trial has a duration of roughly 0.25 seconds (508 samples). Moreover, each trials in total resulting in an input matrix of size [128,508,140,5].

2.2 Hyperparameters

In Machine Learning and Deep Learning, the hyperparameters are those parameters whose values are set prior to the training process, for example, the number of nodes in the reservoir layer, or the probability of connection between layers.

Hyperparameters are important because they directly control the behaviour of the training algorithm and have a significant impact on the performance of the model that is being trained.

In our program we have four hyperparameters that need to be defined beforehand. These hyperparameters are: the number of nodes in each reservoir (num_nodes), the probability of connection between the input layer and the reservoir layer (input_probabiliy), the probability of connection between the nodes in each reservoir (reservoir_probability) and finally, the probability of connection between the two reservoirs (join_probability).

In the previous work to the modularization of the network, only the first three hyperparameters were available, and the values were studied in such a way that the program would learn more accurately. After this study through heatmaps, it was deduced that the best values for the hyperparameters in the non-modularized network were input_probability=0.15, reservoir_probability=0.5 and num_nodes > 100.

However, in our network we have an extra hyperparameter, the join_probability. Therefore, we will now proceed to study how this factor affects the other three values and what are the new optimal values for the program to learn more accurately.

2.3 Classifiers

To train our network, we need a classification model to classify the training matrix of our network. To obtain the cataloguing of the five states, the program consists of two classifiers: the linear and the logistic one. We will now proceed to explain in more detail what a classification model is and the types of classifier we use.

Classification models are used on labelled data sets. These data are considered labelled because they belong to a certain class, which is known and bounded (has a finite range of values). Examples of classification models could be predicting whether a customer will make a purchase or not, predicting whether a customer will leave our company or not, or predicting whether a customer will be able to repay a loan or not. These are examples of binary classification (there are only two classes, yes or no) but there are also examples of multiple classification. These examples would be detecting which animal a picture belongs to or identifying which musical instrument a melody belongs to. Now we are going to talk about two kind of classification model: lineal regression and logistic regression.

2.3.1 Lineal regression

Linear regression in statistics, is an approach based on the conditional probability of y given X, used to model relationships between scalar independent variables and one or more, dependent variables. In this approach, data are modeled through a LPF and weights are computed to allow credible predictions for unknown inputs of the same type which the regressor was trained on. [6] Linear regression in machine learning is a supervised learning algorithm whose output computes as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + .. + \theta_n x_n.$$

Where h_θ is the hypothesis formulated by the regression and the x's are the input. Here you can see an example of Linear Regression over a dataset of houses.

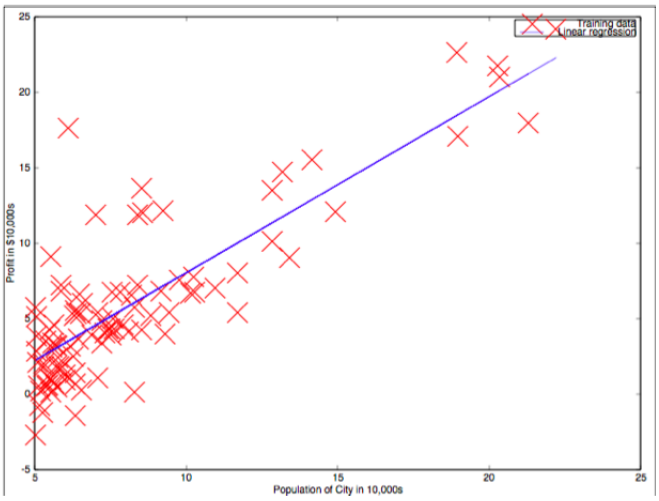

Figure 2.2: Linear Regression over a dataset of houses

2.3.2 Logistic regression

This problem is considerably different from the linear regression one, composing another field in the Analysis of regressions. Logistic regression is a relatively simple machine learning technique whose results are interpretable and widely used. It works very well when there is a lot of data and the interrelationships between them are not very complex. Given a set of data, it studies the probability of a data to belong or not, to a specific class. Compared to Linear Regression, this algorithm solves the task providing an hypothesis $h_\theta(x)$ in the range [0,1] computing the probability $p(y = 1|x; \theta)$ that a data belongs to a positive class $(y = 1)$ or a negative one $(y = 0)$. This is done computing the hypothesis using a logistic function, in the range [0, 1]:

$$h_\theta(x) = g(z), \quad \Longrightarrow \quad g(z) = \frac{1}{1 + e^{-z}} \quad \Longrightarrow \quad z = \vec{\theta}^T x;$$

Where h_θ is the hypothesis formulated by the regression and the x's are the input. Below you can see and example of Logistic Regression over a dataset of exams' scores. [6]

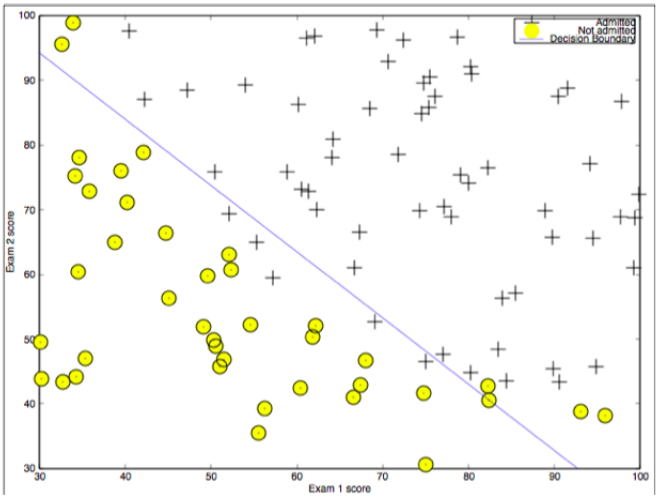

Figure 2.3: Logistic Regression over a dataset of exams' scores

2.4 Script explanation

It is important to discuss the operation of the network so that it is easier to understand why it works and the final comparisons between the modularized and non-modularized program. Mainly, the program consists of three classes: Network, Data and Reservoir, besides having a main where we manage the use of the classes and the articulation of the script. We will proceed to explain briefly each class individually.

2.4.1 Data class

It serves basically to manage the neuronal information of the non-human primate, which we give to the program at the beginning of everything to be able to train and test itself. Specifically, the file with the information is a .mat that contains a matrix of dimension four that we will comment later.

2.4.2 Reservoir class

This is where we create and define the characteristics of the reservoir individually with the hyperparameters set in the main. In addition, this class is in charge of defining, training and testing the network. It is worth mentioning that in the modularized network three different reservoirs and three different networks are created: two of them we consider partial since they are in charge of managing independent information and represent different parts of the brain, and the third one is the combination of the two previous ones and the only one we train and test. Before proceeding to explain the next class, we are going to comment on some of the attributes of the class.

- To analyze performance of different brain functions, we can filter the data in different frequency bands, known to encode different brain processes. Filter_name is an attribute that determine which frequency band are we using.

- Classifier is the name of the classifier that we are using, it could be linear o logistic.

- Num_nodes is the number of nodes that has each parcial reservoir.

- Input_probability, reservoir_probability and join_probability are probabilities to define the adjacency matrices of the network.

2.4.3 Newtork class

Here is where we apply most of the mathematical operations necessary to operate the network. Next we will explain the general operation of our neural network, but roughly speaking, it is in this class where we operate the matrices that are created with the information of the connections between the nodes of the input, reservoir and output layers of our network. In this class is important to talk about the main parameters to understand who the script works. So let's see a briefly explanation of it.

- K is the dimension of the input denoted.
- N is the dimension of the hidden layer denoted.
- L is the dimensional of the output.
- T is the number of training steps
- Input connections, $W^{in} \in M(\mathbb{R})_{N\times K}$, adjacency matrix with the connection weights from the input units to the internal units of the reservoir. $W^{in}_{i,j}$ gives the weight of the connection from the j-th input unit to the i-th internal unit.
- Internal connections, $W \in M(\mathbb{R})_{N\times N}$, matrix with the connection weights between the different internal units of the reservoir. $W_{i,j}$ gives the weight of the connection from the j-th internal unit to the i-th internal unit.

2.4.4 How the script works?

Initially, we define the two partial reservoirs; the pre-engine and the engine, and then the full reservoir of the network which is made up of a combination of both reservoirs. It is important to note that these two reservoirs are simply initialised, i.e. we define the attributes but neither train nor test them. As we said before, we use the information of the partial reservoris to create the total reservoir, which is the one we train and test.

In the first part of this project, we have explained theoretically how reservoir computing works. However, will be at this point where we are going to explain more in general how our network works with the particular information that we use and after the modularization. Below, we show a diagram of the phases that the network goes through, which will make the explanation of the script more understandable.

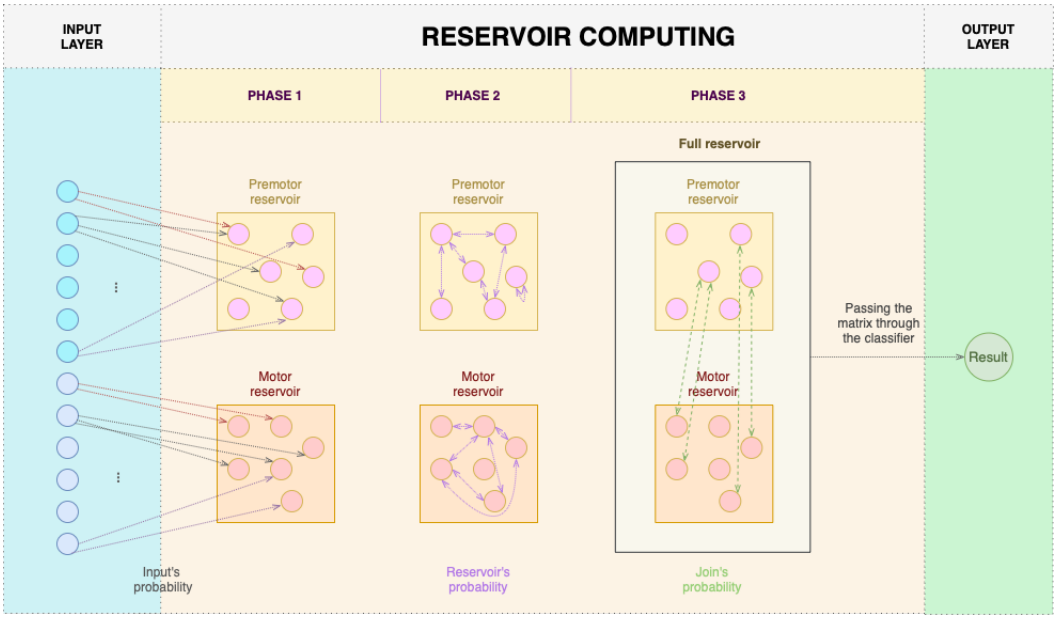

Figure 2.4: Diagram of how the script works

As you can see in the figure above, first there is an input layer, where each blue circle symbolises a different input. In our case, because of the type of data we are dealing with, we have 128 different inputs that represent the electrodes that were put into the non-human primate's brain to obtain the information. However, it is important to explain that we pass the first 64 inputs to the premotor reservoir and the remaining 64 to the motor. Initially, our program creates the W matrix for each reservoir individually, and is in these where we store the random connections between the nodes of the input and those of each partial reservoir. To create these random connections we use the hyperparameter input_probability.

Next, we create the W^{in} matrix, also for each separate reservoir, where analogously to the previous one, we randomly generate the connections between the internal nodes of the reservoir. However, for this matrix we use the hyperparameter reservoir_probability.

Subsequently, we have to repeat the proces with the full reservoir. Therefore, we create the W and W^{in} matrices for the total reservoir using the information from the partial matrices and generate the new matrix components randomly using the join_probability hyperparameter.

It is crucial to understand that W and W^{in} matrices are adjacency matrices representing the connections between the nodes of the layers, and their role is essential for training the network. The next step in our script is to train the network, and we do this by training each node in the network using mathematical opera-

tions. These operations involve the use of the adjacency matrices described above, and as a result, we obtain an average training and testing matrices. It should be noted that the calculations used in the functions to train the nodes are based on Núria Sánchez's master's thesis [7]. Finally, to train the program we pass the mean training matrix through a classifier, which in our case we have defined the linear and the logistic, and we can proceed to test the network.

2.4.5 The program

If you want to access the complete program and see the different classes explained above as well as test it, you can do so using the following QR.

2.5 Results

The main goal of the program that we have just explained above, has achieved the better accuracy's testing for the network. As we said previously, there are some parameters that must be defined before the network starts the training process. These parameters are the hyperparameters, and to know which ones are the most optimal, we will proceed to make some tests with it. Mainly, we have four hyperparameters that have been discussed above:

- The probability of binding between the input and the reservoir layer (input_probability).
- The probability of binding for each reservoir individually (reservoir_probability).
- The probability of binding between the two reservoirs (join_probability).
- The number of nodes.

2.5.1 Study of input_probability and reservoir_probability according to the number of nodes

To begin with, we will focus on the study of the optimal *number of nodes* that should have our reservoirs. Mainly, what we want to work at is, from what number of nodes the results are good and from what value of nodes is the accuracy stabilized. Obviously, the greater the number of nodes, the greater the computational expense of the program. Therefore, thanks to finding these accuracy thresholds based on the nodes that the reservoir has, we will be able to manage the relationship between the computational cost with good results.

To do so, we have made some heatmaps (each one with a given number of nodes) where there is the accuracy of the network using join_probability=0 and varying the input_probability and reservoir_probability.

The number of nodes that we have studied are: 1, 2, 5, 20, 50 and 100, and we have tested them with both classifiers: first with the linear and then with the logistic. To clarify the results you will see below, it is important to note that the number of nodes indicated on the title of the graph is for each partial reservoir. That is to say, if it indicates one node, it means that both the motor reservoir and the premotor reservoir consist of a single node each, and the number of nodes in the total one is the sum of these, in this case two nodes.

Studying the number of nodes using linear classifier

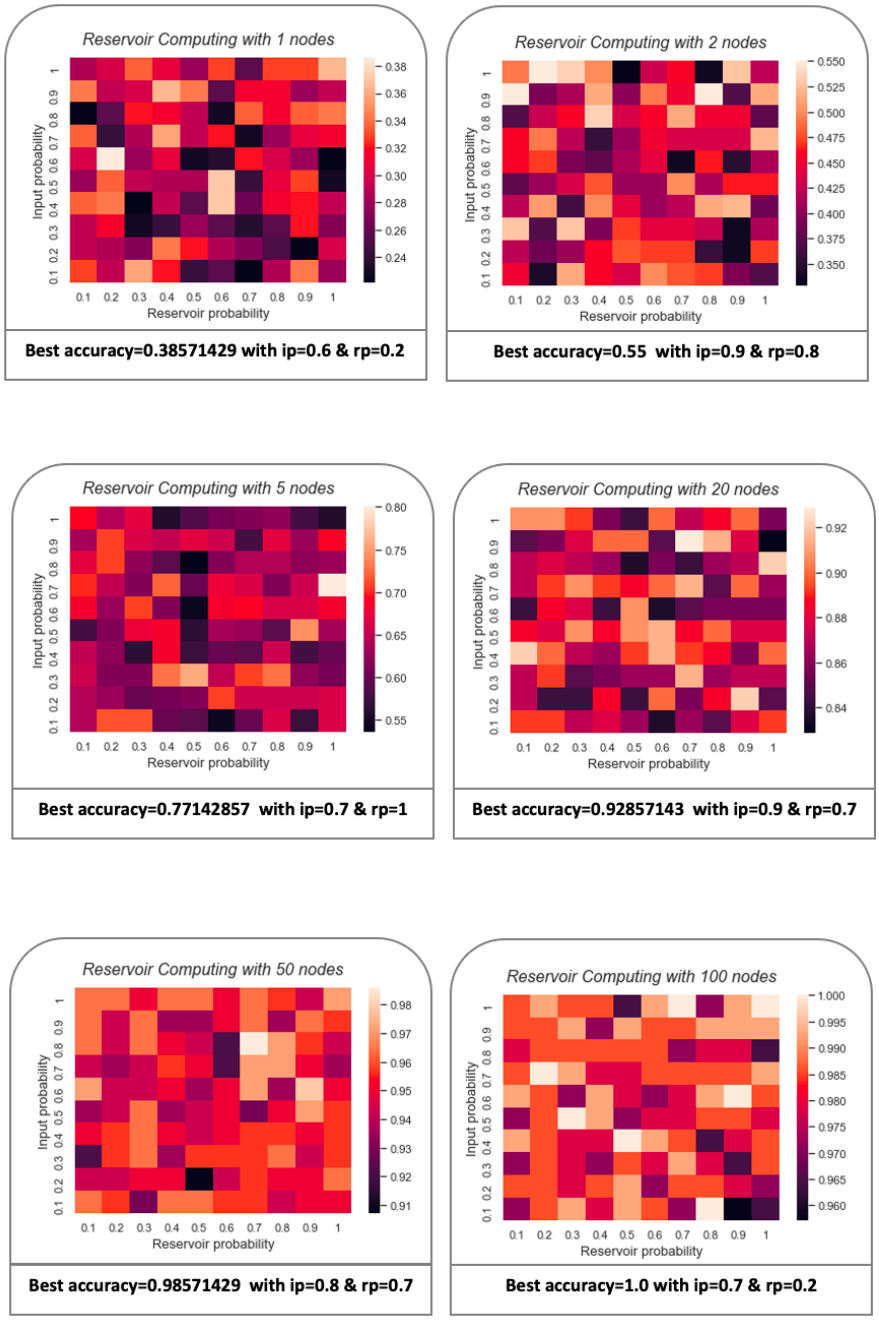

Figure 2.5: Accuracy of the network in terms of input_probability and reservoir_probability with linear classifier

Studying the number of nodes using logistic classifier

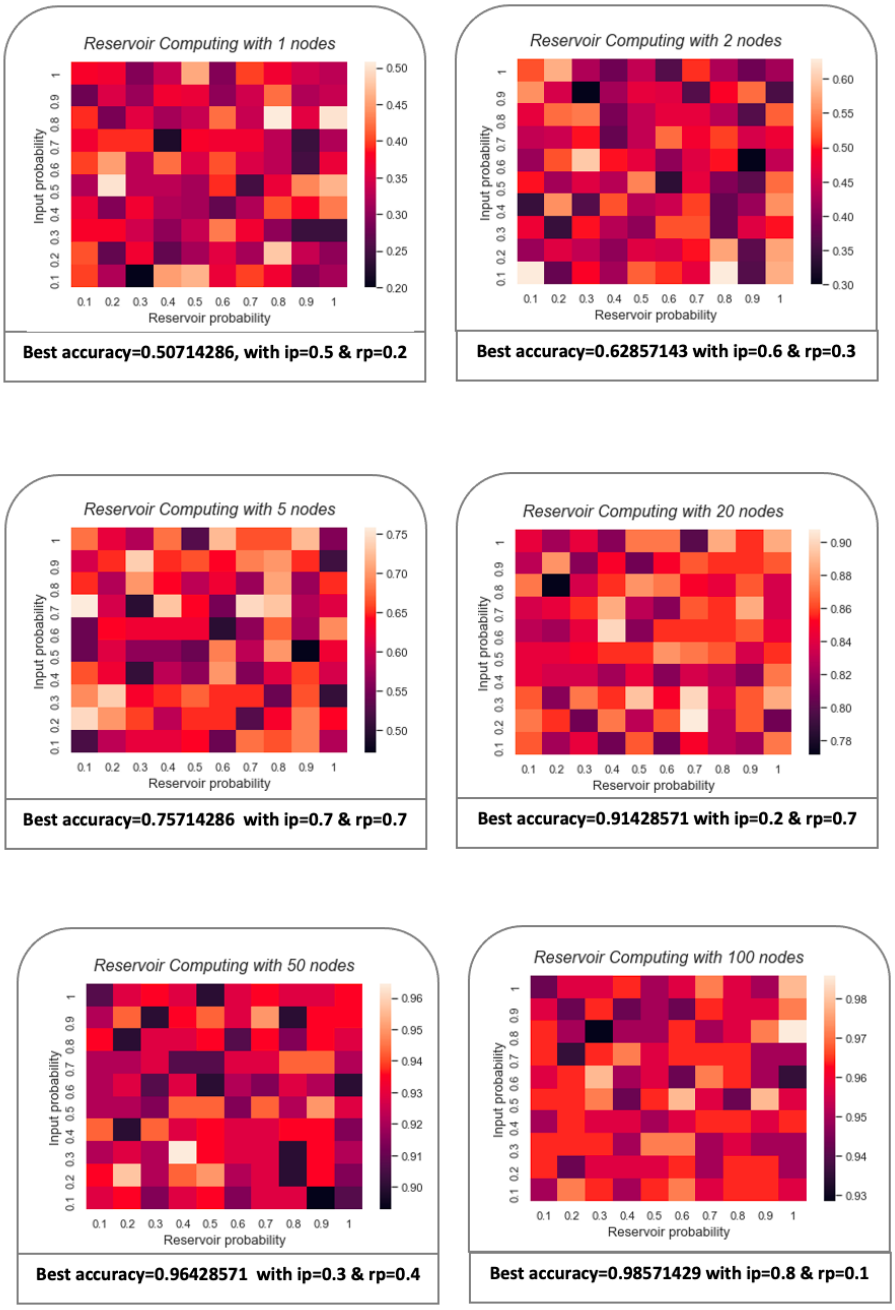

Figure 2.6: Accuracy of the network in terms of input_probability and reservoir_probability with logistic classifier

First of all, we can observe that the results with both classifiers are really similar. For that reason, we will discuss the graphic above in general. According to the previous heapmaps, the accuracy varies a lot using different number of nodes when we use a few, and almost doesn't vary when we have a lot. In particular and as an example of what we just said, we can observe that the maximum increase of accuracy from having one node to have two is 42%. On the contrary, the increase of accuracy from having fifty nodes to have 100 is 0.02%.

Another conclusion that we can deduce is that from having twenty nodes for each reservoir, we start to achieve an excellent accuracy. Furthermore, we also can say that when our network has more than two hundred nodes (a hundred nodes for each partial reservoir) the accuracy is almost perfect for both classifier, and therefore, the results are stabilized.

In addition, in the following figures you can see that for each classifier, a graph showing the evolution of the accuracy of the network as a function of the number of nodes.

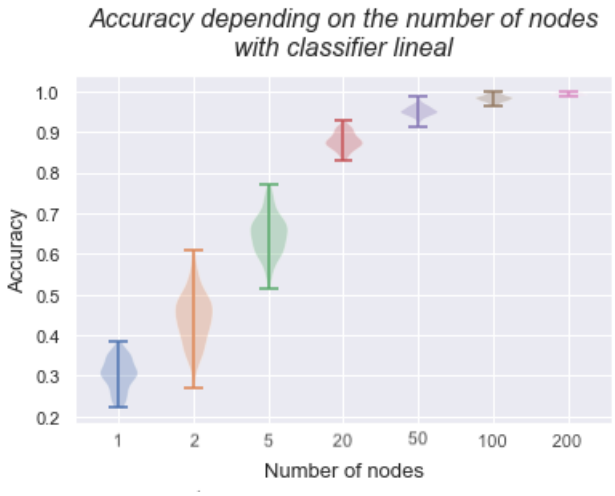

Figure 2.7: Evolution of accuracy of the network depending on the number of nodes with join_probability=0 and using lineal classifier

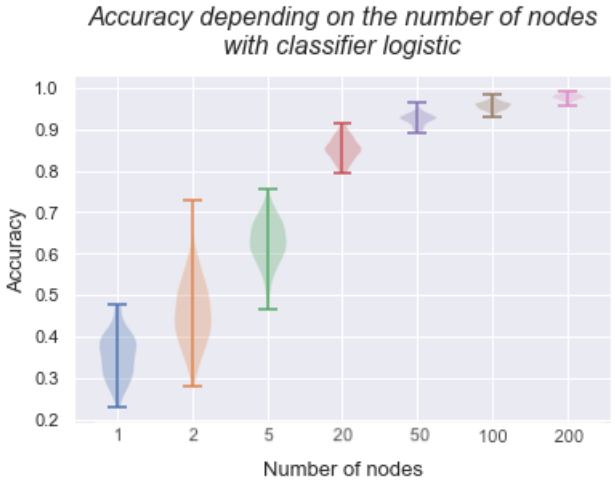

Figure 2.8: Evolution of accuracy of the network depending on the number of nodes with join_probability=0 and using logistic classifier

To sum up and as we expected, the higher number of nodes, the higher accuracy of the network. Moreover, we also found out that the maximum performance is reached asymptotically around two hundred nodes, and that it is achieved by following a logarithmic function. Finally, at this point of the network' study, we can remark that the two classifiers obtain very similar results, but the linear classifier obtains, in general, more accurate results.

2.5.2 Study of join_probability

Up to this point, we have studied the accuracy without any connection between the two partial reservoirs, the motor and the premotor. From now on, we are going to connect them, i.e. we are going to vary the join_probaility to see how it influences the accuracy of the network.

To make the graphs you will see below, we have set three of the four hyperparameters: num_nodes, input_probability and reservoir_probability. It is important to note that the values we chose to set the input_probability and reservoir_probability are chosen from the previous heatmaps. To find these, we look for each number of nodes, the probability values for which the highest accuracy has been obtained. In this way, in each graph, we will observe the accuracy of the network as a function only of the join_probability. In this case, we will also test first for the linear classifier and then for the logistic classifier.

Studying the join_probability using linear classifier

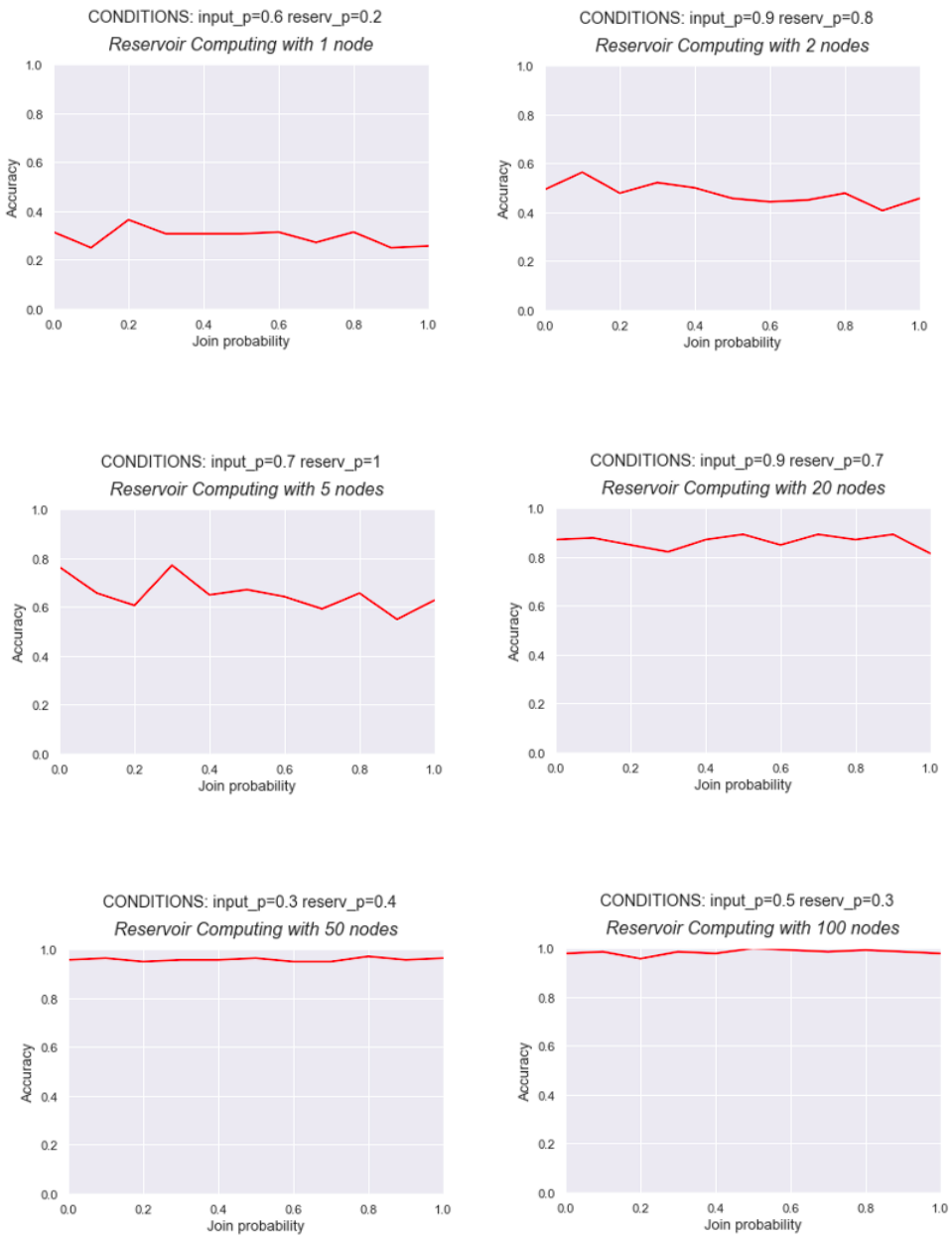

Figure 2.9: Network's accuracy in terms of join_probability and using linear classifier

Studying the join_probability using logistic classifier

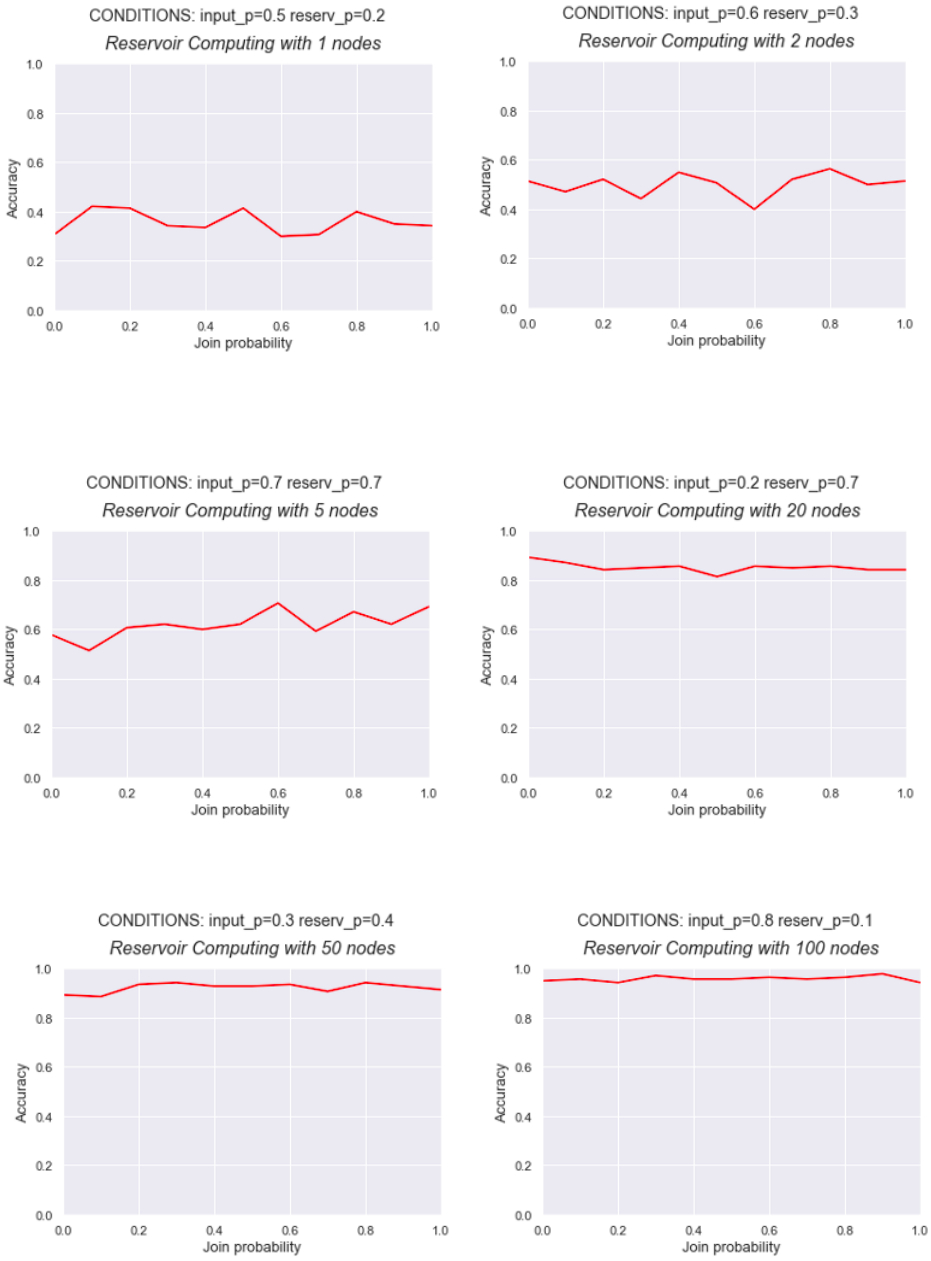

Figure 2.10: Network's accuracy in terms of join_probability and using logistic classifier

As before, we observe that with both classifiers we achieve very similar results, so we are going to comment the graphics in general. At a glance, join_probability doesn't seem to have much influence in the accuracy. As we can see, the results of most of the graphs vary around a fix accuracy, but without following a clear direction of growth.

On the other hand, and in the same way than with the number of nodes, the evolutions of the accuracy in concordance with the join_probability have a logarithmic increase. In the following graph you will observe how the accuracy evolves as a function of the number of nodes and the join_probability.

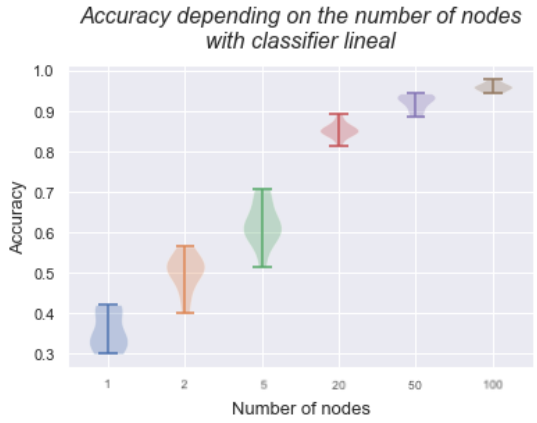

Figure 2.11: Evolution of accuracy of the network depending on the number of nodes with join_probability!=0 and lineal classifier

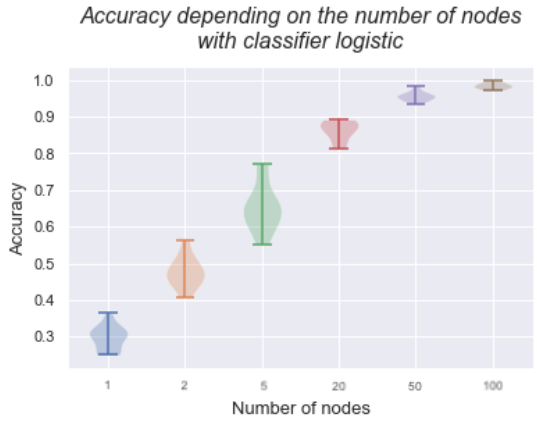

Figure 2.12: Evolution of accuracy of the network depending on the number of nodes with join_probability!=0 and logistic classifier

2.5.3 Comparison of network with and without join_probability

Once we have studied which hyperparameters fixed the best for the accuracy depending on the number of nodes, we will proceed to compare the results of the modularized network without reservoir's connexion with the one that has.

This comparison will help us to solve our hypothesis about join_probability. Since from the last graphics we have suspected that connecting the two reservoirs doesn't make a very significant difference in the accuracy of the network. To develop the table, we are going to select individually the better input_probability, reservoir_probability and join_probability from the graphs above. And again, we are going to distinguish the use of the classifier; first we are going to do the comparison with linear and then with the logistics.

General network conditions			***Modularized network*** *(without join_p)*	***Modularized network*** *(with join_p)*	
NUM_NODES	INPUT_PR	RESERVOIR_PR	ACCURACY	JOIN_PR	ACCURACY
1	0.6	0.2	0.38571429	0.2	0.36428571
2	0.9	0.8	0.55	0.1	0.56428571
5	0.7	1.0	0.77142857	0.3	0.77142857
20	0.9	0.7	0.92857143	0.5	0.89285714
50	0.8	0.7	0.98571429	0.6	0.98571429
100	0.7	0.2	1.0	0.6	1.0

Table 2.1: Comparative table depending on the join_probability using linear classifier

General network conditions			***Modularized network*** *(without join_p)*	***Modularized network*** *(with join_p)*	
NUM_NODES	INPUT_PR	RESERVOIR_PR	ACCURACY	JOIN_PR	ACCURACY
1	0.5	0.2	0.50714286	0.1	0.42142857
2	0.6	0.3	0.62857143	0.8	0.56428571
5	0.7	0.7	0.75714286	0.6	0.70714286
20	0.2	0.7	0.91428571	0.1	0.8928571
50	0.3	0.4	0.96428571	0.3	0.94285714
100	0.8	1.0	0.98571429	0.9	0.97857143

Table 2.2: Comparative table depending on the join_probability using logistic classifier

Looking at the two tables above and as we suspected, we can affirm that connecting the reservoirs does not help to obtain greater accuracy in the network. But rather we obtain very similar accuracies, or even worse results when using the logistic classifier. When we developed the idea of join_probability, we thought that it would have had a crucial role in the network, but these data dismantle the first hypotheses, since we expected that connecting the reservoirs would lead to better accuracy and it is not that way.

2.5.4 Comparison of modularized and non-modularized network

At this point, we have observed that the join_probability is not a significant hyperparameter for the accuracy of the network. Consequently, we decided to compare the results of the modularized network with join_probability=0 with the results of the unmodularized network of Arnau Naval [Naval2020]. Next, we show two tables very similar to the previous ones, where we can see if the modularization really influences the accuracy.

As we explained at the beginning of the work, the modularized network has twice times the number of nodes than in non-modularized network. For this reason and because both networks play on equal terms, we have trained and have tested the non-modularized network with the double of nodes.

General network conditions		*Modularized network without join_p*		*Non-modularized network*	
INPUT_PR	RESERVOIR_PR	NUM_NODES	ACCURACY	NUM_NODES	ACCURACY
0.6	0.2	1	0.38571429	2	0.29285714285714287
0.9	0.8	2	0.55	4	0.44285714285714284
0.7	1.0	5	0.77142857	10	0.6214285714285714
0.9	0.7	20	0.92857143	40	0.8642857142857143
0.8	0.7	50	0.98571429	100	0.9357142857142857
0.7	0.2	100	1.0	200	0.9785714285714285

Table 2.3: Comparative table about results of modularized network and non-modularized using linear classifier

General network conditions		*Modularized network without join_p*		*Non-modularized network*	
INPUT_PR	RESERVOIR_PR	NUM_NODES	ACCURACY	NUM_NODES	ACCURACY
0.5	0.2	1	**0.50714286**	2	**0.5**
0.6	0.3	2	**0.62857143**	4	**0.6214285714285714**
0.7	0.7	5	**0.75714286**	10	**0.6785714285714286**
0.2	0.7	20	**0.91428571**	40	**0.8642857142857143**
0.3	0.4	50	**0.96428571**	100	**0.9285714285714286**
0.8	1.0	100	**0.98571429**	200	**0.9571428571428572**

Table 2.4: Comparative table about results of modularized network and non-modularized using logistic classifier

This time, we can observe that the modularized network obtains better results in all cases and with any of the classifiers. Just as we hypothesised, modularize the network is a good way to improve the accuracy. And not only that, we can add that in the best case, the accuracy improves by up to 0.15 points, so we get a percentage increase of around 20%. And in the worst case, the accuracy improves by up to 0.07 points, that is equivalent to a 0.13% of increase.

Conclusions

The aim of this project was to study and characterize the influence of modularity on the neural dynamics of reservoir computing networks. At the very beginning of the work, we asked ourselves if modular reservoir network could outperform a single module one. This is made evident on the previous testing of the spcrit, where we have observed better accuracy with modularized network. Though the comparative table of the network's results with several reservoirs against the basic ones, we can conclude that the accuracy improves by up to 20% in the best case, and 2% in the worst cases.

Additionally, another hypothesis that we established from using non-modularized network was to assume that the hyperparameter of the number of nodes would grow logarithmically. In short, we expected the greater number of nodes, the greater accuracy of the program, and that from a certain value onwards it would stabilise. This aspect makes a lot of sense, since the learning process in our brain involves a large number of neurons, which would be our equivalent to a node. Therefore, it is easy to deduce that with a single node we do not obtain much accuracy, but with a big number of them we yield a notable improvement.

On the other hand, we formulated another operational hypothesis related with the classifier. Based on the results of non-modularized network, we thought that both classifiers would yield fundamentally similar accuracy. Although the logistic classifier uses ordinal encoding and the linear one-hot encoding, both classify using few computational resources while obtaining comparable results.

Finally, and once the network was modulated, the need arose to create the hyperparameter join_probability. This was in charge of the connection between the two reservoirs, and we assumed that it would play a key role in accuracy. Nevertheless, not only did not, but we found out that in certain cases its influence decreased the accuracy of the network by up to 17%. Hence, we deduced that the connection probability between the nodes of the reservoir itself and the connection probability of the reservoir to the input layer were much more relevant factors.

To conclude the project, we can affirm that it has been worthwhile studying the impact of brain-like modular reservoirs in the representation of brain states characterized by temporal series recorded from separate cortical areas.

Bibliography

[1] Michael DePass and Ignasi Cos, *Characterization of movement-related neural states in nhp's*, 2019.

[2] H. Jaeger, *The "echo state" approach to analysing and training recurrent neural networks-with an erratum note.*, Research Center for Information Technology GMD Technical Report, Bonn, Germany, 2001.

[3] Vito Latora, Vincenzo Nicosia, and Giovanni Russo, *Complex network*, Cambridge University Press, United Kingdom, 2017.

[4] Natschlager T. Markram H. Maass, W., *Real-time computing without stable states: A new framework for neural computation based on perturbations.*, Neural computation, 2002.

[5] Arnau Naval Ruiz and Ignasi Cos, *Revealing brain states with reservoir computing*, 2020.

[6] Andrea Roli, *Introduction to reservoir computing methods*.

[7] Nuria Sanchez Font and Oriol Pujol Vila, *Reservoir computing for learning the underlying dynamics of sequencial data points*, 2020.

[8] D. Schrauwen, B. Verstraeten and J. Van Campenhout, *An overview of reservoir computing: theory, applications and implementations. in proceedings of the 15th european symposium on artificial neural networks*, 2007.