Evolution of Modular Neural Networks

by

Victor Manuel Landassuri Moreno

A Thesis submitted to the
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computing Sciences
College of Engineering and Physical Sciences
University of Birmingham
September 2011

Abstract

It is well known that the human brain is highly modular, having a structural and functional organization that allows the different regions of the brain to be reused for different cognitive processes. So far, this has not been fully addressed by artificial systems, and a better understanding of when and how modules emerge is required, with a broad framework indicating how modules could be reused within neural networks. This thesis provides a deep investigation of module formation, module communication (interaction) and module reuse during evolution for a variety of classification and prediction tasks. The evolutionary algorithm EPNet is used to deliver the evolution of artificial neural networks. In the first stage of this study, the EPNet algorithm is carefully studied to understand its basis and to ensure confidence in its behaviour. Thereafter, its input feature selection (required for module evolution) is optimized, showing the robustness of the improved algorithm compared with the fixed input case and previous publications. Then module emergence, communication and reuse are investigated with the modular EPNet (M-EPNet) algorithm, which uses the information provided by a modularity measure to implement new mutation operators that favour the evolution of modules, allowing a new perspective for analyzing modularity, module formation and module reuse during evolution. The results obtained extend those of previous work, indicating that pure-modular architectures may emerge at low connectivity values, where similar tasks may share (reuse) common neural elements creating compact representations, and that the more different two tasks are, the bigger the modularity obtained during evolution. Other results indicate that some neural structures may be reused when similar tasks are evolved, leading to module interaction during evolution.

To my family, who supported me during all this time and though me that the best kind of knowledge to have is that which is learned for its own sake.

Acknowledgements

This research project would not have been possible without the support of many people. First, and foremost, I would like to thank my supervisor John A. Bullinaria who was abundantly helpful and offered invaluable assistance, support and guidance: thank you very much John. I look forward to our future collaboration and continued friendship. Deepest gratitude are also due to my *research monitoring group*, Dr. Xin Yao and Dr. Ata Kabán without whose knowledge and assistance, this study would not have been successful. I am also indebted to Dr. Larry Bull for making different suggestions to improve this study.

The author wishes to express his love and gratitude to his beloved families; for their understanding and endless love, through the duration of his studies. Thank you Cris, Alex, Luz, Miguel and Alejandro for your unconditional support.

Special thanks also to Monica for listening to me whenever I needed, the lively discussions and the help along the way, and all my colleagues and graduate friends, my stay in Birmingham will not be complete without you. Not forgetting to my best friends in Birmingham who always been there, thank you Luis, Margarita, Carolina, Leandro, Jose, Veronica and Rehana.

I sincerely thank my former research group in Mexico who helped me to switch off and relax sometimes during the course of this studies (particularity at the end), also for their encouragement during all this journey, thank you very much Dr. Figueroa, Carmen and Lydia.

The author would also like to convey thanks to the Mexican Council of Science and Technology (CONACyT) for the scholarship granted: 182657/303256. Last but not the least, I would also like to thank the School of Computer Science Research Committee for the travel grants I was allocated for attending to workshops and conferences, and to the School of Computer Science and the University of Birmingham through the resources given to carried out this research.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence	3
ANN	Artificial Neural Network	1
AR	Autoregressive	53
ARMA	Autoregressive Moving Average	60
BP	Back Propagation	3
EA	Evolutionary Algorithm	1
EANN	Evolutionary Artificial Neural Network	1
EP	Evolutionary Programming	18
EPNet	Evolutionary Programming of Artificial Neural Networks	9
ES	Evolutionary Strategies	18
FS-EPNet	Feature Selection EPNet	11
FS$_{AD}$**-EPNet**	Feature Selection with asymmetric delays EPNet	11
GA	Genetic Algorithm	3
GMLP	Generalized Multi-layer Perceptron	23
GP	Genetic Programming	5
HDANN	Hand designed artificial neural network	60
LCS	Learning Classifier System	45

MSE	Mean-square Error	71
MBP	Modified Back Propagation	24
M-EPNet	Modular EPNet	10
MNN	Modular Neural Network	1
MA	Moving Average	53
MLP	Multi-layer Perceptron	21
MSP	Multi-step Prediction	8
NEAT	Neuroevolution of Augmenting Topologies	17
NMSE	Normalized Mean Squared Error	59
NRMSE	Normalized Root Mean Squared Error	59
RMSE	Root Mean Squared Error	60
RNN	Recurrent Neural Networks	57
SA	Simulated Annealing	20
SSP	Single-step Prediction	8
STP	Successful Training Parameter	86
TS	Time Series	8

Chapter 1

Introduction

Artificial Neural Networks (ANNs) and Evolutionary Algorithms (EAs) have been widely inspired by biological organisms, usually giving them superior performance when both are applied together to solve a problem than when they are applied in separate stages. Moreover, Modular Neural Networks (MNNs) can extend the performance of conventional networks, borrowing ideas from the modularity present in the human brain. In this context, modularity is known to have benefits for neural systems and their evolution, like faster learning or the reuse of neural elements in both structural and functional ways. Even though those benefits, it is clear that the human brain is more than the sum of its parts, but that is not currently true of ANNs nor of MNNs.

This thesis is aimed, therefore, at exploring in more depth the evolution and reuse of modular neural networks in different cases and scenarios, in order to gain a better understanding of when and how modules interact between themselves. This will certainly be helpful to drive further implementation of Evolutionary Artificial Neural Networks (EANNs) in more intelligent systems.

This chapter begins with a further explanation of how ANNs and EAs have been motivated by nature, followed by different advantages provided by modularity. Thereafter, the research approach of this study is explained, preceded by the core research questions that motivate it. Hence, the contributions and publication derived from this thesis are shown before it is outlined the structure of the dissertation.

1.1 Nature inspired process

The human brain is known to be one of the most complex systems in nature and the most remarkable in the human body, performing thousands of operations per second over different kind of tasks in a parallel approach. It has evolved over millions of years in an environment restricted by natural resources like space and energy among others, and its actual organization is clearly a function of those resources, including the kind of tasks it faces.

Space and energy may have an interesting relationship in the living organism, as bigger individuals (more space) may require more energy to survive. In the case of the human brain one might be forgiven to thinking that a long connection between two neurons will require more energy to transmit the information than if the connection is shorter (i.e. with a smaller physical distance between them), without considering the attenuation caused by a larger distance [1, 2].

For several years, neuroscientists have investigated the human brain, and it is now widely appreciated that it is highly modular [3–5]. Thus, a *module* could be seen as a set of closely interconnected neurons sparsely connected with neurons from other modules, that is, it is expected that nodes of the same module will have several connections between them while there are just a few connections between nodes of different modules. Clearly this structural definition of modules fits with the optimization of finite resources in nature (remarked in the previous paragraph), which makes us think that we are going in the right direction.

Besides the structural approach, there are diverse studies and theories indicating that the brain reuses neural circuits for different cognitive purposes (functional organization of the brain) [3, 4, 6, 7], i.e. the same module (structural) may be reused several times on different tasks (functional). Thus, modules may by made up of cortical regions in the brain which are not only anatomically neighboring but also by functionally related [3].

More important is the fact remarked by Anderson [4], where there is no current doubt whether there is significant, widespread, and functionally relevant module reuse in the human brain, but how that reuse is carried out is a current open question. Even though there is no discussion on whether neural reuse is beneficial in the brain [4] as a feature of brain organization, for computer scientists, more specifically those working in the Artificial Intelligence (AI) field, the same has not been demonstrated for artificial systems.

Several studies have been carried out to apply the features already known from the human brain to artificial systems to make them more autonomous, among other advantages. Much of this work, to simulate a small structural and functional part of the human brain was influenced by the pioneering development over ANNs by the neuro-physiologist Warren McCulloch and the mathematician Walter Pitts in 1943. Later, development of similar ideas from Werbos, Parker and Rumelhart led to the re-discovery of the Back Propagation (BP) algorithm by Rumelhart et al. in 1986 [8] allowing us to train the networks to acquire some kind of "knowledge". Currently, ANNs are widely understood and powerful tools in control, classification and prediction tasks.

Nevertheless, looking for the optimal network architecture that solves a problem is a challenging task that may depend on the designer's knowledge (expertise) or an optimization technique like EAs [9, 10]. Here EA uses a population-base metaheuristic to look for optimal solutions based on mechanisms inspired by biological evolution, i.e. they are based on Darwin's theory of evolution, trying to copy the processes carried out in nature, evolving living organisms and applying them to problems that currently are impossible to solve in polynomial time (NP-complete problems). Therefore, the use of an EA, like the popular Genetic Algorithm (GA), could be beneficial in the search for (optimization of) a suitable set of parameters of neural networks (number of nodes, connections, etc.) that allows an accurate solution. Interestingly, any network parameter can be subjected to the evolutionary process, but the search space grows as more parameters are involved, making it more difficult to find an acceptable solution. Note that both approaches (ANNs and EAs) have been inspired by nature and they could deliver more robust systems than if ANNs or EAs are applied separately to the problem [10].

Even though ANNs can be adapted automatically with EAs to solve diverse problems, there remain a huge amount of work to simulate a small region of the human brain. Moreover, ANNs are just focused on solving a simple task. Thus, MNNs can be considered the next step from artificial networks where the aim is to solve more than one task at the same time, focusing on module emergence [11, 12] or on automatic problem decomposition [13, 14]. Regarding module reuse, [15] has introduced a modular algorithm that reuses neural substructures automatically over a board game domain. Nevertheless, until now it has not been clear how we may develop modules that may be reused in subsequent tasks, and most of the work carried out in modular architectures has been in the investigation of two compound tasks [12, 13, 16, 17]. Clearly, the brain is used to performing several tasks at the same time; thus, probably we are missing some insights from that process when limiting the study to only two tasks.

1.2 Advantages of neural modularity

As remarked above, neural reuse is a fact of the human brain, but until now there has not been an artificial approach that takes advantage of all the benefits of modularity as presented in the brain. More recent work done by Bullinaria [12, 18] has indicated that modules may emerge (during evolution), if certain conditions are present: if there is interference in the learning process caused by the differences between two tasks evolved simultaneously; and if the number of connections is constrained during the evolution, similar to the limited connectivity found in the brain, which is known to be less than half of the maximum possibly allowed [2]. Elsewhere, modular networks have been employed with the aim of automatic problem decomposition [13, 14], although the main potential of modularity may not be seen by problem decomposition, because it can be assumed that it is a basic requirement to have modular systems. Other potential advantages are:

a) Learning complex classification problems in a reduced amount of time as presented in [12].

b) Reusing neural substructures by problem decomposition as shown in [15, 19] for the board game domain.

c) Good Generalization [20] in complex multicomponent language processing tasks or handwritten digits data set [21].

d) Coping with changing environment [19, 22–25].

e) More generally, robustness and evolvability [26].

Overall, those insights can contribute to a better understanding of the human brain [11, 12, 15, 19, 21]; meanwhile they help to improve our computational techniques.

Even where there is information indicating that modularity is beneficial, there is also evidence that sometimes non-modular architectures are better for solving problems [12]. For that reason more evidence is needed as well as computational techniques assisting with module formation and module reuse. For example, it is necessary to obtain information that allows more autonomous systems, in the sense that they evolve modules as required without wasting resources (reusing); or else to discover the complex relationship between structural and functional reuse to develop modules that could be used as building blocks for bigger representations. Note that building blocks have been clearly developed for Genetic Programming (GP) [27–31] and during software engineering, where a piece of code is reused several times for different tasks. For example, a function designed to read a file (software engineering) can be used in a bigger algorithm to read a TS and forecast it, to read a configuration file and set up parameters or to load information for a data-base. If we talk about object oriented programming, multiple copies of the method is made at run-time to be reused in different cases. Interestingly, we do not yet know if that may be possible with ANNs (developing modules as building blocks) but the more information we can obtain the better models we would develop.

We may also consider whether or not to reformulate our definition of modules for artificial systems, as commonly it refers to independent partition of nodes, which means to have pure-modular architectures [12, 17]. Thus, we might well obtain a better understanding of this problem if a relaxation of this definition were adopted (e.g. module communication [32]), as it is well known that modules exist in the brain but certainly they may communicate between them. For example, consider [3] indicating that modules may exist inside modules. If they exist at different levels, we may wish to keep in mind the idea that they are interconnected at similar levels or in bigger containers (also modules).

The major difficulty in this case, is that currently we do not know how modules are developed in the brain and for which function they emerge, nor how they are interconnected (wired) to interact between them. Thus, the knowledge that we already now about the large module reuse that take place into the brain, clearly is justified by the fact that it is constrained by resources and by the tasks it processes, making the module reuse a fundamental process of the brain.

1.3 Research questions

Considering the previous studies and the advantages that modularity could bring, there remain many and various questions to answer. The following are the core questions of this study.

Taking into account that previous studies have evolved only two tasks at the same time, focusing mainly on the emergence of pure modular architecture, there is the question whether evolving several related tasks could lead to the evolution of modular architectures, avoiding interferences caused by small differences between them and reusing several neural components, which clearly will allow smaller representations (but considering that the performance is no worse than if they were solved individually). From there it is also interesting to consider in which cases modular and non-modular architectures emerge and why. It is expected that the more different two tasks are, the bigger the modularity obtained during evolution, but how different do they have to be to always produce independent partition of nodes?

Considering that previous results have pointed out that sometimes non-modular networks produce better results than modular ones, the question is why that could happen. Because of the computational power provided by having more connections? Is it because the interference during learning could be absorbed by the huge number of connections? Or is it simply because they were quite similar, so that a homogeneous architecture could solve them.

To explore module emergence and module reuse, clearly we need an EA that helps in the evolutionary part. However, would it be possible to find a more intelligent way to perform the evolution, than by relying on the stochastic operators usually adopted? Clearly the brain is constrained by different factors and therefore biased in some way to form modules, which makes us think that stochastic processes do not play a key role in the module formation. Another requirement for this study is the use of an algorithm that avoids the permutation problem presented during evolution (explained in Section 1.4). Thus, it could be possible to have an algorithm that evolved suitable modular architectures with just mutation operators? If not, might it be possible to adapt this kind of algorithm for the module evolution?

Assuming we could have a suitable algorithm to evolve modular architectures, if one experiment evolves n tasks independently and the second experiment uses them in combination to evolve MNNs, could the single networks evolved for each task have a similar structure (number of nodes and connections) and performance to those evolved with the modular algorithm? Whether or not this assumption is valid, could it be true for all tasks tested during this study, or might a subset of them have a better performance than others if they are evolved separately? Clearly those questions may help to answer in which cases modularity is beneficial for the tasks used here.

Further questions can be raised by the fact that there are different ways to carry out forecasting, e.g. Single-step Prediction (SSP) and Multi-step Prediction (MSP). It has been found that MSP is more difficult to perform than SSP (Section 3.2). Moreover, classification tasks are performed in the same way as SSP, and previous publications have evolved MNNs for classification tasks; thus, is it possible to evolve modules for tasks that use the SSP method as was done for classification tasks, even if they are a different kind of problems? Clearly it will be a bigger challenge to do the same for MSP tasks, but might reliable predictions be possible if modular networks are evolved for tasks using the MSP method? Note that as far as this research has been carried out, no work has been found that uses MNNs to predict Time Series (TS) as done in this study.

The last and perhaps the most important question is regarding the module reuse, could it be possible to reuse a module previously evolved to solve the same task or another similar? So far no study has demonstrated nor explained if that is possible with ANNs. Moreover, previous attempts to reuse modules have been missing a deeper explanation of which kind of modules are reused and for which tasks.

1.4 Research approach

The first step to start this study is at evolving ANNs and obtain a baseline results that later could be compared against the modular approach. One intention here is to use an algorithm that avoids the permutation problem [33], mainly caused by crossover operators (explained in Section 2.1.1) and affecting the evolution of ANNs. Moreover, the connectivity patterns in MNNs are more specific that in ANNs, i.e. in normal networks any connection can appear between two nodes, but in modular architectures clearly it will be desired to have less number of connections between two modules. Therefore, if the permutation problem may affect ANNs where any connectivity pattern is allowed, it can be expected that this issue affects too modular architectures. Note that this aspect is out of the scope of this study but clearly was considered to chose an algorithm. Nevertheless, recombination operators could be a good tool in combination with mutation operators to evolve any other numerical parameter (e.g. learning rate) in neural networks [12, 34]. Hence, the crossover could help in some cases but could introduce noise during evolution in others.

Taking into account this, it has found that the Evolutionary Programming of Artificial Neural Networks (EPNet) algorithm, introduced by Yao and Liu [35], does not use crossover operators because of the permutation problem. Therefore, addressing the evolution and reuse of MNNs with this kind of algorithm, will allow us to investigate whether it can make use of the advantages provided by modularity and to explore whether it is possible and functionally adapt it.

At an early stage (Chapter 4), the EPNet algorithm is studied and optimized over different parameters to enable us to have confidence in its behaviour. Thereafter, it was noticed that it had not been used to tackle feature input selection during evolution, an essential characteristic needed for module evolution. Hence, this aspect of the EPNet algorithm has been improved to make it a more suitable algorithm with more general applicability, where the features were evolved using the existing mutation operators in the algorithm to avoid unnecessary and more complex implementation in terms of mutation operators. This was performed taking into account that a new mutation operators would be introduced for the module evolution, as explained in the following paragraphs.

Neural networks exist with varying degrees of modularity, ranging from pure modular networks, characterized by disjointed partitions of hidden nodes with no communication between modules, to pure homogeneous networks with significant connections throughout. In between are apparently homogeneous networks that can be seen to have some degree of modularity if the hidden nodes are analyzed appropriately, i.e. modules are formed (having some interaction in between), and to a certain extent, module reuse of some neural structures. To address modular representation within the improved EPNet algorithm, a modularity measure is presented and extended in Chapter 5 that can be applied to any neuron at any level in the network to provide a fine-tuned analysis of node partitioning. It also allows the rearrangement of nodes to create modules in homogeneous networks. Moreover, the information provided by the modularity measure has been used to identify which element contributes to each module and this information has been translated into new mutation operators that favour the evolution of modular architectures.

In this sense, the modularity measure allows us to quantify (during evolution) the contribution of each node to each module taking into account the connectivity patterns. This information can automatically be translated to the identification of *shared nodes* and *shared connections*, which are neural components that contribute to more than one module; or, from another point of view, elements that help in communication between modules. Hence, the information was used to create mutation operators that delete such neural components. The new mutation operators with the modularity measure provide a different way to analyze module formation and module communication during evolution, and they deliver the Modular EPNet (M-EPNet) used to investigate the core idea of this study.

The key aspect to the evolution of modules in this study is based on the connectivity level and the interference produced by two tasks during learning (weight update) as discussed by [12, 16, 18, 36] in artificial systems or as previously concerning over the human brain [4, 6, 7]. Therefore, the principal idea of module evolution is mainly performed when two or more tasks are solved (evolved) simultaneously (Chapter 5). On the other hand, as there is no fundamental constraint that requires solving two or more task at the same time, module emergence during evolution was also investigated in single tasks in Section 5.4 applying an extension of the modularity measure (Section 5.1.4).

In the last stage (Chapter 6), the M-EPNet algorithm was used with different and related tasks at the same time (for module formation) and at different stages (for module reuse) to investigate whether modules communicate in a bigger representation. This allows measurement of the level of neural reuse in incremental neural systems at different levels and stages, i.e. the system learns different tasks at different stages (times), incrementing the overall architecture as new tasks arrive to be solved. Not surprisingly, this is another feature borrowed from nature and included in this study, where the human brain is though to develop new connectivity patterns (including modules) as it is influenced by new experiences (tasks).

To test all this ideas and algorithms, several prediction and classification tasks have been used to evolve ANNs and modular architectures. Moreover and as explained above, there are two general ways to forecast a TS: the MSP and SSP methods. Hence, both methods have been used on prediction tasks during this study.

1.5 Thesis contributions

The main objective of this thesis is to empirically investigate neural network module formation through evolution and the advantages that modularity brings in terms of module reuse. Hence, the following contributions are presented in the order they appear in this study, i.e. first concerning the EPNet, later the M-EPNet, and finally the module reuse.

The EPNet algorithm is studied and extended to tackle feature selection during evolution using the existing mutation operators. From there, two variations of the algorithm are developed, Feature Selection EPNet (FS-EPNet) and Feature Selection with asymmetric delays EPNet (FS$_{AD}$-EPNet), to solve prediction and classification, showing robustness (i.e., better performance) of these algorithms when compared with previous studies. Also, a new method is introduced to create an extra data set for evaluating the validation set in the same terms as the generalization performance is measured for tasks requiring the MSP method.

M-EPNet algorithm: an appropriate modularity measure is developed that enables the discovery of node dependency on a given output partition (or module) in a manner that allows improvements over previous studies. Here, it is used to discover cross-connections between nodes from different modules, or from another angle, neural components that contribute to more than one module or output partition. This allows the rearrangement of nodes into modules for a clearer graphical representation of the final modules evolved, which enables a different way to analyze the evolved networks. The fact that the modularity measure was previously developed for networks with more than one output unit, leads to another contribution, where the modularity measure is extended to networks with a single output unit. Finally, the M-EPNet algorithm is obtained when the information from the modularity measure is translated into three new mutation operators to evolve modular neural networks, where the new operators produce low connectivity levels between modules during evolution, while they encourage the increase of intra-modular connectivity.

This approach differs from previous studies in the sense that it does not constrain the number of nodes or connections directly, i.e. they are constrained by the sequence of applied the mutation operators, thus no extra metrics are required for that purpose. Nevertheless, some constraints are imposed to study similarities with the human brain. Also, the module communication during evolution is allowed, which means that networks are not constrained to pure-modular architectures. That allows a more natural way to evolve the modules with the M-EPNet algorithm.

Module reuse: because the M-EPNet algorithm was found to evolve more independent partitions of nodes than the EPNet algorithm, it was used in the final stage of this study to demonstrate that modules previously evolved can be reused efficiently when a new task needs to be performed. Also, a simple metric is presented that proves to be a useful measure of the level of module reuse in terms of nodes reused from other tasks. This is an important result as previous studies assume the neural resources are reused, but no evidence is provided, nor explanation of what kind of modules emerged and how they were reused for particular sub-tasks. This study also contributes to the knowledge about how and when modules emerge in different compound tasks and when modules are reused in simple experimental set-ups. Thus, the M-EPNet algorithm provides a powerful basis for future studies of modularity, both for real world applications, and for understanding biological systems.

1.6 Publications resulting from the thesis

There have been several refereed publications resulting from this study in addition to the contributions previously stated:

Chapter 4:

V. Landassuri-Moreno and J. A. Bullinaria. Feature selection in evolved artificial neural networks using the evolutionary algorithm EPNet. In *Proceedings of the 2009 UK Workshop on Computational Intelligence (UKCI)*, 2009.

V. Landassuri-Moreno and J. A. Bullinaria. Neural network ensembles for time series forecasting. In Franz Rothlauf (editor), *Genetic and Evolutionary Computation Conference 2009 (GECCO)*, pages 1235–1242. ACM, 2009.

Chapter 5

V. Landassuri-Moreno and J. A. Bullinaria. A new approach for incremental modular neural networks in time series forecasting. In *Proceedings of the 2008 UK Workshop on Computational Intelligence (UKCI)*, 2008.

V. Landassuri-Moreno and J. A. Bullinaria. Biasing the evolution of modular neural networks. In Alice E. Smith (editor), *2011 IEEE Congress on Evolutionary Computation (CEC)*, pages 1952–1959. IEEE Computational Intelligence Society. IEEE Press, 2011.

1.7 Overview of the dissertation

This thesis is divided into three main parts: background (Chapters 2 and 3), a description of algorithms and their evaluation (Chapters 4, 5 and 6) and finally, conclusions and further improvements in Chapter 7.

Chapter 2 reviews prior work concerning the modularity in biological systems and different modularity measures used in neural networks. Also, it introduces different previous studies involving ANNs, MNNs and EAs. A comprehensive description of the EPNet algorithm is given with similarities to others algorithm like the NEAT and modular NEAT.

Chapter 3 starts describing the prediction and classification problems, with the different tasks used throughout this study to test the evolution of single and combined tasks to evolve ANNs and MNNs respectively. Also shown are different prediction methods (SSP and MSP), the similarities between SSP and classification tasks, and different performance measures used in prediction and classification tasks.

Chapter 4 presents a detailed study of the key parameters of the EPNet algorithm and the improvement it offers over input feature selection during evolution, something that was essential for its later improvement for MNNs. The training, validation and testing sets are presented with an explanation of the extra test set used to measure the fitness during evolution for tasks that require the MSP method. At the end of the chapter can be found a comparison with previous work in the literature to show the robustness of the algorithm at finding correct network parameters.

Chapter 5 explains the modularity measure used in this study to test how close a network is to being pure-modular or pure-homogeneous. The modularity measure allows the measurement of a dependency value that was used in this work to extend the EPNet algorithm into its modular version, the M-EPNet algorithm, with the introduction of new mutation operators aimed to delete shared neural components that contribute to more than one module (output partition) and to increase the intra-module connectivity. As the modularity measure was originally designed for networks with more than one output unit, its extension is also presented in Chapter 5 to be applied with networks with a single output unit. That allows us to apply it to any node at any level in the network (fine-tune measure), and to study the modularity of networks during evolution for single tasks, like the Lorenz TS prediction.

Different combined tasks results are presented for cases in which the tasks solved are similar and when they are more different to study where module emerge. The combined tasks are also evolved with two or three tasks at the same time to investigate if the more tasks solved simultaneously the more interference generated during learning. As previous studies have pointed out, in the human brain the connectivity is known to be limited [2], causing MNNs make the most of the available connections [12], thus, here was investigated the effect of evolving MNNs with and without connectivity constrained during evolution. The results of the M-EPNet algorithm are compared with the same simulation carried out with the standard EPNet algorithm to illustrate the benefits of using an algorithm designed to preferentially evolve modules.

Chapter 6 explains how the different tasks already learned were reused to solve new tasks, i.e. simulating that a new task arrives to the system and reusing previous neural structures evolved. It is also introduced a measure that could be implemented (using the modularity measure) to quantify the number of nodes reused to solve the current task.

Chapter 7 contains the final conclusions and suggest further improvements of the EPNet and M-EPNet algorithms for evolving neural networks: modular and non-modular.

Appendixes A provides extra information about tables and figures from Chapter 4.

Chapter 2

Background

As noted in Chapter 1, this study is aimed to study more deeply the module formation with neural networks. More important is to have a method that allows us to discover modules for their further reuse in more complex implementation. Hence, the EPNet algorithm was used as the base algorithm to deliver the evolution of networks in this thesis, because it avoids the permutation problem and because it allows a natural extension with potential to evolve MNNs.

Considering this and the current research questions that motivated this study, this chapter will provide the background material and literature review related to ANNs, MNNs, modularity in biological and artificial systems, and the evolution of them through EAs. Moreover, it will explain the basis of the EPNet algorithm for its further improvement in later chapters, and related algorithms like the modular Neuroevolution of Augmenting Topologies (NEAT) algorithm that has been used to evolve and reuse modules previously. Despite the attempts of module formation and module reuse with the modular NEAT, it has not found any previous study that clearly demonstrate that module reuse is possible and beneficial with neural networks.

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) were inspired by biological evolution, as remarked previously, and considered metaheuristics because they are iterative methods which attempt to optimize a problem through the improvement of candidate solutions. While they do not guarantee that an optimal solution is ever found, they are the only methods that can tackle NP problems by creating a search space for potential solutions, where each solution can be tested in polynomial time. Thus, they belong to a class of population-based stochastic search algorithms, being particularly useful for problems with many local optima.

EAs include: Evolutionary Programming (EP) [37–39], Evolutionary Strategies (ES) [40, 41] and GAs [42]. As an example, Fig. 2.1 shows a general evolutionary algorithm; its main steps can be summarize as follows: a population of individuals is randomly initialized as the first step, later, it is measured the fitness of each individual to determine their degree of adaptation, thereafter, a certain number of individuals are selected to be recombined (crossover) and mutated. Fitness evaluation, selection, crossover and mutation create the main loop of the algorithm, where it is repeated until some stopping criteria is achieved, e.g. after a fixed number of generations of evolution, where each generation is an iteration of the algorithm. Finally, after the evolutionary process has finished, the best individual is obtained being the best solution found after all the evolutionary process.

Other related concepts concerning EAs will be briefly introduced in the remain of this chapter, though without a detailed explanation of all the different kinds of EAs and their operators. They will be explained in the context of the evolution of neural networks, since that will be their role in this thesis. For example, the crossover operator will not be explained in this study as this operator is not used here to evolve networks, however, the following section will explain the permutation problem, an issue that could appear with crossover operator at evolving ANNs.

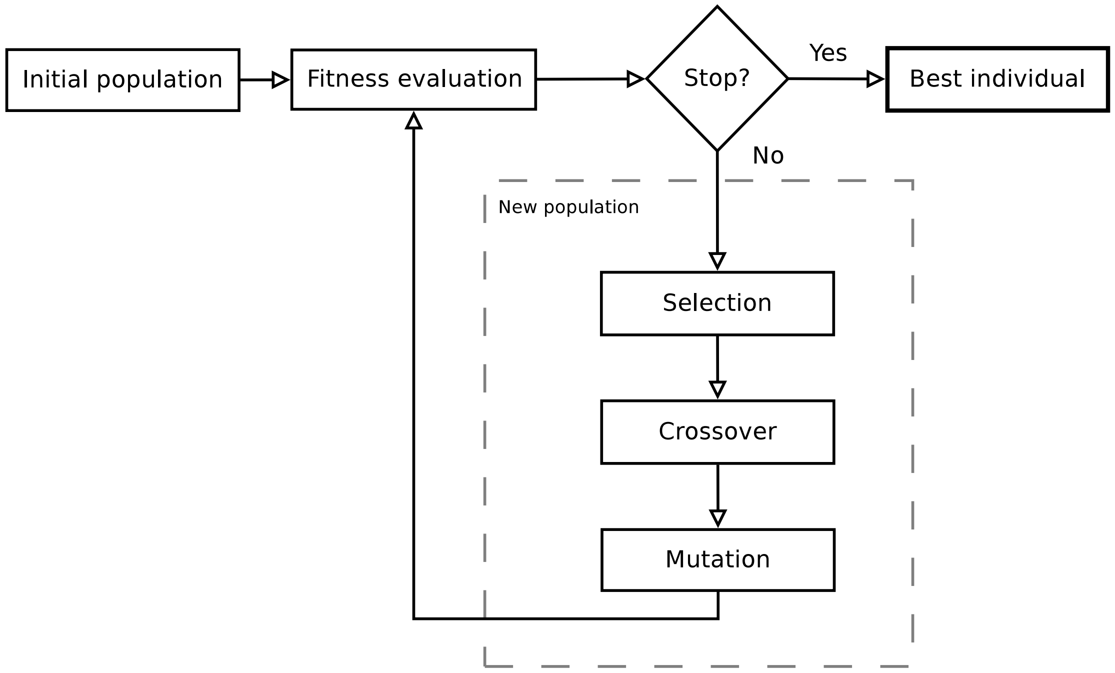


Figure 2.1: General evolutionary algorithm

2.1.1 Direct or indirect encoding scheme in ANNs

If we are going to use an EA, we need a way to encode the given problem (phenotype) into the EA (genotype), and to do this we have two general methods: direct and indirect encoding. The *direct encoding* is quite straightforward to implement as each connection of the network can be represented in a binary matrix C. In this way, each connection ($c_{i,j}$) has a direct one-to-one mapping to the network architecture, thus, the operators (crossover and mutation) can be applied directly over the phenotype, i.e. there is not need to construct a genotype because it is the same as the phenotype. On the other hand, *indirect encoding* used more elaborated representations where different characteristics of the network's architecture can be encoded into the genotype. Two examples of algorithms using *direct* and *indirect encoding* are the EPNet [35] and NEAT [43] algorithms receptively.

As stated in [10], direct encoding is useful to find small networks' architectures but there may arise difficulties with the permutation problem [33], mainly caused by crossover operators and also known as the "competing convention problem" [44]. In this case, this issue can produce functionally similar networks when any permutation of nodes takes place, moreover, it could produce invalid representations destroying the behavior learned by the parents, making crossover operators not the best option for offspring creation [10, 45]. On the other hand, indirect encoding lets us have a small representation of the genotype, but it is not so good for giving us small networks architectures nor good generalization [10]. Nevertheless, the NEAT algorithm (Section 2.2.2) can produce smaller networks architectures by complexification, i.e. starting from minimal architectures and adding components as required. Considering both schemes, this study will employ a *direct encoding* without a crossover operator to avoid the permutation problem.

2.1.2 Lamarckian and Baldwinian methods

If one considers the evolution of weights, or from another angle the learning acquired by the network, one could use two main approaches to inheritance: the Lamarckian and the Baldwinian methods. In the first one, the parent can pass on characteristics acquired during its lifetime to its offspring, meanwhile in the Baldwinian evolution this inheritance is not possible, however, it is assumed that offspring will have an increased capacity of learning new skills. Therefore, in the Lamarckian inheritance the network is trained (it could be partially trained too) and the new weights are introduced in the chromosome, allowing them to pass to the next generations as done in [35, 46, 47]. In the Baldwinian method [48], the network is trained to obtain the required fitness, but the new weights are not put back into the chromosome as was done in the Lamarckian inheritance. That means that in the Baldwinian method, we need to fully train the networks every generation, nevertheless, it is more closely related to natural evolution.

A potential benefit of Lamarckian inheritance could be seen as faster evolution of networks when they are partially trained through evolution. Even this method may be trapped in local optima (given the low convergence rate), the usage of other techniques like BP, Simulated Annealing (SA) and appropriate genetic operators may avoid this issue (as done in the EPNet algorithm, see Section 2.2.1). Contrary, Baldwin effect consume more computational resources every generation, allowing a faster convergence of the networks.

Thus, Larmarckian inheritance allows information learned during a lifetime to be encoded in the DNA (chromosome) and passed on to offspring. That is now known to not be possible in biological systems [49], though some inheritance is possible via the Baldwin Effect [48, 50], but still the amount of information that can be encoded in the DNA in that way is limited. Artificial systems do not have such biological constraints, and can benefit from Lamarckian evolution, so that will be employed in the study of this thesis.

2.2 Evolution of Artificial Neural Networks

Artificial Neural networks are mathematical models inspired by the structural and functional organization of biological neural networks. They are characterized by having input, hidden and output units with interconnection between them, where each connection has an associated weight which is updated during the training phase to allow the network to learn a given task.

The Multi-layer Perceptron (MLP) representation has been the standard to organize nodes in a layered approach, where only connection between adjacent layers are allowed. Hoverer, exist another representation (see Section 2.2.1) that allows a different organization of nodes. Independently of the representation, the networks can be constructed in a feedforward approach, where no loops are allowed in the nodes, therefore, the information follows just one path, or in a feedback approach where loops are allowed.

The training of a network could be a *supervised* training where labeled samples (patterns) are presented to it, i.e. inputs and outputs are indicated, whereas in the *unsupervised* learning the output patterns are not required. We can also have an stochastic learning if the patterns of the training set are randomly reorganized every epoch, producing to have more probabilities of escaping from local minima with a gradient descent algorithm like the BP algorithm, which is an advantage over others methods like Conjugate Gradient or quasi-Newton Algorithms [51, p. 255]. Also, during training there are different ways to set-up the number of training epochs, i.e. the number of times the entire training set is presented to the network (during training phase). One option to set-up automatically the correct number of epochs and avoid over-fitting (when the network is not able to generalize the information learned) is applying a method called Early Stopping [52].

Since their origin, they have been used to solve control [39, 43, 53–55], classification [12, 35, 47, 56] and prediction [57–60] tasks, showing a performance and adaptability superior to those of conventional mathematical models, as it is actually known that one hidden layer with sufficient nodes can approximate any classification decision boundary [51, p. 130].

Even though neural networks have proved to be a robust method for solving different kinds of problem, they involve several different parameters that need to be chosen appropriately to obtain a functional network. Some of these parameters are the number of input, hidden and output nodes, the connectivity patterns between them, the weights associated with each connection, learning rate and so forth.

Early studies used to select many of those parameters by trial and error [51, p. 269], e.g. in [61] it is indicated that one should usually allow a slower learning rate in the final layers and a large learning rate in the first ones, as the gradient tends to be steeper in the final layers. Another difficulty is that some of these parameters may change over time, and thus more elaborate methods are needed to adjust them.

Evolutionary Artificial Neural Networks (EANNs), sometimes called neuroevolution, have been remarkably useful [10, 12, 47, 62, 63]. For example, the learning rate may be evolved to optimal values during evolution [12, 34, 35, 64], where [64] adopts a learning rate per node while [9, 12] uses different learning rates for different sections of the network. Another example can be evolving the weights, where they can be encoded into the chromosome to find initial weights (a promising area); thereafter the BP algorithm can be used to gave a fine-tuned value of the weights as is shown in [47, 63], providing faster training.

Furthermore, EAs can improve the robustness of networks because they are less likely to be trapped on local minima than traditional gradient-based search algorithms [10].

2.2.1 EPNet algorithm

The EPNet algorithm, introduced by Yao and Liu [35, 65], is a *steady-state* algorithm that uses Lamarckian inheritance to evolve populations of neural networks for particular tasks. The EPNet algorithm is based on Fogel's Evolutionary Programming (EP) approach [37–39] using a population-based stochastic search approach, where only mutation operators carry out the evolution of ANNs, i.e. crossover operators are not used during evolution because the permutation problem might arise (Section 2.1.1).

EP is a general mutation-based evolutionary method that can be used to search the space of neural networks. For example, as a general optimization algorithm for weight update, in EP the individuals are represented by two n-dimensional vectors, where n is the number of weights in the network. One vector contains the weights and the other the standard deviation of the first one. Thus, a parent network could be assembled using the weight vector and the offspring could be generated by applying Gaussian noise to each element of the weight vector with standard deviation derived from the second vector. Note that this explains only how to evolve the weights, and it is different to evolve complete network architectures.

Continue with the EPNet algorithm, it uses Generalized Multi-layer Perceptron (GMLP) [66, pp. 272-273] to represent the network, where any connection between nodes is allowed in a feedforward approach. This is different from the MLP representation, where the nodes are organized by layers and only connections between consecutive layers are allowed. Consequently, GMLPs may allow more general, and potentially more powerful, architectures than MLPs. The EPNet algorithm has been tested on prediction [65, 67] and classification tasks [35, 68, 69] to evolve single neural networks [35, 65] and neural network ensembles [67, 70]. The EPNet algorithm emphasizes the evolution of ANNs behaviours by EP, like node-splitting, which maintains the behavioural (i.e. functional) link between the parent and its offspring. For example, algorithms that insert a hidden node into a layer which is already fully connected, using random weight initialization for such a node, tend to destroy the behaviour already learned by the parent [35]. To avoid this, the EPNet algorithm splits a selected node, dividing the original weights into the new nodes. In this way partial training is enough to alleviate any disruption caused by the introduction of a new node.

The flow chart which represents the main procedure of the EPNet algorithm with its mutations is presented in Fig. 2.2. The general algorithm (Fig. 2.2a) may be related to any evolutionary algorithm of ANNs. First, a population of individuals (networks) is initialized and then partially trained with the Modified Back Propagation (MBP) algorithm. Second, the evolutionary cycle is performed selecting an individual with the rank base selection procedure and mutating it. This cycle is repeated until some stopping criterion is reached, e.g. a fixed number of generations or reaching a certain level of error, among other conditions. Finally, further training is applied to all individuals before the generalization performance is tested. To simplify the flow charts presented, we use *hidden node deletion* instead of *hidden node deletion mutation*. The same convention applies for the other mutations: addition/deletion of nodes, connections and hybrid training.

The EPNet algorithm does not have a crossover operator, nor a genotype to represent the individuals. Instead it carries out the evolutionary process by performing only five different mutation operations (Fig. 2.2b) directly on the phenotype: (1) hybrid training; (2) node deletion; (3) connection deletion; (4) connection addition; and (5) node addition. The algorithm may perform only one such mutation or all of them on the selected individual in each generation before the new generation starts, i.e. there is a minimal generation gap as the new generation starts after one mutated individual replaces another individual in the population. As the EPNet algorithm was developed to have fixed inputs during evolution, the previous mutations in the architecture take place only over the hidden nodes and over all connections in the network.

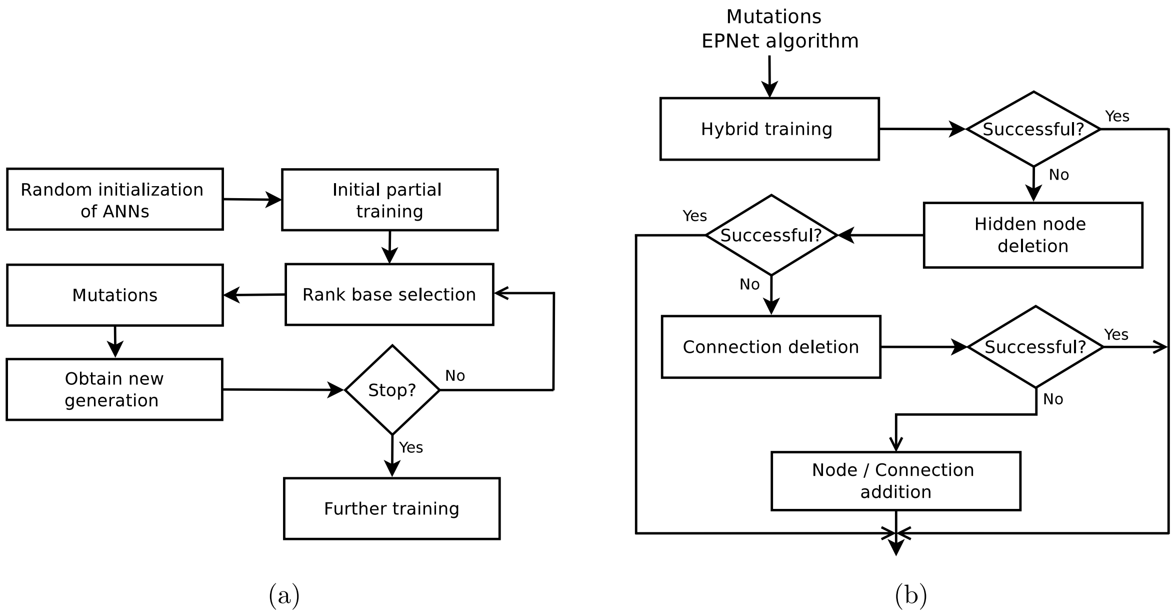


Figure 2.2: EPNet algorithm. General procedure (Fig. 2.2a) and EPNet mutations (Fig. 2.2b)

As can be seen in Fig. 2.2b, the mutations are applied in a sequential way giving greater preference to the first mutations; e.g. when performing a node deletion mutation on a parent to create a child, if the child is better than the last individual in the population, the rest of the deletion and addition mutations will not be performed on the parent network. The last step in the algorithm performs the additions, where the selected parent creates an individual (child) from each mutation. After partial training is applied to the new offspring, all of them compete in a tournament to survive and replace the worst individual in the population. As the additions are performed after the deletion mutation, they have the smallest possible probability of being applied in the population, which regulates the growth (nodes and connections) of each individual.

It is a characteristic of the EPNet algorithm always to give preference to smaller and compact architectures. For example, in [43] it is pointed out that the size of the network may be incorporated into the fitness, to create a penalized fitness function aimed at looking for compact representations. However, it could be difficult to know how large the penalty should be, and considering that different tasks may require significantly different network sizes, the penalized fitness function to select smaller network may harm the evolution. In this area, the EPNet algorithm ensures that all unnecessary neural components are deleted first before they begin to be added. This is similar to the NEAT algorithm commented upon in Section 2.2.2 with the difference that the NEAT algorithm starts the evolution with minimal configurations.

As it could be seen, the EPNet algorithm was used during this work given all the previous advantages it provides to evolve networks.

2.2.1.1 Steps in the Algorithm

The main algorithm of EPNet can be summarized as follows:

1. Generate an initial population of m networks at random.

2. Partially train each network in the population using a MBP algorithm, and mark it as a *successful* if the error is decreased by a significant amount (Section 4.3).

3. Rank the networks in the population according to their fitness, i.e. performance on the given task.

4. If the best network found is acceptable, or the maximum number of generations has been reached, stop the evolutionary process and go to Step 11.

5. Use rank-based selection to choose one parent network, and if it is marked as *successful* go to Step 6, otherwise go to Step 7.

6. Partially train the network with MBP to obtain an offspring and mark it in the same way as in Step 2. Replace the parent network with the offspring in the current population and go to Step 3.

7. Train the parent network with SA algorithm to obtain an offspring. If the error is reduced by a significant amount, mark the offspring as *successful*, replace the parent and go to Step 3. Otherwise, discard this offspring and go to Step 8.

8. Delete a random number of hidden nodes from the parent and train the offspring with MBP. If the offspring is better than the worst network in the current population, replace the worst by the offspring and go to Step 3. Otherwise, discard this offspring and go to Step 9.

9. Calculate the approximate importance of each connection in the parent network using the non-convergent method [71]. Then delete a random number of the least important connections and train the network with MBP. If the offspring is better than the worst network in the current population, replace the worst by the offspring and go to Step 3. Otherwise, discard this offspring and go to Step 10.

10. Obtain offspring 1 and offspring 2 from the parent network by adding a random number of connections and a random number of nodes respectively. Train both networks with MBP and compare them. Replace the worst network in the current population by the winner and go to Step 3.

11. After the evolutionary process, train the best network further with MBP until it converges.

Note that in the last step the SA algorithm is not used. Basically, the SA algorithm has been used for two reasons concerning the MBP: a) to avoid local minima and b) to further reduce the error in cases where the MBP cannot improve the learning.

2.2.1.2 Hybrid training

The training in the EPNet algorithm is only partial, i.e. the networks are not trained until they converge. This is motivated by computational efficiency, which lets the evolution advance faster, with the individuals improving their fitness through generations; also, it is used to bridge the behavioural gap between a parent and its offspring [10]. This is mainly supported by the fact that Lamarckian evolution is adopted, so parents can pass the knowledge learned to their offspring. Note that if Baldwinian evolution were used, full training might be needed every time a network was mutated. The hybrid training is the only operation in the algorithm that modifies the networks' weights and it is composed of training with the MBP algorithm and SA separately. It is called Modified Back Propagation because the learning rate is evolved during training under the assumption that each network must have a different learning rate, and in order to accelerate convergence and avoid local minima, known problems of the Back Propagation algorithm. Also the SA algorithm helps to avoid local optima at the time when it looks for better solutions. During training, the error percentage (equation 3.5, in Section 3.4.1) is monitored every k epochs; and if the error is reduced then the learning rate is increased by a pre-defined amount. Otherwise, the learning rate is reduced by the same amount.

2.2.1.3 Architectural mutations

When the hybrid training is not able to reduce the error further, it will be time to modify the architecture of the network to look for better solutions. In this field, hidden node deletion/addition mutations are in charge of the major mutation attempted as they can increase or decrease the number of nodes and connections in the network. On the other hand, connection mutations are concerned only with the connectivity of the network, adding/removing connections between nodes.

Hidden node deletion and addition. Hidden node deletion mutation takes at random certain hidden nodes and deletes them, whereas hidden node addition splits a selected node by a process called *cell division* introduced by Odri *et al.* [72], i.e. selection is made at random to chose node i and a copy of it is performed with all inbound and outbound connections; then the original and new weights are split in the following way:

$$\begin{aligned} w^1_{i,j} &= w^2_{i,j} = w_{i,j}, && i \geq j \\ w^1_{k,i} &= (1+\alpha) w_{k,i}, && i < k \\ w^2_{k,i} &= -\alpha w_{k,i}, && i < k \end{aligned}$$

where w is the weight vector of the selected node i (parent weight), j and k represent the inbound and outbound connections respectively of node i, w^1 and w^2 are the weight vectors of the new nodes (child connections) and α is a mutation parameter fixed or taken at random. In this study α was set with a random value, in range (0,1), every time a node is divided into two. For example, splitting a node i means that the new inbound connections of the new nodes are the same as the inbound connections of node i. Thereafter, each outbound connection of node i is divided by the random value α, as previously shown, to create the new outbound connections for the new nodes: w^1 and w^2.

Connection deletion and addition. Connection deletion mutation selects probabilistically certain connections to be deleted, using their importance as stated by the non-convergent method presented in [71]. The advantage of this method is that it is possible to test a connection without a full training of the network. In contrast, connection addition mutation selects those connections with zero weights at random, then the new connections added are initialized with a small random weight.

2.2.1.4 Feature selection

The fundamental role of input feature selection is the reduction of the dimensionality of the input space through the selection of the most relevant information, discarding unnecessary data values (those which are redundant or irrelevant) that may cause interference in the solution of the task. Thus, if the information is redundant, it will add nothing to solving the task, and if it is irrelevant, it will not improve the results [56]. Moreover, if input feature selection is not used, it is possible that unnecessary inputs will result in bigger ANNs requiring more time in the training process, or introduce unnecessary noise into the network which will typically result in poorer ANN output performance.

Existing feature selection techniques fall into two general categories [56]: *Generational* procedures and *Evaluation function* procedures. In both methods, several different variations are used to determine the most appropriate inputs to the model.

In this area, the EPNet algorithm was developed simply to have a fixed number of inputs, i.e. the input feature selection problem was known *a priori* and only the network's architecture was optimized during evolution. That was a disadvantage as it did not exploit all the facilities provided by evolution. Moreover, if we wish to use the EPNet algorithm to evolve MNNs and reuse modules, we may have a flexible algorithm that allows for feature selection, as the inputs for one module may not necessarily be the inputs for another module. Even though, feature selection during evolution is the preferred method for this study, it should be pointed out that the evolution of inputs may be noisy [10] as their modification can drastically change the behaviour of the network.

The general procedures for evolving the input features with crossover operators are now well established [10, 73–75], and the only work found in the literature that evolves the input feature and network architecture with single mutation operators is [76], using five structural mutations for a visual task system (Mosaic World task). However, it may not be a straightforward procedure to adapt the EPNet to this new task because the input features for TS forecasting (Section 3.1) rely on the correct selection of inputs and delays between them, different from classification tasks where input feature selection may be seen as the activation or deactivation of a selected set of inputs. Thus, at least two different approaches are needed to tackle each one of them.

2.2.2 NEAT algorithm

The GA [42], as previously commented in Section 2.1, is an EA that uses recombination and mutation operators, and it could be used for the evolution of ANNs [47, 77], even though, the GA is not specialized to evolve networks. A more specialized algorithm has just been discussed, the EPNet algorithm, where only mutation operators are required in finding optimal network parameters. On the other hand, Stanley and Miikkulainen introduced the NEAT algorithm [43], a constructive algorithm that uses crossover and mutation operators. With that, complexification of the networks can be achieved at the same time the network structure is optimized, similar to the SAGA algorithm [78, 79] with variable length genotypes. To avoid the permutation problem, NEAT algorithm line up corresponding genes (using indirect encoding) when two genomes cross over to create a child. Nevertheless, the original implementation of the NEAT algorithm does not consider input feature selection, hence it extension (FS-NEAT) has been developed, showing favorable results [74]. Moreover, it has been extended to study the evolution of MNNs (modular NEAT) as it will be shown in Section 2.5.1. Note that all the different versions of the NEAT algorithm have mainly been focused on solving control problems. For example, another extension called Hypercube-based Neuro-Evolution of Augmenting Topologies (HyperNEAT) [80] was developed particularly to solve the geometrical correspondence between sensors and effectors for robots.

2.2.3 Other evolutionary algorithms

There are numerous algorithms in the literature to develop ANNs. Some of them improve previous algorithms and other just focus in specific aspects of the evolution. One example is the Cooperative Synapse Neuroevolution (CoSyNE) algorithm [55] aimed to evolve individual synaptic weights. There is used a cooperative co-evolutionary algorithm which means that each individual represent only a partial solution of the problem, while a complete solution is grouping several of them together. Therefore, the goal of a individual is optimize one piece of the solution and cooperate with other partial solutions that optimize other pieces.

Another example is the Symbiotic Adaptive NeuroEvolution (SANE) algorithm [53] which also uses a cooperative co-evolutionary algorithm having two different populations: one for neurons and the second for network blueprints which indicate how neurons are organized to form a complete ANN.

Even the different advantages that seems to have previous algorithms, it was decided to use the EPNet algorithm to focus in a single algorithm, and because it uses a simple representation (direct encoding not requiring to construct a phenotype from a genotype) to evolved networks with a single population of networks.

2.3 Modularity

Modularity has been much studied in recent years, and a range of definitions for modules [6, 7, 12, 17, 32] and modularity [7, 17, 19, 81] have emerged. For example, *modularity* can be defined as a property of neural networks where the connectivity patterns are organized in such a way that modules consist of disjointed subsets of hidden units [12, 17]. Some definitions consider *modules* to be non-interacting subsystems [12, 17], i.e. independent partition of nodes, while others allow them to interact [32, 82]. In general, all works agree that nodes withing a module are highly interconnected among themselves and that there are just a few connections communicating between modules, something similar to the small-world theory [3, 83].

In the next section will be presented some related concepts regarding modularity in the human brain to have a point in comparison against artificial systems, and to extend the information provided in Section 1.1 about biological systems. After that, more related concepts about modularity in artificial systems are introduced.

2.3.1 Modularity in the human brain

The human brain has been a source of inspiration to develop more sophisticated computational techniques that lead to better performance. It is now known that the number of connections between neurons is limited [2, 84] but also that a bigger brain does not necessarily allow greater intelligence because brains do not scale up very well [1] and also because the bigger the brain, the more difficult it is to maintain connections and the more energy is required to allow communication between neurons, without counting attenuation of any signal transmitted [1, 2].

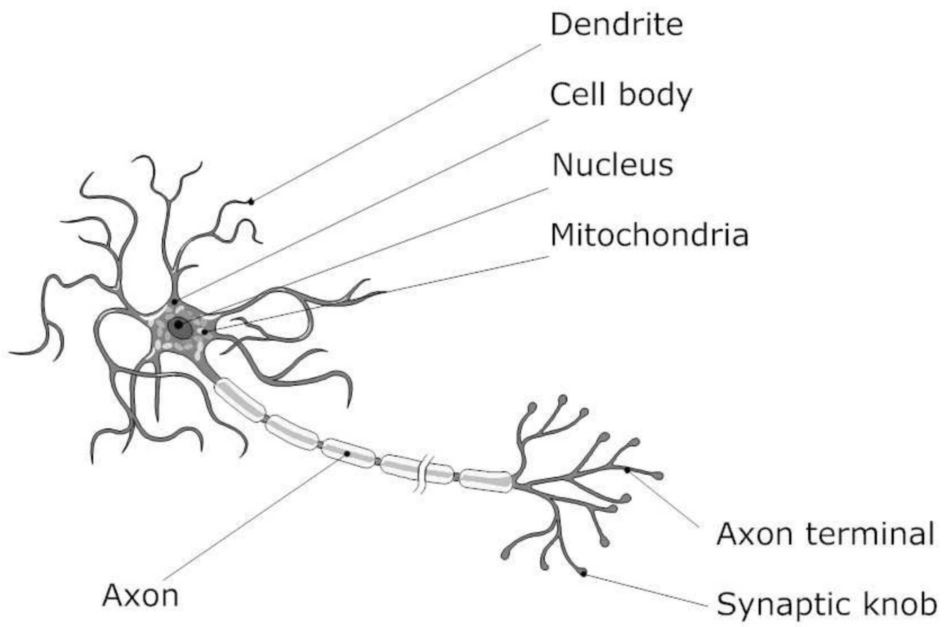


Figure 2.3: Biological neuron. LifeART Collection Images©1989-2001 by Lippingcott Williams & Wilkins, Baltimore, MD

Two neurons can be connected by different elements: *synapses* are the structures that pass a signal between two neurons, *axons* are the transmission lines and *dendrites* the receptive zones. The *synapses* can be found at the end of the *axon* communicating the information from one neuron with a *dendrite* of another neuron. Fig. 2.3 shows a general representation of these elements.

In this case [84] has found that neurons that contribute to different areas (modules) are sparsely connected, receiving less than 3% of connections from their neighbours in a square millimetre of cortex. The sparsity of shared neural components has also been pointed out by the study of [2] explaining that if the level of *dendrites* is exceeded within a certain space, this will result in insufficient space for *axons* and *synapses*. In an evolutionary approach, it can be said that the neural circuits of the human brain have been optimized to minimize conduction delays in *axons* (through shorter connections); to reduce the attenuation in *dendrites* caused by the large connections, and to maximize the density of *synapses* [2, 83]. Hence, minimizing wire length in brain organization is crucial for module formation [2], and thus, leading to its structural and functional organization which allows it to reuse neural circuits for different cognitive purposes [3, 4, 6, 7]. This reuse is mainly achieved through the highly modular structure of the brain [1, 3, 16, 83, 84].

For example, [5] is focused on the understanding of mind brain mechanisms and stating that there are hierarchical structures in the brain that lead to modular organization. Clearly the hierarchical structure of the brain can be seen as a hierarchical modularity where smaller modules could be embedded into bigger ones [3] giving different levels at which to carry out analyses. Those levels may well be the result of the different evolutionary stages that the brain has passed through over the years, e.g. the triune brain theory developed by Paul MacLean [85] indicating that three distinct brains have evolved successively: the reptilian, limbic and neocortex brain. Where the reptilian brain controls the body's vital functions like heart rate, the limbic brain allows emotions and other tasks such as memories of behaviours, and the neocortex brain is the final layer in the brain allowing the development of language, abstract though, imagination and consciousness.

2.3.1.1 Theories of module formation and module reuse

There are different neural reuse theories suggesting that low-level neural circuits are used and reused in different tasks over different cognitive and domain processes as pointed out by Anderson [4]. In terms of resources, two main theories have emerged: anatomical modularity and the optimal wiring hypothesis; whereas concerning module reuse there are four different theories: the neural exploitation hypothesis, the shared circuits model, the neuronal recycling hypothesis and the massive redeployment hypothesis.

Anatomical modularity refers to the functional modularity indicating that our cognitive system can be deconstructed into separable (or almost separable) modules [86]; but [4] suggests that those theories may not be correct because the brain does not have dedicated regions for a single high-level task and because there are some insights pointing out that different cognitive functions could use the same neural circuits together in different rearrangements [87].

Optimal wiring hypothesis refers to the optimization of connections in the brain to maintain low levels of energy consumption, resulting in a considerable number of short connections rather than longer ones.

Neural exploitation hypothesis says that the level of cognition we have is by "the adaptation of sensory-motor brain mechanisms to serve new roles in reason and language, while retaining their original function as well" [88, p.456]. There is some indications that reuse happens in three steps: 1) *understanding* requires imagination, e.g. if one wants to grasp a cup, one needs to imagine its parameters to grasp it; 2) *imagination* can be considered as a simulation of the real action, thus, 3) *simulation* is therefore neural reuse because one may reuse the same clusters of nodes involved when the real action is performed.

The shared circuits model. The shared circuits model [89, 90] is formed by five control layers of similar structure, where each layer has an adaptive feedback loop. The first layer is in charge of the perception-action; the second layer takes inputs from layer 1 and also from the external world; in the third layer the circuit-sharing starts, in which the same circuits in layers 1 and 2 activate basic control circuits and inhibit some output units; layer 4 monitors the output inhibition and layer 5 allows the system to decouple from inputs and outputs.

Note that the shared circuits model has been only proposed, and no computational algorithm has been found that explores its hypotesis.

The neuronal recycling hypothesis. The neuronal recycling hypothesis [91, 92] is focused on how the brain is shaped and reorganized when we learn different tasks. In line with previous theories, there is an indication here that the circuits already developed will be put to work in fairly similar tasks when they are reused. However, the greater the difference between two tasks the more difficult it will be to acquire the new functionality, because of neural plasticity required to adapt to a new task and overcome the existing cortical biases.

The massive redeployment hypothesis generally indicates that evolution has favoured the emergence of the functional organization of the brain, but maintains a close relationship with the recycling for neural circuits on different cognitive purposes where there has not been enough time for some neural circuits to become specialized.

As remarked by Anderson [4], each of these theories seems to be limited in aspects where others are focused, and probably a combination of all of them may lead in the future to a better theory that explains the whole scenario.

2.3.2 Modularity in artificial neural networks

We have seen different theories and information of modularity in the human brain. Henceforth, we will focus in modularity on artificial systems as well in different studies over modular neural networks. For example, Genetic Programming (GP) [27–31], commented in Section 1.2, is the technique of evolving computer programs using operations such as crossover and mutation. Modularity is naturally presented in GP because modules may be seen as branches of the program code (subtrees). Therefore, GP has the characteristic to automatically create, modify and delete modules which can be used in a hierarchical fashion [27], moreover, modules can be reused in different locations. Considering neural networks, [30] used a genetic programming for automatic design of modular neural networks. However, external rules were required to represent an ANN through a tree structure.

2.3.2.1 Cross-talk interference

Cross-talk interference is a problem when learning two task simultaneously, where the learning of one task interferes with the learning of the second one, producing weight conflicts inside the learning algorithm. The cross-talk between two modules or partition of nodes can be classified in two parts:

Temporal cross-talk: may happen when the network is trained by several epochs with patterns of one region of the input space, and later it is trained by several epochs with patterns from another region [36]. That can also be seen as training the network for different tasks at different times, similar to changing environments [19, 22–25].

Spatial cross-talk: appears when the output errors of a network present conflict errors to a hidden node during the training [36]. To avoid that, hidden nodes from a given module may not communicate with output nodes from different modules. This is something that has been investigated by several different researchers as shown in the following sections.

The human brain can deal with such interference up to some reasonable levels, e.g. as explained in [21], one can drive a car and listening a radio at the same time with little interference. Moreover, one can perform easily multiple-tasks in parallel with not too much interference, but similar tasks, like two auditory or two visual tasks, are prone of more interference.

2.3.2.2 Stability-plasticity dilemma

Derived from the Temporal cross-talk, the stability-plasticity dilemma [93, 94] is presented when the local minima of the objective function of one training set could be different to the local minima of the objective function of another training set. As a consequence, a network tends to forget the previous training patterns when it is presented a new training set.

2.3.2.3 Modularity measures

Previous studies have developed different ways to measure how close a network is to being a pure-modular architecture, i.e. having independent partition of nodes.

For example, a clustering coefficient measure has been introduced by [95] that allows the measurement of clusters where the bigger the value, the more modular the network will be. On the other hand, Bullinaria [12] measures the amount of connectivity between modules during evolution; if the amount of connection decays to minimum levels (or zero connections) modular architectures appear, as there are independent partitions of nodes. Another example is the presented in [15] were the modularity of networks is measured in terms of cross-connections between partitions. In addition to these studies, Newman and Girvan [96, 97] have developed a more structured modularity measure that allows the analysis of community networks, i.e. the relationship of different nodes at different levels, or more precisely the modularity level at different levels of the network. However, the modularity measure presented in [96, 97] requires the optimization of the best partition of nodes into the network to allow calculation of the modularity measure, i.e. maximize the node partition. The best partition can be found with the SA algorithm but also any other search/optimization algorithm could be used, like a standard GA algorithm.

The next modularity measure is the one introduced by Hüsken et al. [17] using the connectivity patterns to determinate an architectural modularity ($M^{(arch.)}$) or taking into account the network's weights to obtain a weighted modularity ($M^{(weight)}$). It has been found [17] to be less computationally expensive than [97] because in $M^{(arch.)}$ or $M^{(weight)}$ it is assumed that output partition is known *a priori*, so no extra optimization process is required.

Also, [97] may not be useful for this thesis as every time it is run the metric, it produces different results given the heuristic to maximize the node partition, for what is suggested to compared several runs. On the other hand, the modularity of [17] has a bigger applicability in different problems with different sub-tasks and more precise than [12, 15] previously described. Another advantage provided by [17] is that it measures the contribution of each node into each module, which allows the further extension of the EPNet algorithm.

For these reasons, and for the purpose of this study, the modularity measure introduced by Hüsken et al. [17] will be enough to test the different research questions previously presented in Section 1.4. This modularity measure is detailed, presented and extended in Chapter 5 with the introduction of the modular algorithm designed to develop modular architectures. Independently of the modularity measure used, it is clear that the general increase in modularity during evolution supports the intuition that modular networks are indeed advantageous for the task at hand. If it is a modular problem, higher levels of modularity can be expected. Some modular tasks used throughout this thesis are presented in Section 3.3.

After a modularity measure has been used and applied to identify modules inside a network, regardless of whether it is pure-modular or pure-homogeneous, we can assign different roles to the nodes of the network or measure different parameters to create a more detailed classification of the roles per node, as shown in the following sections.

2.3.2.4 Within-module degree and participation coefficient

In [98] is presented two measures to classify the nodes: a) within-module degree to determinate how well connected the node is against other nodes in the same partition, and b) participation coefficient to determinate the degree of participation in different modules. Nodes with similar roles are expected to be in the same module, i.e. they may have similar relative within-module connectivity:

$$z_i = \frac{k_i - \bar{k}_s}{\sigma_{ks}} \tag{2.1}$$

where k_i is the number of inbound and outbound connections from node i to/from nodes in the same module s, $\bar{k}_s$ is the average of k over all nodes in s, and σ_{ks} is the standard deviation of k in s. In this way, the *z-score* allows us to compare the degree of intra-modular connectivity of a node against all other nodes in the module. The participation coefficient P_i of node i is defined by:

$$P_i = 1 - \sum_{s=1}^{N_M} \left(\frac{k_{is}}{K_i} \right)^2 \tag{2.2}$$

where k_{is} is the number of links of node i to the nodes in module s, and K_i is the total degree of node i. P_i is close to 1 if all its connections are uniformly distributed over all modules and 0 if only is connected to nodes of the same module.

2.3.2.5 The role of nodes

In [17] nodes are differentiated by the connections within nodes of the same module, being called *pure nodes*, and their connection are called *pure connections*. Nodes that contribute to different partitions are called *mixed* nodes as they contribute in some degree to different modules. But no definition is used to indicate connections that communicate between nodes from different modules.

Using the within-module degree and participation coefficient measure previously described, [98] creates the next classification of nodes: the *provincial* (or peripheral) node in cases where the node has only a few (or zero) inter-modular connections; *connector* node in cases where it contributes to several modules; *hub* node when the *z-score* of a node is higher than the other nodes in the module, and *non-hub* in the other case. Other node classifications can be found in [98] depending on the cut-off values measured, but in general they are the combination of all previous ones, e.g. *connector-non-hub* node is a hub node that is connected to other modules.

All these definitions give different names to different parts of the networks, but none of them cover all those found up to the present. For the purpose of this study, the term *shared* will be used to identify neural components that contribute to different modules: *shared* nodes and *shared* connections (or cross-connections). For the case of components within the same module, there would be no problem in using the terms *pure* nodes and *pure* or *intra-modular* connections. Hence, only these neural components will be required in this study with the modularity measured presented in [17]. The within-module degree and participation coefficient measures will not be used, even they provided more information. It is hoped that in a further study, these measure could be used to extend this work.

2.4 Modular neural networks and their evolution

Modular systems are usually seen as a collection of independent components that work together for specific purposes, e.g. with each component specialized to perform a particular sub-task that may be used multiple times. This occurs naturally in neural networks where it is possible to have disjoint partitions of the neurons (i.e. modules). If there is no communication between modules (i.e. independent partitions), one has a pure MNNs, whilst if the nodes are highly interconnected (i.e. with no independent partitions), one has a homogeneous or non-modular network. The degree of modularity can be measure with different methods as remarked in Section 2.3.2.3.

An early review on MNNs may be found in [99] presenting that modular architectures require three steps: 1) decompose a task into subtasks; 2) organization of the modular architectures and 3) perform the module communication. Those steps can be carried out in two different ways: a) through the partition of the data set into subsets where each subset represent a sub-task having a predefined architecture (module organization and communication) as shown in Section 2.4.2 or, b) leaving all these steps to be done automatically by an EA as presented in section 2.4.1. In the first case, one forces to have independent partition of nodes, while in the second one, evolution is in charge to find the optimal structure, which may or may not be modular.

Concerning the evolution of MNNs, in Section 2.2 was shown the evolution of ANNs with the EPNet algorithm; and an overview of other methods like the NEAT algorithm was provided. The evolution of modular architectures may not be as different as the evolution of normal networks, if modules are identified with another technique (like Section 2.4.2), each module could be evolved with a standard EANN algorithm. On the contrary, if one is using a compound task like the what-where data set (following section), then an standard EANN algorithm also should be capable of finding independent partitions of nodes if modular architectures are beneficial for the task.

2.4.1 Automatic modular representation through evolution

In the transition to understanding and applying the information we know about the brain (Section 2.3.1) to artificial ones, we may find the early study of Rueckl et al. [16] which investigates modularity in the human brain with simplified retinal images called "what" and "where" (what-where data set). It was pointed out that better internal representation and easier learning were achieved if modular architectures were used than a fully homogeneous network.

The work carried out by Rueckl et al. has been widely examined by later studies [17, 21, 36] and extended in other cases [11, 12], where [36] uses non-evolutionary algorithms and [11, 12, 17, 21] use some kind of EAs. For example, in [12] was showed [16] to be wrong because it used a poor learning algorithm. It depends on the task whether modularity emerges, unless the degree of connectivity is constrained. Also, it was shown that some networks (fully connected) are less likely to have cross-task interference, and therefore do not require modules, but on the other hand this problem is more severe in some ANNs. The missing aspects by [12], indicated in the same study, leave some questions open, like: where modules are not an advantage and how the modules should interact. Moreover, this makes us wonder whether the brain could work better if it were fully connected, because we have seen that its modularity is mainly affected by the cost of having longer connections rather than shorter, and because there is not enough space to hold full connectivity, besides the interference resulting from different tasks.

In this thesis it has been adopted the emergence of modules during evolution because is more general representation than previous studies presented here, and because it is a more general approach than when we force to have modules as in the next section. For example, the module evolution in [12] is carried out using a module for each tasks and an extra module for shared neurons. The nodes are only evolved because it is assumed full connectivity between layers. Thus, if the shared neurons are zero during or at the end of the evolution, the network generated is a pure-modular network, on the contrary it can be said that it is an homogeneous network. Another example is the presented in [100] with two population of individuals, the first one containing modules while the second one synthesizes complete systems by drawing elements from the pool of modules (see Section 2.4.3). If one evolves nodes and connections for modular architectures as done in this thesis (like any other network), one will have a more general representation to investigate the module formation in any kind of task, rather than using just two modules, or a population of independent modules, as previous works.

Until now, this section has presented related studies that learn two tasks simultaneously, where input patterns are connected to a single homogeneous networks (as initial state), expecting to find independent partition of nodes during evolution that minimize this interference [12, 18, 101]. However, the cross-task interference will not appear if a modular representation like the presented in Section 2.4.2 is used, e.g. ensemble of networks.

2.4.2 Modular representations by data partitioning

In the literature it is common to call MNN any architecture that is not monolithic (or homogeneous) as long as it can be localized different modules that solve specific tasks such as Ensembles and Mixture of experts. These methods rely on problem decomposition, or better said, on the data set decomposition into sub-tasks as explained in the following sections.

2.4.2.1 Ensemble of networks

Ensemble of networks refers to training different non-modular networks to solve a single problem under the assumption that there is more valuable information in a whole evolutionary population than in a single best individual [68, 102]. For example, each individual in the population could form the ensemble [69] or the best individual of each independent run [67]. The final decision could be made by the majority vote method, where each network nominates a class, or taking the average of all outputs for each network in the ensemble and so forth. Usually each network is trained on a different partition of the data set as done in [68]. The EPNet algorithm has been used to evolve ensembles [67, 68, 103] where [103] treat them as modules, finding and integrating them in the same evolutionary stage, different to first develop them and later consider their integration into the ensemble as [104]. The issues raised with the method followed by [104] is that it will be more difficult to determinate what module has a bigger contribution because one does not have a feedback to determine that. Bagging and Boosting are ensemble learning algorithms that achieve improved performance by training different learners on different distributions of the data when combining their outputs [105–109].

2.4.2.2 Mixture of Experts

In mixture of experts [36] there are different networks (experts) receiving the same input pattern, and each one of them are considered as a solutions to specific subproblems. The experts compete with each other to learn training patterns thorough a gating network, which also receives the same input patterns. The weights of experts and gating network are adjusted differently during learning in such a way the gating network decide which expert is the winner to solve a specific pattern. In this sense, the input space is divided and solved by each expert [36, 110–112], similar to the division of the input space used in bagging and boosting algorithms.

A further version of this algorithm may be found in the Boosting mixture of experts [110]. There does not use evolution to find the experts nor the data partition, instead each expert is trained with a portion (overlapped or not) of the data training at the beginning. Thereafter, is taken all the patterns in which the expert was less confident, then a new expert is added with such patterns. In this way, experts focus on a part of the input space and the boosting is in charge to split the training data and assign difficult patterns to new experts.

2.4.2.3 Others

There have been further algorithms implementing MNNs through the partition of the input space into different subsets that are not relevant to the study of this thesis. A review of different methods that are not explained here (like Decouple modules and ART-BP models) is presented in [111, 113–115]. For example, in [113] is introduced a new modular representation called Cooperative Modular Neural Network (CMNN) and explaining that this kind of modular representation is sensitive of the task decomposition method and multi-module decision-making techniques. Another example is the Learning Classifier System (LCS) introduced by Holland in 1976 [116]. LCS is a machine learning technique which uses reinforcement learning and GAs to maintain different rules (stimulus-response) to form chains of reasoning, in this case, the GA is in charge of evolving those rules. From the point of view of modularity with ANNs, [117] has built an anticipation system based in LCSs using ANNs, where each rule is represented by a single network. Thus, the problem could be solved by all networks found (rules), where each network is a particular solution for each sub-problem, and that fits in the definition of MNNs.

2.4.3 CoMMoN

In the remaining part of this section will be presented other modular algorithms previously developed. The CoMMoN algorithm [13, 100, 118] was developed for automatic problem decomposition using a co-evolutionary algorithm to evolve modules and complete *systems* with two different populations. The population for the *systems* is in charge of gathering all outputs modules into a final module that produces the correct output. A combined problem is created between the *Lorenz* and *Mackey-Glass* TS using three inputs of each one. Additionally, the number of modules is fixed introducing prior knowledge of the problem. The modularity is defined only in terms of inputs, i.e. if the inputs are correctly partitioned during evolution into the module (task) they belong, the whole architecture is considered pure modular, otherwise it is classified in 3 more categories: fully-connected, impure-modular or imbalanced modular, depending of the connectivity patterns. Clearly, [13] exploits the advantages of co-evolutionary algorithms, however, modules are always considered to be independent among them, thus no module communication is allowed and no modularity measure is used during evolution because the structure of the system remain pure-modular (without considering inputs) during the whole process.

2.4.4 Incremental growing in modular networks

MNNs have also been used to evolve modules, evaluating incremental architectures as done in [119] where is tested incremental growing of ANNs as explained next. The movement of limbs and the vision system of a legged robot is the task chosen to be solved. Using a base network, modules are added when the initial configuration cannot solve the problem satisfactory. In this case, only the module is trained and added to the current structure with an EA. Every time a module is added, the current structure remains fixed.

Modules are added at the top of the previous set-up until the problem is solved. A first implementation gave poor results because the modules added had a lack of neurons, remembering that only the module addition and training was evolved. Thereafter, was added a procedure to add neural components inside the module until its performance was improved. Unfortunately, this procedure was not explained, but clearly an EA was not used for such purpose. In the first stage of the visual system, the robot moves forward if there is light and retreats when there is shadow; later more features were added. Considering 4 legs, with 2 degrees of freedom in the movement and the visual system, the final network evolved was formed by 200 neurons distributed into 5 modules.

In addition to all the restrictions (e.g. no evolved modules) imposed by [119], it was not explained how the information flows inside the network, before and after to add a module to the current set-up. Also, it is not said how modules interact and how they get activated, nor if the robot was able to walk without problems or the level of vision achieved by it. Where the module activation refers whether all modules work at the same time, e.g. is the robot always moving its legs?, or it can activate or deactivate some of them in a given moment, or just null inputs are introduced into the module to produce no output in a given limb. Other disadvantages of the method may be found in detail in this study [119], provided by the same authors.

Other modular algorithms have been developed, e.g. [120] implemented a MNN for grasping tasks or [121] created a cooperative constructive neural network for classification tasks. Though full discussion of these less relevant works evolving MNNs cannot be presented for lack of space, they are listed here [58, 120, 122–124].

2.5 Module reuse

It is interesting to note that module formation has been addressed from different angles as seen above, however, module reuse has been quite challenging to tackle. As pointed out in [125], it is expected that similar neurons with their own connections cannot deal better for two problems than the use of separate partitions, but clearly that can be achieved in the brain. Moreover, it may be assumed that this is dependent on the task at hand.

Concerning the genes in the chromosome, they can be duplicated to provide bigger representation as [45, 126, 127], for example, [45] use a cellular encoding which duplicate the genes with the cell division method, as done in the EPNet algorithm. Contrary, the genes can be reused like [15, 128], where [128] uses a EA called Symbiogenic Evolutionary Adaptation Model (SEAM) that reuse genes in the solution, which is more related to a crossover operator reusing genetic material from two parents to create a new one. The work carried out by [15] is the most relevant in this section, and explained next.

2.5.1 Modular NEAT algorithm

As its name indicates, the modular NEAT algorithm [15] is an extension of the normal NEAT algorithm presented in Section 2.2.2. It was developed to decompose automatically a problem (board game) in small sub-problems during evolution to solve the problem more efficiently. It also allows neural reuse, from a simple connection to complete networks/modules.

There are two populations in the modular NEAT, one for modules and one for blueprint or network, similar to [13]. Both populations are evolved together symbiotically as in the SANE algorithm introduced in Section 2.2.3, with the difference that in the SANE algorithm the nodes are only used in one network, whereas in the modular NEAT each module can be bound to different input/output neurons having a constructive effect.

The modules found are reused in different spatial locations of the network, i.e. the algorithm only solves each sub-tasks one time. The modules are encoded in the same way as in the NEAT algorithm, and their evolution is similar to a conventional network. Thus, the original operators for NEAT are used to evolve modules and new crossover and mutation operators are introduced to evolve the blueprint population.

Even with the promising implementation characteristics shown by the modular NEAT, it was only tested on a board game. Where the game is expected to have many reusable modules to evolve specific genes and to reuse them, encouraging generalization instead of specialization.

The algorithm is compared against the standard NEAT algorithm and it was found to have superior performance in generalization and speedup to solve the task. However, the authors did not present a detailed analysis of how the modules interact, or which neural substructures are reused more times, nor if it can be applied to other kinds of problems, like control problems on which the NEAT algorithm has been tested.

2.6 Issues to address

It is clear that there are a number of limitations to the previous work in this area, and this thesis aims to address the following issues.

In the literature it is possible to find several approaches and combinations of them related to input features, e.g. leaving the ANNs' architectures fixed and finding appropriate inputs [129, 130], or leaving the inputs fixed and then evolving the architectures [10, 65, 77] to adapt the networks to the inputs. In this study a more general approach is adopted, considering that the EPNet algorithm has not been tested in this field, in which both aspects are evolved (inputs and architectures). That also applies to the EPNet extension aimed at evolving more modular architectures. Moreover, in Section 4.4 two different ways are tested of evolving features during evolution with just mutation operators in the EPNet algorithm.

Concerning MNNs, in the CoMMoN algorithm (Section 2.4.3) interaction of modules are not allowed during evolution. Moreover, the number of inputs is fixed in the representation and it has not been tested whether it could be extended to the decomposition of tasks if more inputs are involved. As module interaction at neuron level is not allowed, it may lose any possibility to reuse internal neural components of a module in another one. On the other hand, [12] presents a more natural way to evolve modules, i.e. it allows the emergence of pure-modular architectures or pure-homogeneous ones. However, [12] uses a simulated evolution where the modules and nodes are fixed over MLP representation, and only the nodes are allowed to change module (full connectivity between modules and output nodes is adopted during all the evolutionary process).

Regarding module reuse, the modular NEAT algorithm has made the first attempts to reuse neural substructures in different spatial locations to solve each problem just once, encouraging generalization instead of specialization. Nevertheless, an analysis of which modules emerge for which tasks has not been provided, nor a structural and functional analysis of the algorithm and the modules evolved has been carried out. Moreover, it was not said how the resulting modules communicate with each other, nor which kinds of module were reused. Concerning generalization and specialization, there is no solid indication that the module reuse produces good generalization, mainly because there was no quantification of the module reuse.

If one considers combined tasks similar to what-where for evolving modules, one could notice that several studies have tested their algorithms with just two combined task, discarding the fact that in the brain several tasks are performed in parallel. Moreover, classification tasks are used to test module formation, and no clear evidence has been provided if MSP and SSP methods are candidates of module emergence and reuse. Clearly, many of the limitations found in past studies concerning modularity are due the lack of knowledge of how modules are deployed in the brain and how they communicate between each other to allow the reuse of neural structures.

Considering all these missing aspects, it can be seen that a more general and natural way to evolve modules is required, without the need to have independent modules or impose constrains like fixed number of modules, nodes, connectivity between layers or the usage of MLPs instead of GMLPs, which clearly may benefit with a better understanding of module emergence. Therefore, as seen in Section 2.2.1 the EPNet algorithm could overcome some of these limitations of evolving ANNs. Thereafter, it can be extended into the M-EPNet algorithm to tackle module formation, borrowing the basis of EPNet. Moreover, different ideas of module formation in the brain can also be used here to enhance the algorithm for evolving modular architectures.

Concerning module reuse, a straightforward experiment needs to be set-up to test whether it is possible and advantageous, which was a missing aspect from modular NEAT. For example, in a first stage, evolve 2 modules for 2 sub-tasks simultaneously (x-y data set); in a second stage repeat one of the evolved task (x-y-x) and continue the evolution of the previous modules. With this configuration it may be expected that similar tasks will use the same evolved module. Furthermore, the later introduced task may take advantage of the modules previously evolved, reusing all or part of the module to solve its own task, instead of evolving new nodes and connections. To quantify the number of nodes reused, a metric needs to be implemented to measure the number of nodes reused from other modules (Section 6.1) which may help to understand the behaviour of the later tasks introduced.

Chapter 3

Data sets and performance measures

This chapter provides a description of the data sets and performances measures used during this study as well how and where they have previously been used in the literature. The data sets are divided into two categories: a) data sets formed of different patterns from a TS for prediction and b) data sets contained patterns for classification tasks.

In the early stages, a single prediction or classification task will be used to test the performance of the EPNet algorithm, checking if its improved version provides acceptable results against previous algorithms in the literature. In the second stage, two or three tasks will be combined to perform the evolution of modules with the M-EPNet and in the final stage, a combination of previous stages will be used to test module evolution and module reuse: i.e. first a base modular architecture will be evolved, and then a simulation will be carried out where a new task arrives in the system, incorporating the new modules into the base modular architecture.

For that reason, in Section 3.1 a TS definition is introduced with three different methods to perform the prediction and three different chaotic TS (*Logistic*, *Lorenz* and *Mackey-Glass*). *Lorenz* and *Mackey-Glass* are generated and tested with different configuration parameters to make possible comparisons with previous algorithms in the literature, and also to generate similar TS that could be solved with a modular algorithm, i.e. testing the evolution of MNNs with similar tasks. Note that, even though only three TS have been used during the thesis, the improved EPNet algorithm (Chapter 4) has been found to give reliable results on natural TS like Sunspot series or the Laser TS from the Santa Fe competition problem, in two derived publications of this thesis [67, 131].

In the classification field, Section 3.2, three standard and well known data sets are presented: *Breast cancer*, *Optical digits* and *Thyroid* data sets. To finalize the test problems used, Section 3.3 will introduce different artificial classification tasks generated that will be used for the combined data sets, to test the module evolution and reuse. The same concept is used to explain how different prediction and classification data sets are combined to evolve modular architectures. With the limitation of computational resources to test different independent runs for each TS or data set, it is not possible to increase the number of TS for prediction and data sets for classification. Nevertheless, they may be considered as a representative sample of standard prediction and classification tasks and their applicability to other test problems should be clear.

As different studies use different performance measures, Section 3.4 shows several equations used in the error calculation.

3.1 Time series prediction problems

As stated in [132, p. 11], the general theory of linear predictors can be tracked back to Kolmogorov in 1941 and Wiener in 1949. Where linear predictors assume that the system is linear and stationary. On the other hand, the beginning of the modern TS prediction was in 1927 when Yule introduced autoregressive approach to predict the annual number of sunspots [133, p. 4]. Given that, if we want to forecast a TS we need to have at least three things: 1) historical information of the phenomena; 2) that information can be quantifiable, and 3) we need to assume that the patterns of the past will continue in the future. As a reason of the last point, [134] comments that extrapolation is unreliable because we need to trust in this assumption and consequently it is natural to think that not always we are going to have the same behavior in a TS. For that reason, one could think that the information of the past can not describe the future in an accurate way, because we can assume that everything is in continuous change, but we know that the history repeats itself in any or another sense [135]. Then, under this assumption, we can expect to obtain accurate predictions using historical information.

To understand how a model behaves we may focus in two parts: A) Moving Average (MA) models and B) Autoregressive (AR) models. The first one focuses in the inputs of the system, i.e. given a TS e_t, it is modified to produce another series x_t using the present and N past values of e_t. Thus, one input vector can produce one prediction value after an equation or algorithm is applied. For the AR model, it is created a feedback in order to use the actual prediction as an input value, which means that the prediction may be extended without having values of the original data. The combination of both models give the well known ARMA model used to model and to predict TS, or better called Box-Jenkins ARMA models introduced by statisticians George Box and Gwilym Jenkins. This model works only satisfactorily with stationary TS, therefore, if the series does not have this characteristic it needs to be transformed into a stationary TS.

Later appeared the generalization of the ARMA model specialized for TS and called ARIMA which was designed for non-stationary data. In both models it is used a small subset of the recent TS information to perform the prediction creating a loop with the output of the model (predictions) to predict recursively any number of values in the future as done with ANNs (see Section 3.1.2).

Examples of MA methods to predict are: Simple Moving Average, Geometric Moving Average, Exponential Moving Average, Polynomial, Logarithmic, among many others. For example, in the simple moving average, the data used as input is smoothed by arithmetically averaging over a specified period to produce the prediction at given step in the future; Geometric Moving Average does the same smoothing by a geometrically averaging. These both methods were the first used in the prediction field and usually produce poor forecasting results. On the other hand, polynomial approximation uses a polynomial equation (of a given order) using previous values to predict a target one, however it could be the risk that a long polynomial expression may over fit the data producing bad results.

3.1.1 Time series behaviour

A TS may be defined as a vector $X = [x_1, x_2, \ldots, x_t]$ with its values sampled at regular intervals. It could be natural, like daily temperature in a given area of the earth, or artificial, like the *Lorenz* TS usually tested in prediction tasks and generated with a set of differential equations.

The TS behaviour may be classified in terms of trend and seasonality. The trend is the consistent behaviour of the series to move in a specified direction, i.e. it can be increased upwards or downwards through time (where time corresponds to the x coordinate).

Different trends may be: *stationary* when the series changes during time but always in a specific range, it is characterized because the series have constant mean, variance and autocorrelation structure; *linear*, *quadratic* or *logarithmic* as approximated with a linear, quadratic or logarithmic function respectively with a mathematical method, e.g. the least squares method. In these cases the mean is the value about which the series osculate and the variance is the amplitude.

If a TS presents some kind of seasonality, it means that consistent behaviours are presented over time, e.g. similar patterns may appear in the same months every year.

3.1.2 Prediction methods

For the TS prediction problem with ANNs, it is common to try to use a small subset of recent TS information to perform the prediction. This method is called lagged variables, or shift registers, or tapped delay line. If we use the prediction (output of the system) as input, we say that we have an *Autoregressive Model*, whereas the input space is called an

Embedding Space. In this case, the TS is transformed into a reconstructed *state space* using a delay space embedding [136, 137], normally called Takens embedding theorem. This means that we are aiming to obtain accurate predictions using only a finite segment of previous values up to the point to be predicted. Thus we have:

$$x_{t+1} = F[x_t, x_{t-k}, x_{t-2k}, \ldots, x_{t-(d-1)k}] \tag{3.1}$$

where d is the number of inputs, k is the time delay and F is the method or algorithm that performs the prediction (the network for this work). There is one condition that needs to be satisfied: given an attractor of dimension D, we must have $d \geq 2D + 1$ [136, 138]. But because we do not generally know D nor the delay, we need to calculate them, for example by using Average Mutual Information for the time delay, and False Nearest Neighbour for the embedded dimension. For specific TS, d and k are given in previous publications, while in other cases one needs to calculate them before the evolution takes place or it may be possible to evolve them at the same time as the network's architecture is evolving (feature evolution, see Chapter 4). Note that the constrain is $d \geq 2D+1$, which suggest that maybe more inputs may be required to correctly predict the TS. There are three general ways to perform the prediction of a TS in terms of the desired number of values to be forecast. Thus, assume the TS X is $[x_1, x_2, \ldots, x_t]$, the number of points ahead to predict is n, the test set is $[x_{t+1}, x_{t+2}, \ldots, x_{t+n}]$, and the forecast in the same interval is $[y_{t+1}, y_{t+2}, \ldots, y_{t+n}]$. In the following examples, we are assuming that the number of inputs (past information) is 3, delays are set at 1 and the prediction step is $\Delta t = 1$.

3.1.2.1 Single-step prediction

The simplest method is just to predict a value in the future, and we may call this method One-step or Open-loop or Single-step Prediction (SSP). It is called Open-loop forecasting because a pattern is used to predict a value and no feedback is used to continue the predic-

Table 3.1: Single-step prediction

Forecasting	*Inputs*
y_{t+1}	x_t, x_{t-1}, x_{t-2}
y_{t+2}	x_{t+1}, x_t, x_{t-1}
y_{t+3}	x_{t+2}, x_{t+1}, x_t
y_{t+4}	$x_{t+3}, x_{t+2}, x_{t+1}$

tions as in a autoregressive method. Table 3.1 shows the single-step prediction method. A sample of previous works that have used SSP are [13, 139–142], where [13, 139, 140] predict the *Lorenz* TS and [141, 142] the *Mackey-Glass* TS. One example of using SSP can be found in Section 3.1.3.

3.1.2.2 Direct prediction

A variation of the SSP may involve varying Δt and fixing the input vectors. Let us call this method direct prediction, Table 3.2.

Applying direct prediction with neural networks means that we need to train the network as many times as there are desired predictions, i.e. we need to train the network to predict the point y_{t+1}, then we need to train it again to predict the point y_{t+2} (rearrange the data set with the new dynamic, i.e. different target values), and so on. Previous experiments from the research have indicated that the same network may with a similar accuracy solve different values of Δt for the same TS with a corresponding degradation of the prediction as Δt is incremented. However, even though this method is commented upon here, it is not used in this thesis as it means training the same network n times to predict n values.

Table 3.2: Direct prediction

Forecasting	*Inputs*
y_{t+1}	x_t, x_{t-1}, x_{t-2}
y_{t+2}	x_t, x_{t-1}, x_{t-2}
y_{t+3}	x_t, x_{t-1}, x_{t-2}
y_{t+4}	x_t, x_{t-1}, x_{t-2}

Table 3.3: Multiple-step prediction

Forecast	*Inputs*
y_{t+1}	x_t, x_{t-1}, x_{t-2}
y_{t+2}	y_{t+1}, x_t, x_{t-1}
y_{t+3}	y_{t+2}, y_{t+1}, x_t
y_{t+4}	$y_{t+3}, y_{t+2}, y_{t+1}$

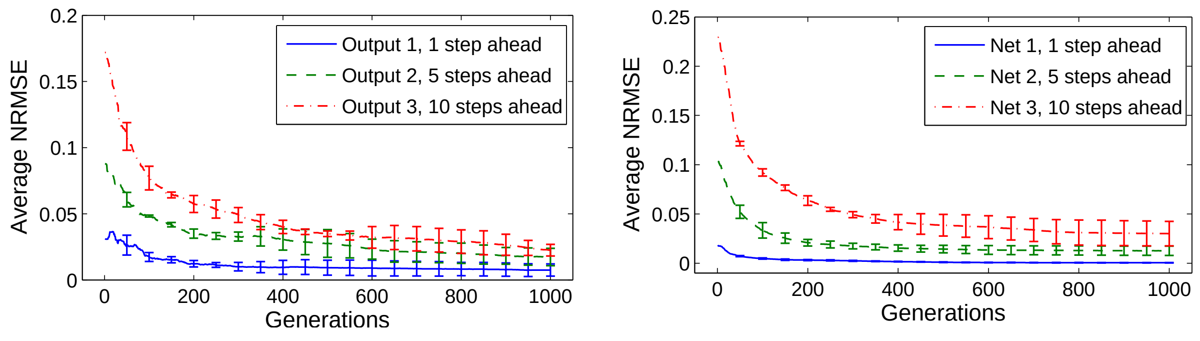
3.1.2.3 Multi-step prediction

Another interesting prediction method is the Multi-step Prediction (MSP) which uses closed-loop forecasting through an autoregressive method as shown in Table 3.3.

Note that in Table 3.3 the predictions are used as input values in subsequent predictions, i.e. it is repeated one-step prediction several times, using the actual prediction to predict the next value. The input vector from the SSP (Table 3.1) and MSP (Table 3.3) methods may be seen as a *window* of d values with k delays that is moved one position ahead every time a value is predicted, to be ready to predict the next value. The real difference between both methods is that the SSP moves the *window* input vector over the original data available, meanwhile the MSP starts with the original data, overlap original and predicted data, and finish with predicted values in the *window* input vector.

As can be seen, direct prediction is the same as SSP with the difference that in the second Δt is usually set to 1, while in direct prediction the input vector is fixed all the time and Δt varies in each iteration (to increase the prediction horizon). Finally, feedforward networks that use the MSP method could be seen, in some way, as Recurrent Neural Networks (RNN), because in both variations there is a feed back formed by the predicted values.

Previous publications that used the MSP method are [35, 65, 77, 141–144], where [143, 144] are focused on the *Lorenz* TS and [35, 65, 77, 141, 142] predicting the *Mackey-Glass* TS.



(a) (b)

Figure 3.1: Average NRMSE per generation for the *Lorenz* TS prediction at 1, 5 and 10 steps ahead with a single neural network (Fig. 3.1a) and using three independent networks (Fig. 3.1b)

3.1.3 Prediction horizon

It is understandable that the bigger the prediction step Δt the more difficult the prediction will be. To illustrate this concept, consider Fig. 3.1 which shows the prediction of *Lorenz* TS (Lo_A TS in Section 3.1.4.2) where the aim as to predict 3 points ahead at the same time with the same network (Fig. 3.1a) and with three independent networks (Fig. 3.1b), i.e. it will be predict Y_{t+1}, Y_{t+5} and Y_{t+10}. In Figs. 3.1a and 3.1b it can be seen that the bigger the prediction step the bigger the error, regardless if a single network is used to predict the three values simultaneously, or if three independent networks are used for the same purpose.

3.1.4 Time series for prediction

In the TS forecasting problem, it was found that previous works use different values to generate the TS and different parameters to test the algorithms, e.g. the size of the training and test sets use to be different between previous studies. For example, the *Lorenz* TS may be generated with the fourth order Runge-Kutta method where, it is possible to settle the *time step* to 0.05 [139] or to 0.01 [140]. Other example may be the fitness performance, in [145] the mean squared error (MSE) is used; [139] applies the root mean squared error (RMSE) and [129] prefers to use the *relative error* measure to evaluate the experiments.

The combination of all these parameters makes it difficult to compare one algorithm against previous implementation in the literature. Thus, in order to establish a set-up to validate algorithms of the kind presented in the thesis, we use some common parameters, as in previous publications to generate specific TS. Even though the sample of parameters may be considered small, it is a representative set of chaotic TS to test the prediction problem.

Note that chaotic time series have been successfully predicted with the EPNet algorithm through the evolution of artificial neural networks in two publication derived from this work [67, 131]. In the following sections will be presented different algorithms that predict the TS used during this study.

3.1.4.1 Logistic time series prediction problem

The logistic map TS is generated with the following iterated quadratic equation:

$$x(t+1) = 4x(t)(1 - x(t)) \tag{3.2}$$

The initial condition for this TS was set up at $x_0 = 0.2$ generating 100 values to train and 100 to test for networks with one input and one output units [65, 146].

In [65, 146] the SSP method is used but the first uses Normalized Mean Squared Error (NMSE) while the second uses Normalized Root Mean Squared Error (NRMSE). Note that applying the square root to NMSE gives NRMSE, so that it may be possible to compare both set of works.

Recurrent neural networks are evolved in [146] whereas GMLP architectures are evolved with the classical EPNet algorithm in [65]. Mikolajczak et al. [145] use the MSE; unfortunately this error cannot be converted to RMSE or NRMSE (standard errors used here), so it is not compared [145] in this study. Section 3.4 presents the different error measures found in the literature.

3.1.4.2 Lorenz time series prediction problem

The *Lorenz* TS [147] is a chaotic system generated by a differential equations system. In its creation process with the Runge-Kutta method, the x, y and z components are generated and just the x series is used in the prediction. Its differential equations are:

$$\begin{aligned} \dot{x}(t) &= -\sigma(x(t) - y(t)) \\ \dot{y}(t) &= -x(t) \cdot z(t) + r \cdot x(t) - y(t) \\ \dot{z}(t) &= -x(t) \cdot y(t) - \beta \cdot z(t) \end{aligned} \tag{3.3}$$

where the initial condition of the system is set as $x_0 = y_0 = z_0 = 1$. In a general overview, some studies used Hand designed artificial neural networks (HDANNs) [129, 140, 143, 148, 149] while others use neuroevolution [13] or a diffident heuristic involved in architecture adaptation or parameter optimization [139, 144]. Other general classification will be in terms of publications that use MSP [143, 144] or SSP [13, 139, 140] for the *Lorenz* TS.

In [139] the following values are used to generate the TS with the Runge-Kutta method (order not specified): $\sigma = 10$, $r = 28$, $\beta = 8/3$ and *time step* $= 0.05$. The data sets in [139] consist of 500 patterns to train and 500 to test using the RMSE to predict the x-coordinate of the system with a pseudo gaussian radial basis function (PG-RBF) which has the facility to add nodes to the architecture or delete unused nodes automatically. In [149] an Autoregressive Moving Average (ARMA) neural network is used to predict the same TS with 1127 values for training and 361 for testing, and using Root Mean Squared Error (RMSE). Given such differences in the data sets, only [139] will be considered for comparison purposes, i.e. a TS with those characteristics will be generated.

Dudul [143] generated the series with the fourth-order Runge-Kutta method. He used the same parameters as [139] to predict the x-coordinate of the *Lorenz* attractor with a regularized neural network-base ARMA (ANN-ARMA) and different linear predictors

(AR, ARMA and State-space) using the MSP method. The training and data sets were of different sizes from those of other publications: 1000 samples for training and 100 for testing at different prediction steps (Δt). The error was measured by performance in percentage (equation 3.11 on Section 3.4.7).

As previously remarked, the differences in the parameters make it difficult to set-up the experiments, as seen until now. Other examples are found in [139] which uses $\Delta t = 1$ whereas [143] uses more prediction steps to test the same TS with different data sets and different error measures. In [129], the importance of choosing an adequate number of inputs and delays for TS forecasting is pointed out. However, they use different parameters to generate the TS: *time step* $= 0.01$, $\sigma = 16$, $r = 45.92$ and $\beta = 4$ over HDANNs. In [13, 118, 147, 148] is used the following parameters: *time step* $= 0.02$, $\sigma = 16$, $r = 45.92$ and $\beta = 4$; 1500 data points are generated where 500 are for training, 500 for validation and 500 for testing.

There may be cases in which some parameters are not clear (or not stated) as will be now demonstrated. In [144] $\sigma = 10$, $r = 28$ and $\beta = 8/3$ are used using different values of Δt with the MSP method, but the *time step* is not mentioned, whereas [140] uses the configuration presented in [139] with *time step* $= 0.01$. Here, 1500 values are used to train and 1000 data points to test but it is not indicated whether MSP or SSP is used.

In the last scenario, the combination of all these parameters may create some confusion, inconsistencies and errors in previous publications. In [150] recurrent neural networks are used to predict the *Lorenz* TS using the following parameters: $\sigma = 10$, $r = 28$ and $\beta = 8/3$; the *time step* is not given but it is referred to [140] which uses 0.01 as *time step*, and thus, it is assumed that such value was used during the generation of the series. Despite the missing values in the generation of the Lorenz TS, [150] compares its algorithm against [139, 149], but these studies used a *time step* of 0.05 in their experiments. Moreover, [150] uses 1500 data values to train and 1000 to test which is different from [139, 149]. Similar issues were found in [150] for the *Mackey-Glass* TS against previous publications [151]. Given those

Table 3.4: *Lorenz* time series generated

TS name	*Method*	*Reference*	Δt	σ	r	β	*time step*	Data set	
								train	*test*
$Lo_A^{(ssp)}$	SSP	[140]	1	10	28	8/3	0.01	1500	1000
$Lo_{B1}^{(ssp)}$	SSP	[139]	1	10	28	8/3	0.05	500	500
$Lo_{B2}^{(msp)}$	MSP	[143]	1	10	28	8/3	0.05	1000	100
$Lo_{B3}^{(msp)}$	MSP	[143]	50	10	28	8/3	0.05	1000	100

inconsistencies in [150], it is not possible to compare our algorithms against it and thus, it will not be used for comparison purposes.

Table 3.4 presents four different implementations of the *Lorenz* TS, where the first two use SSP and the other two the MSP method. Table 3.4 shows two different TS in term of parameter generation (Lo_A and Lo_B) using a different *time step*. But Lo_B has been subdivided in three categories ($B1$, $B2$ and $B3$) as different studies use diverse values in some parameters as shown in Table 3.4. As MSP may be more difficult than SSP it is understandable that the last two TS in Table 3.4 have a shorter prediction horizon than in the case of SSP.

3.1.4.3 Mackey-Glass time series prediction problem

Another common TS commonly used to test prediction algorithms is the *Mackey-Glass* TS, its differential equation is:

$$\dot{x}(t) = \beta x(t) + \frac{\alpha x(t-\tau)}{1 + x(t-\tau)^{10}} \tag{3.4}$$

where all publications found at this stage, used a value of $\alpha = 0.2$ and $\beta = -0.1$ while the fourth-order Runge-Kutta method is the preferred method for its generation. The normal interval to predict the Mackey-Glass TS uses $\Delta t = 6$ with 4 inputs and 6 delays: $x(t)$, $x(t-6)$, $x(t-12)$ and $x(t-18)$. It is common to find in the literature two different values for the τ parameter. If $\tau = 17$ the TS is referred as $MG17$ or where $\tau = 30$, as $MG30$.

Table 3.5: Mackey-Glass time series generated

TS name	*Method*	*Reference*	Δt	$x(0)$	Data set	
					train	*test*
$MG17$	MSP	[35, 65, 139, 149, 153]	6, 90	1.2	500	500
$MG17_A$	MSP	[151]	6	0.9	500	100
$MG30$	MSP	[60, 151]	6	0.9	500	100

Prediction has been made with several different methods like EA [13, 35, 65, 77, 138], incremental construction of ANNs [60] with a method called SNC and with HDANNs in [151]. Classifying previous work by the type of prediction used, we may find several that use MSP [35, 65, 77, 141, 142] or SSP method [141, 142]. The last general classification is for predictions for $MG17$ [35, 65, 77, 139–142, 149, 151] and (less popular) the $MG30$ [146, 151].

In the previous section, it was shown that the *Lorenz* TS is generated and predicted with different parameters. The *Mackey-Glass* TS was no exception: for example [151] uses a initial value of $x(0) = 0.9$ with 500 to train and 500 to test, but [77] uses $x(0) = 1.2$ and 1000 patterns to train and 500 to test. Another example is [35, 65] where the $MG17$ is predicted with $x(0) = 1.2$ and using the fourth-order Runge-Kutta method meanwhile [77] uses the second order Runge-Kutta method and the same initial condition. Another publication that uses MSP and SSP is [151] predicting $MG17$ and $MG30$ TS with 500 values for training and 100 for testing with boosted recurrent neural networks. In [152] it used an adaptive time delay procedure to update the delays and weight of them to predict this TS, but here it is used completely different values to train and test as previous publications. Other publication that used different values to predict this TS are [107, 142].

Table 3.5 presents the TS generated for the *Mackey-Glass*. As mentioned before, previous publications share some common values, $\alpha = 0.2$ and $\beta = -0.1$. But only two works indicate the *time step* used to generate the TS: [35] uses a value of 1 and [140] a value of 0.1. Therefore, α, β and the *time step* are not presented in Table 3.5.

3.2 Classification problems

This work will not be complete if we focus only on prediction tasks, assuming that they are usually more challenging to learn and to solve accurately than classification tasks (see below). In order to better appreciate module evolution and reuse, prediction and classification tasks will be used in the course of this thesis. Three different classification tasks, available in the UCI machine learning repository, are presented in the following sections. All of them have an output per class which allows us to measure the classification error with the "winner-takes-all" method.

We have seen that the prediction of TS requires us to construct a new space (*state space*) in which each point corresponds to a vector x composed of d time series values with k delay lags (equation 3.1). Thus, we have an input vector formed by previous values of the series and a target value $x_{\Delta t}$ being the desired prediction, i.e. a pattern or sometimes what is called a labelled sample. Several patterns (input and target vectors) may be called data sets, and thereafter, the data sets may be subdivided into training, validation and test sets each containing a different number of patterns (see Section 4.2).

In classification tasks, the history of pattern creation is different. Whether the classification problem is artificial or natural, the patterns in data sets are already organized as input vectors describing the object, and the target vector indicating the class to which it belongs. Note that prediction tasks (SSP or MSP) usually require one target value, which is the point to be predicted; thus, the networks may have just one output. Meanwhile, classification tasks normally have one output per class.

Classification tasks can be considered similar to those using the SSP method as one input pattern produces one output value (or vector value in case of several outputs); note that in MSP the actual prediction already calculated is used as input to predict the next value. As seen before, SSP is easier to predict than MSP, so that, it can be expected that some classification tasks may be easier than some prediction tasks. Moreover, the outputs of classification tasks are usually encoded as binary numbers and the performance error does not need to reach a value of zero to correctly classify all patterns in the validation and test sets.

It is of course true that some classification tasks are more difficult than others, like the *Optical digits* (Section 3.2.2) compared against the *Breast cancer* data set, and that could be in terms of the classification boundaries, the number of nodes and connections required to solve the problem, or any other factor involved in it, e.g. the level of noise involved. Even though all this is characteristic of classification tasks, it is quite clear that the human brain solves prediction and classification tasks, and it is expected that certain regions of it contain modules for specific tasks [4]. Nevertheless, in this study will be limited to test separately prediction and classification tasks, i.e. combined tasks (Section 3.3) will be formed only with predictions tasks or with classification tasks, but not mixed between them.

3.2.1 Breast cancer data set

The *Breast cancer* Wisconsin data set was used to perform comparisons against other algorithms. This data set comes from Dr. William H. Wolberg, the University of Wisconsin Hospitals, Madison. The *Breast cancer* data set contains 699 examples formed with 9 inputs and 2 outputs. The aim is to classify benign or malignant tumors based on cell descriptions gathered by microscopic examination where 458 are benign examples and 241 are malignant examples.

In [35] the first 349 examples are used for training, the following 175 examples for validation and the final 175 examples for testing the performance of the EPNet algorithm. Other work that uses an EA is [154] using an algorithm called OC1-GA. The OC1-GA algorithm uses an *ad-hoc* combination of hill-climbing and randomization with a GA to correctly classify this data set. Even [155] does not uses a formal EA, a constructive algorithm called FNNCA is proposed to create the networks, while HDANNs are used in [156]. Another variation is the found in [157] where an optimization method is applied to train feed-forward ANNs with three layers to classify this data set.

In the experiments developed in this work it was found that 16 attribute values had been omitted. That gives 444 benign and 239 malignant examples. For the experiments 341 samples were used for training, the following 167 for validation and the next 175 for testing. For the final training after evolution was complete, the training and validation sets were used.

3.2.2 Optical Recognition of Handwritten Digits data set

The *optical digit* data set was created from handwritten digits with 64 inputs to classify 10 digits (0-9). There are 3823 samples available for training and 1797 for testing.

In [158] only 60 inputs are used to perform the learning with an algorithm that is an improvement on the linear discriminant analysis in combination with evolutionary strategies called EWLDR. Another publication that classifies it is [154], introduced with the *Breast cancer* data set in the previous section.

3.2.3 Thyroid disease data set

The *Thyroid* data set was created form the 'ANN' version of the 'thyroid disease' data set. There are available 7200 examples in two data sets, 3772 for training and 3428 as a test set. The data set is composed of 21 attributes to classify in three different categories (3 outputs), i.e. to determine whether a patient referred to the clinic is hypothyroid: normal (not hypothyroid), hyperfunctional and subnormally functioning.

In [35] the first 2514 examples from the training data are used to train, the rest from the same data set are used to validate, and all the test set is used to evaluate the networks. Others works that classify it are [156, 158, 159], where a brief introduction has been given in the previous section for the last two, and [159] uses HDANNs, training MLPs with a different number of hidden nodes.

3.3 Data sets for module evolution and module reuse

Previously, we have seen different TS and data sets designated for a particular task at the time, i.e. one TS is predicted at a given interval or one data set is classified into different classes. In this section will be presented the combination of two or more tasks (prediction or classification) to be solved by the evolution of neural networks, in which in principle modules for each task may emerge if modular architectures are beneficial for the problem.

3.3.1 Artificial classification tasks

One kind of data set used to test the algorithms of chapters 5 and 6 has been generated with random points over a two dimensional input space. Those points (x and y axis values) correspond to two input values which in turn will be considered the networks' inputs. They were generated in the range [0,1] for two distinct classification tasks as shown in Fig. 3.2, i.e. the networks have to learn the classification boundaries in the two dimensional input space for each output task. For example, consider that two random inputs generated into the two-dimensional space are the point in coordinates (0.5, 0.9) in Fig. 3.2; thus, the class of the first task is 2 and the class of the second task is 3, as coordinate (0.5,0.9) falls into these classes for both tasks.

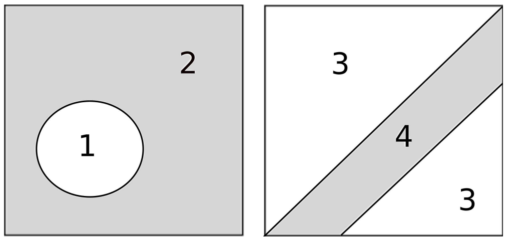


Figure 3.2: *A1-B1* data set. Artificial data sets formed by random points in a two dimensional input space and two outputs corresponding to distinct classification tasks

This data set has been studied previously and shown to result in modular MLP architectures evolved, if the fitness is measured in terms of the time required to learn optimal generalization [12]. Here, 1000 such patterns were used for training, 200 patterns were used for validation during evolution (to provide a measure of fitness), and 200 patterns were used to test the networks after the evolution was finished.

The same procedure to create the data set of Fig. 3.2 could be repeated with different classification boundaries. Fig. 3.3 shows 12 classification tasks with simple decision boundaries where the first 9 (data sets A1 to C3) are formed by two classes and data sets D1 to D3 containing 3 classes per task. In this way, the classification tasks from Fig. 3.2 are formed by Fig. 3.3a in the first task and Fig. 3.3d in the second task, where it can be called data set *A1-B1* as it is formed by those data sets in that order.

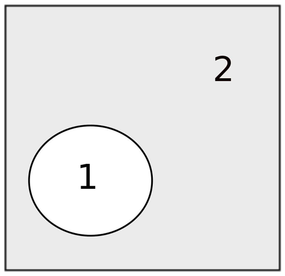
A second data set found in [12] was the one formed by data sets $C1$ and $D1$ (*C1-D1*), finding modular architectures when the connectivity is constrained during evolution.

The same criteria to generate classes $A1$, $A2$ and $A3$ could be repeated to form a class called $A4$, changing the position of class 1 from data set A1. The same procedure may be followed in the case of any other class in Fig. 3.3 to generate more classification boundaries. However, the classification tasks presented so far may be enough for this work investigating the module evolution and module reuse.

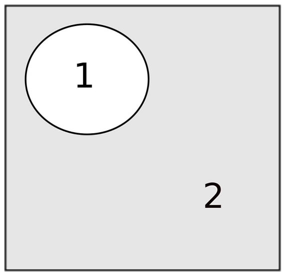
3.3.2 Data sets with multiple tasks

The idea of putting together two data sets to evolve modular neural networks [12], has been adopted in this work to investigate more deeply the modular evolution and reuse of neural networks. With this representation it is possible to put more than two data sets together to be solved with a evolved modular network, e.g. data set *A1-B1-C1*. This idea can be applied too to other data sets, for example, combining the *Thyroid* data set with the *Optical digit* recognition problem, thus making a *Thyroid-Optdigt* data set.

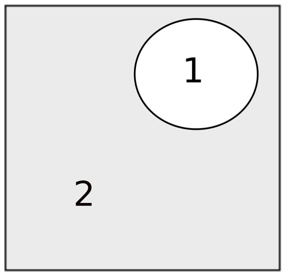
At this point, the kind of task combined will not have a matter of concern, e.g. *Thyroid-Optdigt* data set could not be a tasks that we normally solve in a real world scenario, but for research purposes, that kind of combination will be allowed in this work to study the module interaction during and after evolution. Clearly, the more different two task are, the bigger



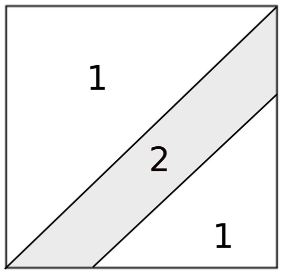
(a) A1



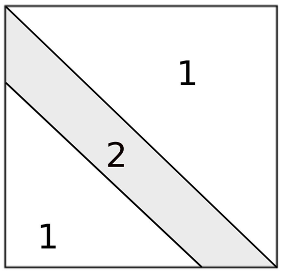
(b) A2



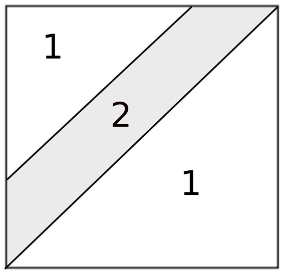
(c) A3



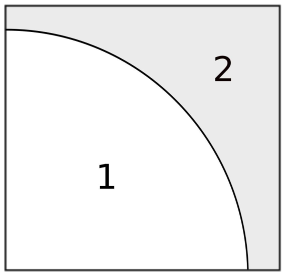
(d) B1



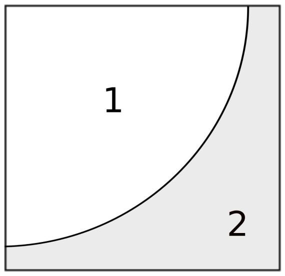
(e) B2



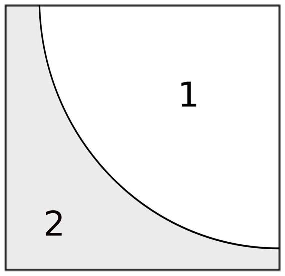
(f) B3



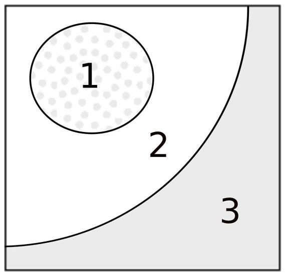
(g) C1



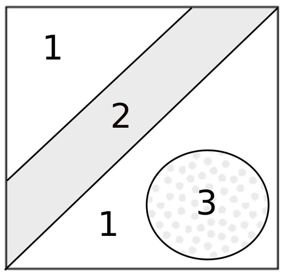
(h) C2



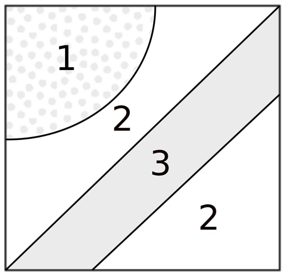
(i) C3



(j) D1



(k) D2



(l) D3

Figure 3.3: Artificial data sets generated for classification tasks. Nine data sets composed of 2 classification tasks (Figs. 3.3a to Fig. 3.3g) and 3 classification data sets composed of 3 classes each (Figs. 3.3j to Fig. 3.3l)

the modularity value expected during evolution, i.e. if two task share many similarities (patterns, trends, etc.) one may assume that they could present less interference during learning, consequently, lower modularity values will be obtained after evolution, which means that no pure-modular architectures may solve both tasks. However, that is still an assumption and needs to be investigated empirically (chapters 5 and 6).

The same idea could be applied in the TS prediction field, e.g. $Lo_A^{(ssp)}$-$Lo_B^{(ssp)}$ or $Lo_{B2}^{(msp)}$-$MG17$, or using the same TS and creating several data sets that predict at different intervals, i.e. different values of Δt like Fig. 3.1a where is predicted the *Lorenz* TS with 1, 5 and 10 steps ahead with the same network, thus, considering 3 potential modules that could be created during evolution. The only restriction with prediction tasks is the method to use (MSP or SSP) and the lapse of time to predict (Δt), where both TS must have the same conditions, because both TS are going to be predicted at the same time.

Note that [13] have previously used the combination of *Lorenz* and *Mackey-Glass* with SSP for automatic problem decompositions using a co-evolutionary algorithm. The major difference between [13] and this study, is that in this thesis we are going to be interested in investigating when and where modular networks emerge with SSP and MSP methods for classification and prediction problems, as well the module communication during and at the end of the evolutionary process with a steady state algorithm, whereas [13] is focused on the automatic problem decomposition and not in the interaction between modules.

3.4 Performance measures

This section summarizes different performance measures found in the literature for prediction and classification tasks. They are presented here as different studies used different measures, and because some errors could be converted and stated in terms of other types of error. For example, [143] uses the Fit error (equation 3.11) which may be converted into NRMSE (equation 3.8).

3.4.1 Error percentage

The first error measure is the standard error percentage presented in [156] and previously used in the original EPNet algorithm [35], and defined by equation 3.5:

$$E_p = \frac{100}{Tn(Z_{max} - Z_{min})^2} \sum_{t=1}^{T} \sum_{i=1}^{n} (Y_i(t) - Z_i(t))^2 \tag{3.5}$$

where T is the number of patterns, n the number of output nodes, and $Y_i(t)$ and $Z_i(t)$ are the actual and desired outputs of node i for pattern t. This error has been used during this work to measure the fitness of classification tasks and to evaluate the error in the validation set. Note that the evaluation of the fitness for prediction tasks was obtained from a test set inside the EA using NRMSE, as previous studies have used this measure to test the networks' performance for TS forecasting.

3.4.2 Mean squared error

Mean-square Error (MSE) can be considered one of the first error measures to test the generalization, but also it has been used to quantify the training error inside the learning algorithm. This measure was found in [145, 150] and it is expressed by equation 3.6.

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (Y_i(t) - Z_i(t))^2 \tag{3.6}$$

where N is the number of patterns in the target vector. This error will not be adopted in the thesis as it may not be as reliable as others to measure the performance; for example NRMSE is a more robust error measure.

3.4.3 NMSE

Normalized Mean Squared Error (NMSE), also called *arv*, has been previously used in [60, 140, 146, 150, 151] and defined by:

$$NMSE = \frac{\sum_{i=1}^{N}\left(Y_i(t) - Z_i(t)\right)^2}{\sum_{i=1}^{N}\left(Z_i(t) - \overline{Z(t)}\right)^2} \tag{3.7}$$

where $\overline{Z(t)}$ is the mean of $Z(t)$. As can be appreciated, taking the squared root of the NMSE gives the NRMSE.

3.4.4 NRMSE

Normalized Root Mean Squared Error (NRMSE) may be considered a robust measure error as the mean of the series is subtracted from each point so that we are working with true variance [160]. Its equation is defined by:

$$NRMSE = \sqrt{\frac{\sum_{i=1}^{N}\left(Y_i(t) - Z_i(t)\right)^2}{\sum_{i=1}^{N}\left(Z_i(t) - \overline{Z(t)}\right)^2}} \tag{3.8}$$

Previous works that use it are: [13, 35, 67, 131, 161]. This error is adopted as the main performance measure for prediction tasks in this work.

3.4.5 RMSE

The Root Mean Squared Error (RMSE) is a common metric to measure error in prediction [139, 139, 149, 150] and is defined by equation 3.9.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(Y_i(t) - Z_i(t)\right)^2} \tag{3.9}$$

This error cannot be converted to NRMSE, so it too has been used to measure error and to be able to compare the improved EPNet algorithm against previous results in the literature.

3.4.6 Relative error

Relative error [129, 162] was an unusual error measure found in the literature to express prediction error:

$$R_{error} = \sqrt{\frac{\sum_t \left(Y_i(t) - Z_i(t)\right)^2}{\sum_t \left(Z_i(t) - Z_{i+1}(t)\right)^2}} \tag{3.10}$$

As it requires an extra value to perform the measurement ($Z_{i+1}(t)$) this measure of error will not be considered in this work to compare with previous works that used it.

3.4.7 Performance in percentage

This metric was found in [143] and defined as:

$$Fit = \left[1 - \left(\frac{\sum_{i=1}^{N} \left(Y_i(t) - Z_i(t)\right)^2}{\sum_{i=1}^{N} \left(Z_i(t) - \overline{Z(t)}\right)^2}\right)\right] \times 100 \tag{3.11}$$

Note that this equation also may be converted to NRMSE.

3.4.8 Classification error

The winner-takes-all method was used to determine classification error. In this method, the output with the highest activation designates the class, and thus, it is assumed that there is an output node per class in the network. After calculating the correct number of classification, it is possible to measure the percentage of correct classifications with an standard percentage formula [156].

3.4.9 Correlation as error

It may be important to remark that [149] uses RMSE as a performance measure but also uses the correlation between the prediction and the original data from the TS. Nevertheless, the correlation may not be as precise as in the case of other error measures, such as NRMSE. For example, consider Fig. 3.4 which presents the prediction of the $MG30$ TS over 100 steps ahead using the MSP for the best network found after 30 independent runs and using the feature input space fixed during 3000 generations of evolution (details of this experiment may be found in Section 4.3 with a successful training parameter set to 70%). In Fig. 3.4 it can be seen that the correlation value between the prediction and the original data is close to one (Fig. 3.4a), but NRMSE provides another scale to measure the error, which makes it more suitable than the correlation, e.g. if another algorithm improves such prediction, it may be difficult to see by what amount it is improved if the correlation is adopted as a standard error measure. Nevertheless, the NRMSE and correlation values will be only displayed (not in tables) in figures of this study to allow another angle to compare the predictions. As the prediction and original data are quite similar and rather difficult to appreciate in Fig. 3.4a, Fig. 3.4b is presented, which represents the error in terms of $Y_i(t) - Z_i(t)$. There it is made clear how the error started to grow in the last point predicted. Note that in Fig.

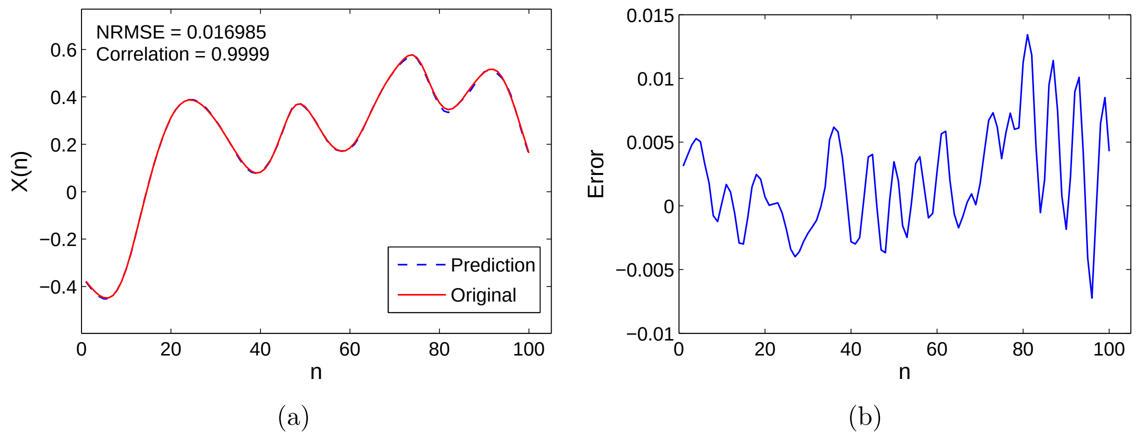


Figure 3.4: Best prediction for the $MG30$ TS with fixed inputs after 3000 generations of evolution. Fig. 3.4a shows the prediction on the final test set and Fig. 3.4b presents the error in terms of $Y_i(t) - Z_i(t)$

3.4a the original data are not plotted, previous to the first value to be predicted. That has been done in the interest of clarity and will be maintained during the rest of the work.

3.5 Summary

This chapter presented the second part of the literature review, showing the TS and classification problems as well as different series for prediction and data sets for classification.

Note that it has been chosen one easy tasks for classification and prediction (*Breast cancer* and *Logistic* TS), as they produce accurately results in early generations, and also it has been selected more difficult tasks to forecast and classify.

The combination of two or more tasks, to generate suitable data sets for the modular evolution, have also been presented. That combination is the key aspect in the evolution of modular neural networks, as the weights from one task may interfere with the other during learning; thus, modular architectures may provide a better fitness during evolution as they present independent partition of nodes. Considering all data sets presented, it was illustrated how similar and different kind of tasks could be combined to test the module evolution and module reuse, i.e. combining two or more classification tasks or combining two or more prediction tasks.

The next chapter will present the first experimental stage of this study which is in charge of showing the EPNet algorithm with different modifications; there were three reasons for doing that: 1) show the basis of the algorithm for its understanding as subsequent algorithms, M-EPNet and module reuse with the M-EPNet, that are based in the EPNet algorithm; 2) study, analyze and optimize or improve some basic parameters and methods of the EPNet to make it a more suitable and general algorithm and 3) to make a comparison of the EPNet algorithm against previous studies in the literature, providing a robustness of the improved algorithm and obtaining a baseline results that will be used in comparison against the modular implementations.

Chapter 4

The EPNet algorithm

This chapter will introduce the experimental analysis of the EPNet algorithm [35, 65] to understand how neural networks are evolved with a steady-state algorithm and using only mutations to carry out the evolution, in addition of the explanation provided in Section 2.2.1 for this method. The EPNet algorithm has not been extended for several years, and never tested in simple scenarios like the evolution of features needed for prediction and classification tasks. Thus, in this chapter the experimental analysis starts with optimizing the configuration of the EPNet algorithm (parameters and feature evolution) to allow further extensions in subsequent chapters for the module evolution in Chapter 5 and module reuse in Chapter 6.

The study of the EPNet algorithm is a fundamental requirement to understand its basis at evolving ANNs for single tasks, and also, to clarify how it has been improved for the module evolution. Clearly, this study will not be complete if we just focus in the module evolution and module reuse with the EPNet algorithm, but analyzing its behaviour when single tasks are solved (as done in this chapter) will give the baseline results (for further comparisons) and confidence to explain its behaviour when applied in more complex tasks like the evolution of modular neural networks. Considering that the EPNet algorithm has bee explained in Section 2.2.1, Section 4.1 introduces the general behaviour of the standard EPNet algorithm evolving the *Thyroid* data set, followed by the information organization (training, validation and test sets) in Section 4.2. In terms of new contributions over the EPNet algorithm, Section 4.2 also presents why TS predicted with the MSP method require to use an extra validation set during evolution; after that, Section 4.3 shows a detail study of a parameter called “successful training” that may damage the evolution if it is not configure correctly. Thereafter, the last contribution of this chapter is presented in Section 4.4 with three improved versions of the EPNet algorithm to evolve the input feature selection, where the feature evolution is evolved differently in each version. Thus, the standard EPNet algorithm and its improved versions are compared in Section 4.5 and a final comparison against previous algorithms in the literature is presented in Section 4.6, to show the robustness of the improved algorithms.

4.1 The standard EPNet algorithm

It has been said that the the EPNet algorithm gives preference to smaller architecture. To corroborate that empirically, consider Fig. 4.1 where the evolution of the *Thyroid* data set is carried out over 3000 generations. When the algorithm starts with bigger values, e.g. hidden nodes, the algorithm first decrements them (Figs. 4.1c and 4.1d in the first 300 generations) as needed, as in a pruning algorithm, because mutation deletions have a bigger probability than additions as explained in Section 2.2.1. When it reaches a bottom line in the number of parameters, as more deletions do not help to reduce the error any further, it starts to increase the number of parameters (like constructive algorithms) looking for bigger networks that could possibly solve the problem better.

The same behaviour has been reported in the original implementation of the EPNet algorithm [35]. Nevertheless, there has not been any previous comment on the increment in the parameter deviation as the generations advance, as seen at the end of the evolution for the hidden nodes and connections in Figs. 4.1c and 4.1d. This is a characteristic behaviour

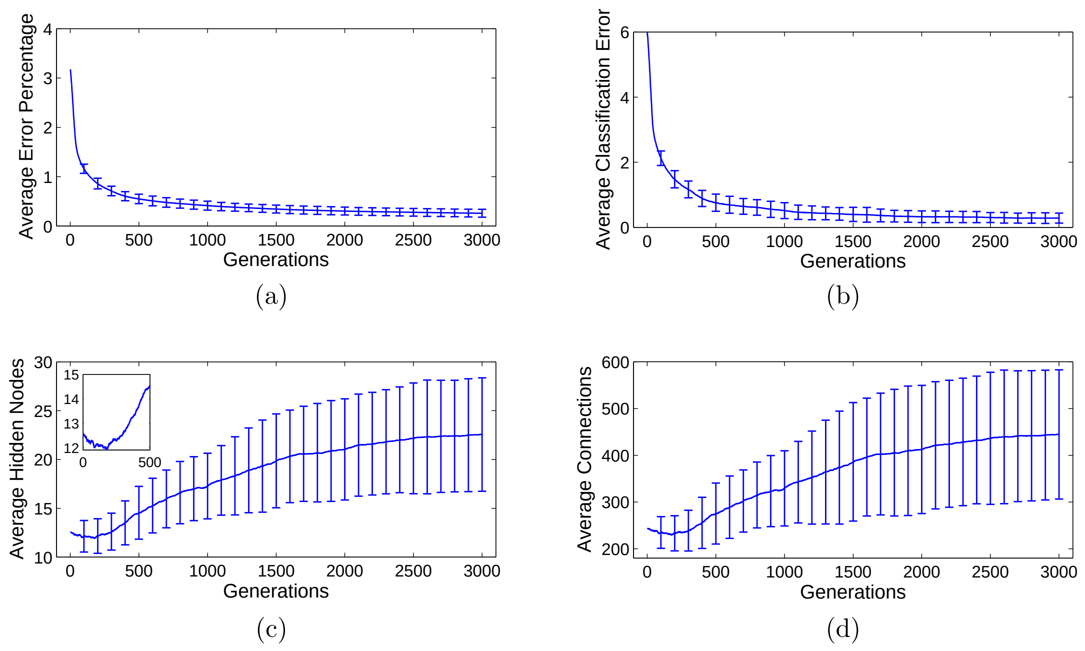


Figure 4.1: Behaviour of the EPNet algorithm during 3000 generations of evolution for the *Thyroid* data set over 30 independent runs. Fig. 4.1a shows the average error percentage and Fig. 4.1b the average classification error. The average number of hidden nodes and connections are shown in Figs. 4.1c and 4.1d respectively

noted in all experiments carried out for prediction and classification tasks during this study. It is also notable that the performance of the *Thyroid* data set (Figs. 4.1a and 4.1b) is not affected when the networks' architectures are decremented (the first 300 generations) or incremented, nor by the bigger deviation in the parameters which indicates that the diversity of individuals is getting bigger, a desirable effect in EANNs.

4.1.1 Evolution of learning rate per node

It has certainly been found elsewhere that evolving the learning rate for specific parts of the network can improve the performance of MNNs [12] over simulated evolution with a generational algorithm. Thus, it is reasonably expected that evolving a single parameter for each node may improve the performance of the EPNet algorithm over MNNs. However, first attempts to evolve such configuration with just single mutation operators over the EPNet algorithm led to worst results in prediction tasks cause by a destabilized learning during training. For classification tasks the noise introduced was not as severe as for prediction tasks. That indicates that this topic may be addressed with more care in further implementations. Hence, this chapter uses the normal evolution of the learning rate (one parameter per network as introduced in Section 2.2.1.2) while a learning rate per task (module) is evolved for the the rest of this study with MNNs.

4.1.2 User-specified parameters

There are several parameters in an EA that need to be settled before the evolution starts. In this case, some of them were chosen from several preliminary experiments intended to chose the most appropriate values, others like the number of individual in the population were set at convenient traditional values and are not intended to be optimal.

Learning rate (η): initial learning rate 0.15, minimum learning rate 0.01, epochs for learning rate adaptation 5, and learning rate update set at 0.02.

Connection scheme: the population of networks was initialized before the evolution starts with a simplified connection scheme set using the same probability ϕ for each connection in a connectivity matrix C, in this chapter all networks used a value of $\phi = 0.5$ allowing the networks to add or delete connections from early generations. More information about this parameter can be found in Section 5.2.1.

Bias: the bias was another parameter included in the evolution, where any node can have associated a bias to it or not. The weight value of each bias was evolved as any other weight in the network.

Mutations: maximum number of mutated hidden nodes 3, e.g. 1 to 3 hidden nodes can be mutated when the node deletion mutation is applied; maximum number of mutated connections 3. In the case where the input feature selection is evolved (Section 4.4): maximum number of mutated inputs and delays are set at 2. Even those values were used here, it is important to say that depending upon the problem at hand those values may be adjusted, e.g. for small networks architectures (requiring fewer than 50 nodes) the maximum number of hidden nodes that could be added or deleted may be 1-3, but if we are dealing with problems that require hundreds of nodes it may be a good idea to set those parameters within a bigger range. Clearly those ranges too may be evolved in future implementations, but they are outside the scope of this work.

Epochs of training: for classification tasks like *A1, B1, ..., A1-B1,* and so forth, it was enough 25 epochs of partial training. Data sets like *Breast, optical digit* or *Thyroid* have 100 epochs of partial training. Prediction tasks were harder to be learned, thus, all of them uses 1500 epochs in this parameter.

SA: 5 temperatures were used in the SA algorithm with 100 iterations per temperature.

EA: population size of 30 individuals with a maximum of 5000 generations of evolution allowed, being the number of generations one stopping criterion. For prediction tasks, if the error of the best individual is not reduced every 300 generations the algorithm stops whereas classification tasks stop the evolution if no further improvement is found every 100 generations. The differences in those parameters are by the fact that prediction tasks were more difficult as previously remarked. Another stopping criterion in classification tasks is given by the classification error, if one individual classifies all patterns in the validation set, the evolution finish automatically.

Note that several tasks continue decrementing their fitness through generations, which makes difficult to correctly settle a generation gap to stop the algorithm if no further improvement is seen. For that reason, some experiments were run during 3000 generations being the generations the only stopping criterion, to obtain figures that shown the evolution of parameters with the same number of generations in all independent runs.

Thereafter, the same experiments were run allowing to stop the algorithm at any moment to investigate how many generation of evolution on average are suitable to produce satisfactory results, avoiding to waste unnecessary generations if the error can not be significantly reduced in subsequent generations. To clarify one particular case where the error is continuously reduced during evolution, consider Fig. 4.2 which presents the worst, average

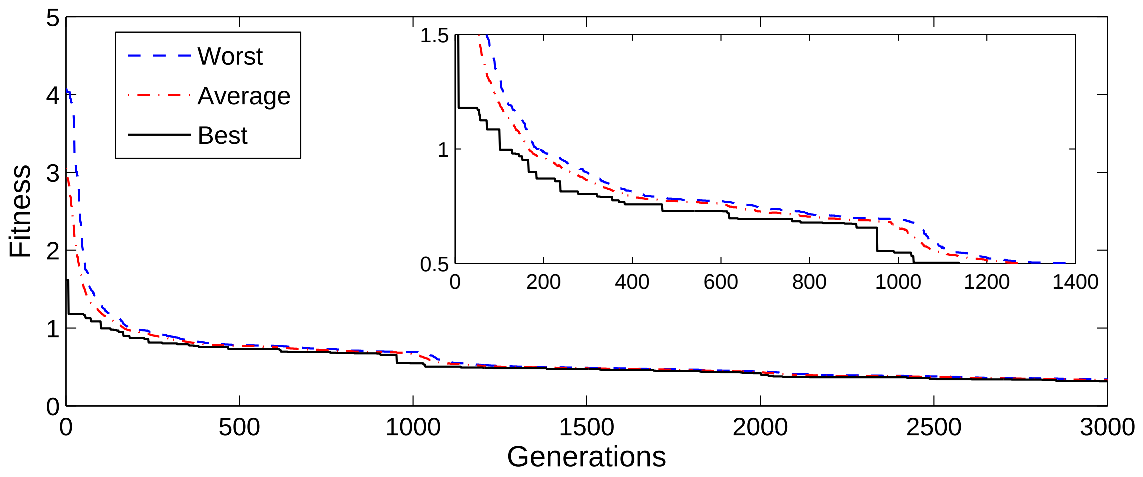


Figure 4.2: Worst, average and best fitness values per generation for the *Thyroid* data set during 3000 generations of evolution for one independent run. A zoom during the first 1400 generations is presented to distinguish the different trends

and best fitness per generations for one independent run for the *Thyroid* data set during 3000 generations of evolution, this figure is related to the experiment shown in Fig. 4.1. In Fig. 4.2 can be seen that in early generations, the best error found so far is reduced several times, but less frequently as generations advance. The best individual showed the last reduction in its error around generation 2900, which is not statistically significant if compared against the last 2000 generation, but clearly it may continue its reduction if more generations are allowed. Another fact to note here is that the fitness of the entire population (average fitness) converge to similar values during evolution, clearly an expected behaviour for an EA.

As standard in this study, each evolutionary experiment was repeated 30 times to allow a reliable statistical analysis of the results. After each evolutionary run had finished, the best individual evolved was identified by measuring the NRMSE (in prediction tasks) or Error percentage (in classification tasks) on an independent test set, and the error, inputs, hidden nodes and connections among other parameters were recorded for performing the comparisons. T-test analysis (two-tailed with unequal variances) was carried out at 3 different scales as commonly used in several publications, where p values of less than 0.05 are considered almost significant, $p < 0.01$ significant and $p < 0.001$ highly significant.

4.2 Training, validation and test sets

The size of the TS and data sets was used as stated in Chapter 3 and split into four sub-sets for prediction tasks and 3 sub-sets for classification tasks.

Before explaining how the data set are divided, remember that prediction tasks with SSP are similar to classification tasks as one input vector produce one output vector and there is no feedback in the output as in MSP. Having said that, a standard procedure in the literature is to evaluate a validation set with the SSP method for classification and prediction tasks. Moreover, any publication has been found so far that uses MSP over the validation set, that kind of evaluation is never said and it may be assumed to be SSP as it is the standard.

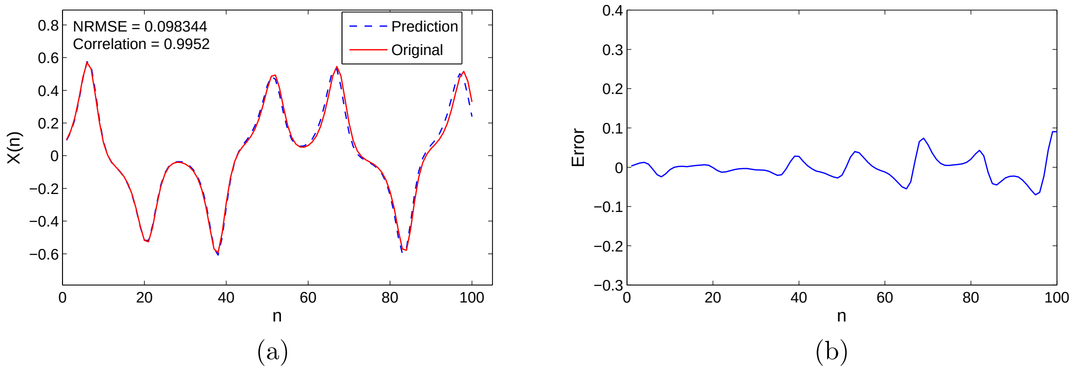
Prediction case. The first set is the "training set" that is used to perform the learning task with MBP or SA; then there is a "validation set" that is used to ensure that there is no over-fitting of the learning and used to evolve the learning rate, then a "test set inside EPNet" to simulate a real prediction (MSP or SSP) and obtain the fitness of the networks, and finally there is the "final test set", that is only applied after the whole evolutionary process has been completed, to evaluate the final individuals on the generalization performance. During this work it is going to be called *pre-validation set* to the previous validation set and *validation set* to the "test set inside EPNet" to avoid confusions.

Classification case. In classification tasks there are just training, validation and test sets as normally done for this kind of task. Here an extra test set is not required like in the prediction case because the validation set uses the same method to evaluate the fitness during evolution as in the generalization test, i.e. like the SSP method.

For prediction tasks, it was decided to use an extra test set during evolution as tasks solved with MSP require to validate the generalization performance with the same method (MSP) when the evolution finish. Thus, the networks may not be correctly evaluated (misleading fitness) if the single SSP is used during evolution and then the MSP at the end of it. For example, in Section 3.2 it was remarked that SSP method is easier than MSP by the feedback in the later, therefore, if a prediction task requiring MSP is evaluated with SSP during evolution (validation set), it will probably produce a different fitness than if the MSP is used in the same validation set, which could produce a bias in the selection process with networks not so fit. For that reason it was needed to use an extra test set in prediction tasks, so the pre-validation set is used mainly to evolve the learning rate and the validation set to measure the fitness as it were a real prediction.

To illustrate this, consider Fig. 4.3 for the best predictions found for the $Lo_{B2}^{(msp)}$ with MSP (Figs. 4.3a and 4.3b) and SSP (Figs. 4.3c and 4.3d) on the validation set during evolution and using MSP on the test set. Interestingly to note (and as previously remarked), the average fitness of the networks evaluated with SSP on the validation set have a lower error during all the evolutionary process than the fitness obtained from the MSP as can be seen in Fig. 4.4. The best prediction error on the validation set at the end of the 3000 generations of evolution were smaller with the SSP than with the MSP as expected too (no shown here). It was also obtained a smaller error with the SSP on the average fitness over all independent trials as shown at the end of generations in Fig. 4.4. At the end of the evolution, even the network that uses MSP have a bigger fitness error in the validation set, it obtained the smallest generalization error (Fig. 4.3a) because the selection mechanism during evolution was in the same terms as the generalization measurement.

As a general rule, tasks that use SSP will use SSP for the fitness during evolution and to evaluate the generalization performance. Contrary, tasks that use MSP will use the same MSP method in both parts of the process. When the whole evolution has finished, the networks will be further trained with the combined training and validation as done in the first implementation of the EPNet algorithm [35]. Thus, the validation(s) and test set inside the EPNet are unseen data sets by the network during the partial training and throughout the evolution. At the end of all generations, the only unseen data set is the final test set used for generalization purposes. During training, the patterns of the training set are reorganized randomly every epoch (stochastic learning) using the sequential learning to



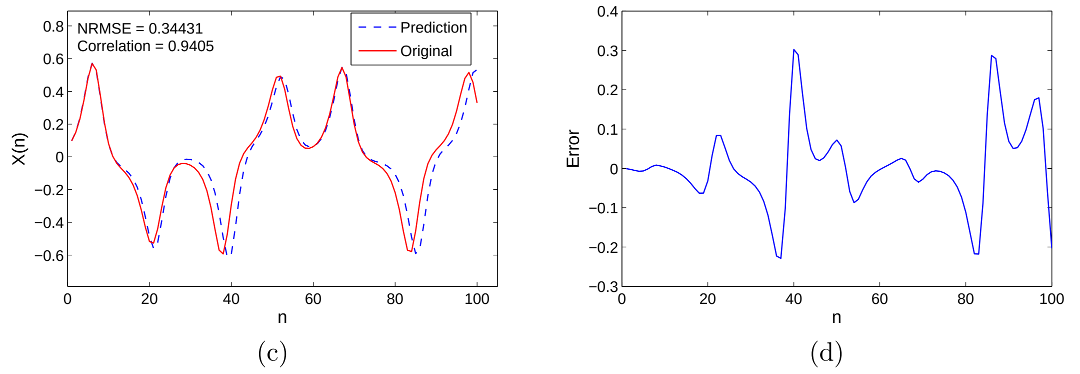


Figure 4.3: Best predictions found for the $Lo_{B2}^{(msp)}$ with MSP (Figs. 4.3a and 4.3b) and SSP (Figs. 4.3c and 4.3d) on the validation set during evolution and using MSP on the test set after 3000 generations. Figs. 4.3b and 4.3d present the error in terms of $Y_i(t) - Z_i(t)$

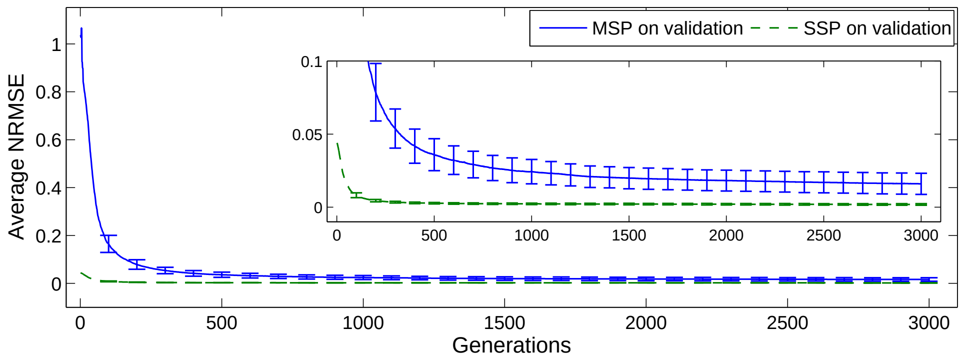


Figure 4.4: Average fitness value of $Lo_{B2}^{(msp)}$ with MSP and SSP over the validation set

avoid local minima with a gradient descent algorithm like the BP algorithm, which is an advantage over others methods like Conjugate Gradient or quasi-Newton Algorithms [51, p. 255].

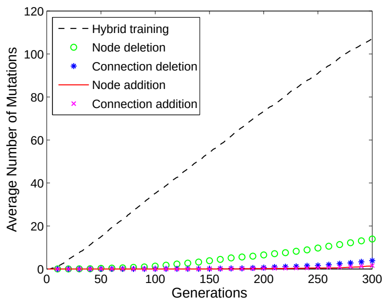
4.3 Successful training parameter (STP)

As mentioned in Section 2.2.1.1, there is a crucial parameter in the EPNet algorithm that determines what is called *successful training*. We have "success" if the training error is decreased "considerably", or "failure" if it is not. In the literature, this parameter is never discussed in detail (i.e. to determine how much is "considerably"), and it can easily be set with incorrect/inappropriate values. The EPNet algorithm proves to be much more robust with regard to its other parameters.

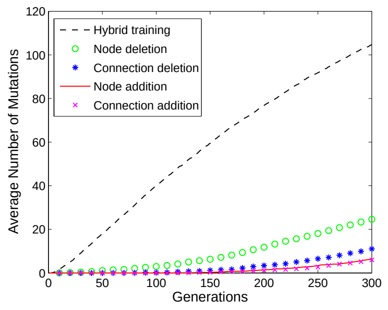
Consequently, our study began by running several experiments with different values for the Successful Training Parameter (STP): 30%, 50% and 70% for prediction and classification tasks. For example, for the 70% value the training is marked as "successful" if the error is reduced by 70% or more (a strict value), and for the value 50% it is marked as "successful" if the error is reduced by half or more (a more relaxed value). It was found that this parameter has a major impact on the performance of the algorithm for prediction tasks, because if we use a value which is too relaxed (e.g. 30%, with the error needing to be decreased by only 30%) the networks enter the training stage and easily achieve a sufficient reduction of the training error (leading the training to be marked as "successful"), and thus pass directly to the next generation, without allowing the architectural mutations to take part in the evolutionary process.

That may produce networks with a poor performance at the end of the evolution (i.e. bigger errors are obtained) because hybrid training was used more times than other mutations during evolution.

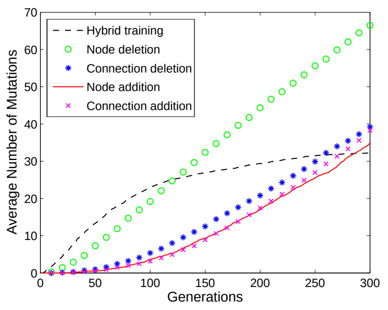
Fig. 4.5 presents the average mutation rates over the entire evolution of 300 generations for the *Mackey-Glass* TS with three values in the *successful training* parameter (set to 30%, 50% and 70%). In Figs. 4.5a and 4.5b it can be seen that the hybrid training dominates the evolutionary process, with the other mutations used only a few times. Conversely, in Fig. 4.5c it can be seen how the other mutations are used more frequently if a strict parameter value is set (i.e. 70%), which shows that there are more modifications in the architectures



(a)



(b)



(c)

Figure 4.5: Average number of mutations for *Mackey-Glass* TS. *Successful Training* parameter set to 30% (Fig. 4.5a), 50% (Fig. 4.5b) and 70% (Fig. 4.5c). Similar trends were found between $MG17$, $MG17_A$ and $MG30$

than with a relaxed value. That is clearly a desirable behaviour if we want to look for more solutions in the search space. Analysing this issue for the other TS revealed that the TS behavior has a major effect on the evolutionary process. For example, for the *Logistic* TS the average mutation rates arising for the three different parameter values (30, 50 and 70%) were similar (using all the mutations in the evolutionary process) to the patterns seen in Fig. 4.5c, i.e. there is no negative effect if a relaxed value is used for this TS. In other words, for this TS, this parameter was not so crucial. However, that result is clearly influenced by the fact the the *Logistic* TS has an easy dynamic, making it easy to predict. Consequently, low error rates are reached with fewer epochs of training than in other TS (*Mackey-Glass* and *Lorenz*), allowing the use of other mutation operators during evolution. Similar trends as presented in Fig. 4.5c were found in all the prediction task for the *Mackey-Glass* and *Lorenz* TS if a strict value is adopted.

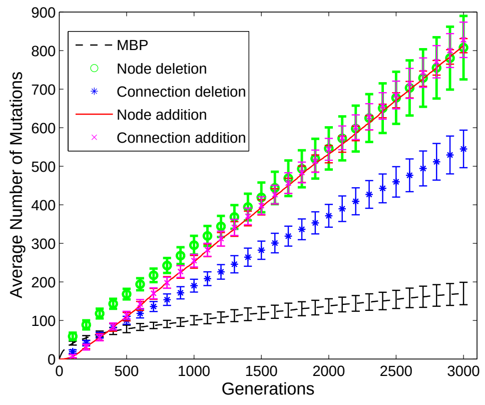
As commented in Section 3.2, some classification tasks are more easily solved than some prediction tasks. Looking at the classification tasks (*Breast, Thyroid, Optical digits, A1, B1, C1, D1 and A1-B1*) the same scenario was found as *Logistic* TS. Analysing the complete set of experiments for classification and prediction tasks and repeating all of them with greater or lesser number of training epochs, it was found that classification tasks combined with any number of epochs (from 25-300) are not affected by any value chosen in the successful training parameter. This is because they can reduce the error significantly with a small number of epochs or in early generation of evolution. Nevertheless, prediction tasks are affected by this parameter as it takes more time (epochs) to reach smaller errors, meaning that the hybrid training is used more times if a small number of epochs is used during evolution.

4.3.1 Effect of varying epochs and generations

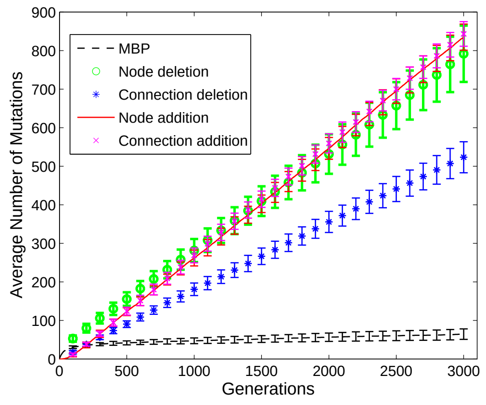
To understand the effect of the number of epochs and generations over the STP, two more experiments for prediction tasks were run with the following parameters: 1) 300 epochs of partial training during 3000 generations and 2) 1500 epochs during 1000 generations. Note that 1500 epochs too are considered here to be partial training, as more epochs are usually needed for full training. Up to 3000 epochs have been used and no overfitting has been found in prediction tasks. Both experiments used the same algorithm with equal values in the rest of the parameters; thus, only the numbers of epochs and generations were modified. As a representative example, it was chosen the $Lo_A^{(ssp)}$ TS. The average error obtained after 30 independent trials showed a NRMSE = 0.029 ±0.021 for the first experiment with 300 epochs and a NRMSE = 5.00E-04 ±2.46E-04 for the second experiment. That clearly shows that a bigger number of epochs can obtain smaller results (statistically significant) with fewer generations than using more generations and fewer epochs.

4.3.2 Effect of varying the learning algorithms

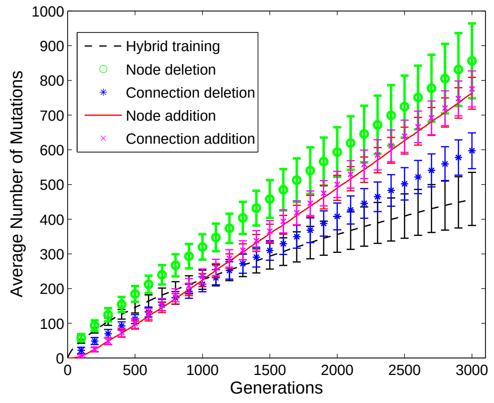
Another scenario was tested where just the MBP was used as a learning algorithm and compared with the standard implementation (using the hybrid training) during 3000 generations of evolution. The reason is because previously a close relationship between the learning of the networks and the successful training parameter have been shown; thus, using just the MBP or the combined training may have a different effect on the number of mutations used as well the error obtained.



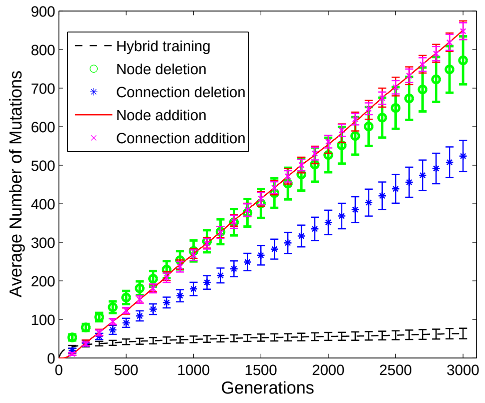
(a)



(b)



(c)



(d)

Figure 4.6: Average number of mutations for $Lo_{B2}^{(msp)}$ TS with the STP and SA. *Successful Training* parameter set to 30% without SA (Fig. 4.6a), 70% without SA (Fig. 4.6b), 30% with SA (Fig. 4.6c) and 70% with SA (Fig. 4.6d)

Fig. 4.6 shows the average number of mutation over 30 independent runs for the $Lo_{B2}^{(msp)}$ using the MBP with a STP = 30% in Fig. 4.6a and a STP = 70% in Fig. 4.6b. The same scenario is repeated using hybrid training with STP = 30% in Fig. 4.6c and with STP = 70% in Fig. 4.6d.

As can be appreciated, if a strict value for the STP is again used, the learning algorithms (MBP or MBP with SA) are used fewer times (Fig. 4.6b and 4.6d) than if a relaxed value is used (Figs. 4.6a and Fig. 4.6c). On the contrary, and as expected, the use of the hybrid algorithm allows a bigger reduction in the training error producing to be used more times the hybrid training than in the case of the single MBP with a relaxed value in the STP (Figs. 4.6a and 4.6c) where the hybrid training is used more than twice as often as the MBP, presenting a difference that is highly statistically significant.

4.3.3 Successful training parameter discussion

Summarizing the importance of the successful training parameter, it can be said that any value chosen does not affect the evolution of the network if a small number of training epochs are used, because it is quite probable that the error is not reduced significantly during training. But if many epochs of training are required, a strict value in the STP may produce better networks as the search space will be explored in more depth. This is because architectural mutations have more probability of being used during evolution and consequently the EA will look for an optimal solution across a bigger search space.

It may be important to remark that the generalization error is not severely affected if a relaxation of the STP is adopted when several generations of evolution are allowed. However, as seen before on Fig. 4.5a, a relaxed value may consume almost all generations of evolution just training the networks, which clearly will provide a degraded performance in comparison to another implementation that uses more times the architectural mutations than the hybrid training. Another fact to remark is that the hybrid training provides two stages to reduce the error (one for MBP and other for SA), thus combined with a strict value in the STP it may be possible to have fittest networks than if just the MBP is used with a relaxed value in the STP as shown in Fig. 4.7 for the $MG17_A$ on the validation set after 3000 generation of evolution. There is appreciated that, as the STP value is increased, the error is reduced and a slightly improvement in the error may be noted between two consecutive values in the STP when the SA is used; but comparing a STP = 30% and 70% the reduction on the average error is more evident. Note that the generalization error (on the test set) is not presented as there is not involved the SA algorithm (step 9 on the EPNet algorithm, Section 2.2.1.1).

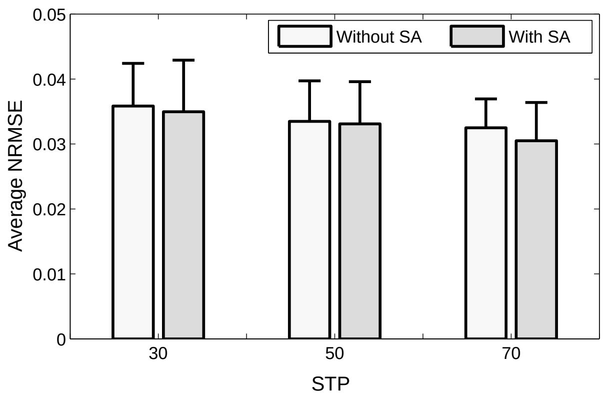


Figure 4.7: Average error on the validation set for the $MG17_A$ varying the learning algorithm and the *Successful training* parameter at generation 3000

For the rest of the work a value of 70% will be used in the STP to perform the evolution of ANNs and MNNs. Other results supporting these experiments over different TS can be found in [67].

4.4 Evolution of features and architectures

For some TS there is enough previous domain knowledge to know many of the different optimal parameters needed to predict, e.g. for the *Mackey-Glass* and *Logistic* TS [35, 65, 77, 145] or for the *Lorenz* TS [129]. But even then, there are other studies that differ from the standard parameters, such as [142] for the *Mackey-Glass* TS. In this field, we are interested in the optimal number of inputs and delays to evolve the network's architectures. However, it is not always possible to obtain this information from the literature, e.g. to predict a new TS that has never been studied before and for which there is no previous information (as is likely to be the case for many real-world scenarios). For the classification field, it is understandable that not all inputs may be useful to classify the input pattern. We should remember that many of them have been selected from previous domain knowledge of human experts (like the *Breast cancer* data set), but given the ANNs capabilities to solve this kind of problem and the optimization advantages provided by EA, it may be clearly expected that the input feature selection will help in the correct identification of the optimal number of inputs to solve the task. In this case, it was decided to not use an *Evaluation functions* (Section 2.2.1.4) method to determine appropriate inputs, because this procedure adds an extra stage in the solution of the task, i.e. first an appropriate number of inputs has to be determined, and only then is the main procedure applied to solve the problem [130, 163–165]. This approach has previously been applied to forecasting [163, 164] and to classification [130, 165].

Consequently, the approach adopted here belongs to the *Generational* procedures, because it evolves the ANNs' inputs with an EA [63, 76, 146] to allow the automatic adaptation of the inputs at the same time as the rest of the architecture evolves.

For the remaining of this chapter, Evaluation functions are used to fix the inputs during evolution as in the classical EPNet algorithm, while a Generational procedure (evolution) will be adopted to look for better performances with a small input space. Two different algorithms are implemented to test the evolution of features selection in the predictions problem – FS-EPNet (Section 4.4.2.1) and FS$_{AD}$-EPNet (Section 4.4.2.2) algorithms. The latter can be used in the feature selection for classification tasks.

As we are concerned with the evolution of features for prediction tasks (inputs and delays), the algorithms presented next will have mutation operators to add/delete inputs/delays, with the exception of the algorithm presented in Section 4.4.2.2 where the delay mutation operators (addition and deletion) are not presented. Thus, for this case a series of experiments was performed to determine the sensitivity of the evolution of inputs in the EPNet algorithm, i.e. it was explored whether it is better to evolve the inputs. Since evolving the inputs means that there are more parameters to evolve, this could potentially compromise the performance of the algorithm, and it was important to test whether that does happen and if so, when.

Next are presented three algorithms that will be used to demonstrate if the feature selection can be implemented with just mutation operators in the EPNet algorithm: 1) the fixed input case which uses the standard EPNet algorithm; 2) the FS-EPNet algorithm that evolves the features from scratch using symmetric delays between inputs and 3) the FS$_{AD}$-EPNet that only uses a mutation operator to evolve inputs, thus the evolved inputs may have asymmetric delays between them. The FS-EPNet algorithm was mainly designed for prediction tasks, whereas the FS$_{AD}$-EPNet algorithm can be applied to both, prediction and classification tasks. Thus, the algorithm explanation of each one of them will be shown next and the results from these algorithms will be presented in Section 4.5.

4.4.1 Fixed Inputs

For the fixed input case, a *Evaluation function* procedure is needed to determine an appropriate fixed number of inputs and associated delays before the evolutionary process begins for prediction tasks. As remarked in Section 3.2, the data set for classification tasks is already organized into input and target vectors and thus, the evolution of the input features will be carried out by activating and deactivating a set of inputs. Here the Takens embedding theorem was employed, which has been used before for the TS prediction task [137] and stated in Section 3.1.2. With this, the number of inputs is chosen with the False Nearest Neighbour method, and the delays between them are fixed using the Average Mutual Information. Both techniques were obtained and applied from the Visual Recurrent Analysis package [136]. Other studies have used these methods, such as [129, 143], but these did not evolve the ANNs and instead used fixed architectures to predict the *Lorenz* TS.

There was an issue calculating the inputs and delays with the Visual Recurrent Analysis package. The problem arises when different number of samples are used to obtain the number of inputs and delays. For example, the $Lo_A^{(ssp)}$ TS with 10000 values produced 2 inputs and 16 delays after applying the algorithms from the package, but if 2500 values are considered in the calculation of them, 10 inputs and 15 delays are obtained. That may be a factor of concern when decided how much information is used to train, validate and test the predictions with ANNs. The number of inputs and delays for each TS used in this chapter is presented in Section 4.5.1.

4.4.2 Input feature selection in the EPNet

As explained in Section 4.4, the *Evaluation function* approach was not chosen as a preferred method for this study, because the selection of inputs is then a separate process from the evolution of the ANNs, and because the number of inputs and delays may change if different amount of information is required. Rather, it is natural that everything should be evolved at the same time, allowing the inputs to adapt as required. Even more, it may be possible to obtain better (or the same) performance values with fewer number of inputs, but that needs to be tested empirically.

To avoid introducing complex and unnecessary new operators for evolving the inputs, they were evolved inside the EPNet using the same operators already developed for the original algorithm, i.e. the inputs were simply treated in the same way as any other node in the ANN. There is a series of potential input nodes corresponding to the TS at points in the past, i.e. $\{x_t, x_{t-1}, x_{t-2}, ...\}$, and the add/delete node mutation process selects which subsets are actually used in the network. Similarly, the connections from inputs to hidden nodes were treated as another connection between hidden nodes, so the mutation add-or-delete connection could be applied to them.

That provides a way to evolve the inputs in this kind of algorithms (EP) but as the delay is another factor, the number of possible combinations to evolve inputs and delays are increased. Thus, we are focusing here on just two algorithms in order to evolve them: the FS-EPNet algorithm presented in the next section which evolves inputs and delays (symmetric delays between inputs), and the FS$_{AD}$-EPNet algorithm in Section 4.4.2.2 where only input nodes are evolved with a mutation operator. Unlike in the fixed inputs case (Section 4.4.1) and the FS-EPNet algorithm, the n chosen inputs in the FS$_{AD}$-EPNet algorithm will not necessarily be separated by some fixed time delay d, i.e. the delays are implicit in the representation and asymmetric delays between two inputs may appear.

The adaptation of the delays through evolution could be seen as the adaptive time delay carried out by [152] with a mathematical method to adjust them in a non-evolutionary approach. There [114] calls the model presented in [152] as a MNN to predict chaotic TS (*Mackey-Glass* TS) but [152] uses a mathematical method to adapt the delays and it does not use the same module definition as used in this thesis. As far as this research has been carried out, no work has been found that uses MNNs to predict TS.

4.4.2.1 Feature Selection EPNet (FS-EPNet) algorithm

The first modified version to evolve the features is called FS-EPNet algorithm (Fig. 4.8) which introduces the addition/deletion of inputs and delays. If such operators are not considered, then the original mutations of the EPNet algorithm are obtained (Fig. 2.2b). Note that the general algorithm for the FS-EPNet is the same as 2.2a.

Whether it is an input or a delay that is added, the data sets (input and target values to train and test the networks) change and it needs to be readjusted to the new parameters. The input mutations, add or delete a certain number of input features (1-2 inputs, see Section 4.1.2) from the input vector, whereas the delay mutations change the delays between two consecutive inputs to the new value, e.g. if all the inputs have a delay of 2 between each of them and the delays are incremented, the new data set will have the same number of inputs with a delay of 3 between each of them. That will result into symmetric delays between inputs with the FS-EPNet algorithm. As can be seen in Fig. 4.8, the input and delay deletion are introduced at the end of all the previous deletions, which means that they have less probability of being applied, e.g. if a hidden node is deleted, the rest of the deletion and addition mutations will not be performed over the selected network. These new mutations were introduced in this position (i.e. of having a smaller probability of being used) because the search space of hidden nodes (and connections) is usually bigger than the search space of features, even if the number of inputs can be increased during evolution, i.e. the number of required hidden nodes is usually several orders of magnitude bigger than the input space; nevertheless, this depends on the problem at hand. Thus, the node and connection deletion will have a greater probability of reducing the network size before the

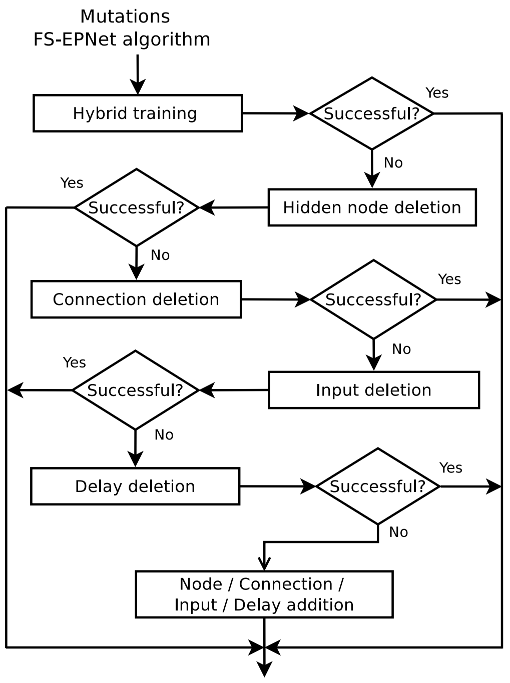


Figure 4.8: Mutations for the FS-EPNet algorithm

focus is directed to the reduction of the dimensionality of the input space. Similar to the EPNet mutation additions, the FS-EPNet algorithm has a tournament stage between all the children in the last mutation attempted, but here there are two new mutation operators: the addition of inputs and delays.

4.4.2.2 Feature Selection with asymmetric delays EPNet (FS$_{AD}$-EPNet) algorithm

The FS-EPNet algorithm evolved symmetric delays between inputs, incrementing or decrementing the number of delays. Nevertheless, if we restrict the research to the symmetric case we may be losing potential new algorithms. For that reason it will be investigated the special case in the evolution of features when the delays between inputs may not be symmetric. For example, in a symmetric case (FS-EPNet algorithm), if it is assumed that there exists a delay of 2 in an input vector of 3 values, there will be an input vector formed as x_t, x_{t-2}, x_{t-4} to make a prediction x_{t+1} assuming $\Delta t = 1$. In the FS$_{AD}$-EPNet algorithm (Fig. 4.9), it is assumed that there exists a delay of 1 between 2 consecutive inputs at the

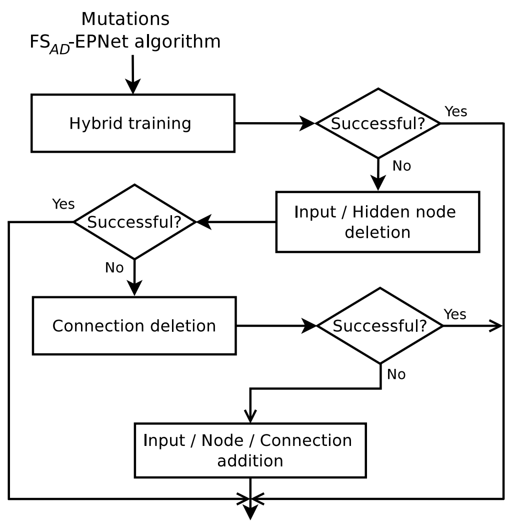


Figure 4.9: Mutations for the FS$_{AD}$-EPNet algorithm

beginning of the evolution. The asymmetric delays appear when input nodes are deleted (incrementing the delays between the adjacent inputs around the node deleted). Thus, if we consider the next input vector: $x_t, x_{t-1}, x_{t-2}, x_{t-3}$, if input x_{t-2} is deleted, inputs x_t and x_{t-1} have a delay of 1, meanwhile inputs x_{t-1} and x_{t-3} have a delay of 2. In Fig. 4.9, the input and hidden node deletion are at the same level, which means that one of them is randomly selected.

Several different kinds of combination could be used to rearrange the mutation operators to evolve the input feature in the FS-EPNet and FS$_{AD}$-EPNet algorithms, but for the purpose of this study, those algorithms may be enough to investigate this aspect in the networks. Further improvements may present a more sophisticated combination of them.

4.5 Feature evolution comparison

This section presents some experimental results that are focused on determining whether it is better to evolve the inputs as another parameter inside the EPNet algorithm. To do this, four experiments were set up for the prediction and two for the classification problems.

Prediction problems: 1) the inputs and delays were fixed (Evaluation function methods) as mentioned in Section 4.4.1 and the EPNet algorithm only evolve the hidden nodes and connections; 2) the features were fixed as experiment one, thereafter evolved with the FS-EPNet algorithm; 3) the inputs and delays where evolved from random initial values with the FS-EPNet algorithm and 4) just the inputs were evolved with the FS$_{AD}$-EPNet algorithm. The second experiment (with fixed and evolved features) was set-up in that way to make the evolution of inputs easier by starting at the computed values so that evolution can improve on them where possible, but hopefully not end up with inferior values. Also experiment one and two will serve to question whether the False nearest neighbours and Average mutual information from the Visual Recurrent Analysis package [136] are good enough to provide the best predictions.

Classification problems: 1) the standard EPNet algorithm is used to evolve just the architectures, where the initial number of inputs are determined by the problem at hand and 2) the FS$_{AD}$-EPNet algorithm is used to evolve the inputs, thus, this will be the only method tested for feature evolution for classification tasks.

Those results will be taken as baseline performance for the evolution of MNNs in Chapter 5. To clarify the results of this section, prediction and classification results will be presented in separate stages.

4.5.1 Prediction tasks

Tables 4.1 - 4.4 summarize the results of the different algorithms showing the NRMSE and network sizes for the prediction tasks. Table 4.1 presents the EPNet algorithm with fixed inputs and delays, Table 4.2 shows the EPNet with Fixed and evolved features, the FS-EPNet implementation is presented in Table 4.3 and finally the asymmetric evolution of inputs is shown in Table 4.4. On all these tables, columns 2-5 give the mean, standard deviation, minimum and maximum values of the NRMSE (evaluated on the previously unseen test set, i.e. the test set used after the evolution has finished) for the best individuals from the 30 runs. Columns 6-9 give the number of inputs, delays (with the exception of Table 4.4), hidden nodes and connections of the best individuals overall from the 30 runs,

Table 4.1: Individual TS results for the EPNet with fixed features. Evolved NRMSE and architecture parameters for the best individual values over 30 runs with 3000 generations

Time Series	*NRMSE*				*Best individual*			
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Delays*	*Hidden*	*Connections*
Logistic	0.005393	0.002775	0.001210	0.011550	1	12	6	18
$Lo_{A}^{(ssp)}$	0.027544	0.015870	0.012344	0.080799	10	15	23	106
$Lo_{B1}^{(ssp)}$	0.035553	0.005317	0.027962	0.044020	2	3	31	153
$Lo_{B2}^{(msp)}$	1.034266	0.052348	0.951246	1.126090	2	3	12	47
$Lo_{B3}^{(msp)}$	1.005136	0.019211	0.971249	1.066980	2	3	13	60
$MG17$	1.050964	0.215891	0.555976	1.712120	3	10	5	22
$MG17_A$	0.290963	0.346790	0.126737	2.070020	3	10	10	68
$MG30$	0.100739	0.066835	0.039385	0.285613	6	14	23	196

corresponding to the networks with the NRMSE in column Min. Table 4.4 does not present the delays found with the evolutionary algorithm because many complex variations are possible (explained in Section 4.4.2.2).

Columns 6 and 7 in Table 4.1 were obtained from the Visual Recurrent Analysis package as previously stated, where the embedded dimension (D) was used as input vector to the networks. That was done because using $2D + 1$ inputs did not improve the results on all TS used, only for the $Lo_{B2}^{(msp)}$ there was shown an smaller error than the presented in Table 4.1, but it was not better than the obtained in Table 4.4 with the FS$_{AD}$-EPNet algorithm. Hence, it was used the embedded dimension to set-up the number of inputs in all TS in Tables 4.1 and 4.2.

Detailed average values for the rest of the parameters (inputs, delays, hidden nodes, connections, validation error and test set error) may be found in Appendix A. Moreover, the statistical analysis, presented ahead, was made considering the average values from Appendix A for the NRMSE, inputs, hidden nodes and connections (or the average values from Tables 4.1 - 4.4 for the NRMSE).

As a general overview, the EPNet algorithm obtained the smallest error values for the $Lo_{B3}^{(msp)}$ TS, the FS-EPNet algorithm was better on $Lo_{A}^{(ssp)}$, $Lo_{B1}^{(ssp)}$, $MG17$, $MG17_A$ and

Table 4.2: Individual TS results for the FS-EPNet with fixed and evolved features. Evolved NRMSE and architecture parameters for the best individual values over 30 runs with 3000 generations

Time Series	*NRMSE*				*Best individual*			
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Delays*	*Hidden*	*Connections*
Logistic	0.005081	0.002339	0.001528	0.008653	1	12	7	20
$Lo_A^{(ssp)}$	0.001190	0.000998	0.000049	0.003161	6	1	42	384
$Lo_{B1}^{(ssp)}$	0.004593	0.001917	0.001856	0.006593	4	1	20	147
$Lo_{B2}^{(msp)}$	0.664276	0.309823	0.072599	1.109790	3	3	20	132
$Lo_{B3}^{(msp)}$	1.005700	0.018933	0.965653	1.079210	2	5	10	48
$MG17$	0.274505	0.149549	0.142961	0.667333	5	12	22	237
$MG17_A$	0.051479	0.029031	0.017289	0.134359	4	12	17	178
$MG30$	0.131988	0.086060	0.036435	0.322622	6	14	17	141

Table 4.3: Individual TS results for the FS-EPNet evolving features from scratch. Evolved NRMSE and architecture parameters for the best individual values over 30 runs with 3000 generations

Time Series	*NRMSE*				*Best individual*			
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Delays*	*Hidden*	*Connections*
Logistic	0.002723	0.002599	0.000173	0.007720	1	4	12	56
$Lo_A^{(ssp)}$	0.000990	0.000922	0.000248	0.003206	5	2	27	247
$Lo_{B1}^{(ssp)}$	0.003686	0.003132	0.001025	0.015032	8	1	36	420
$Lo_{B2}^{(msp)}$	0.523017	0.261622	0.098344	0.926392	6	3	12	91
$Lo_{B3}^{(msp)}$	1.032600	0.030747	0.980117	1.099000	8	4	12	69
$MG17$	0.168096	0.046265	0.073971	0.309649	6	3	23	158
$MG17_A$	0.027261	0.026994	0.007204	0.117616	10	3	26	359
$MG30$	0.090790	0.041860	0.018697	0.158511	11	3	19	167

$MG30$, meanwhile the FS$_{AD}$-EPNet algorithm obtained lower error on the remaining TS (*Logistic* and $Lo_{B2}^{(msp)}$).

To facilitate the comparison and analysis of statistical significance with the T-test method, Table 4.5 shows the *p-values* found of comparing all the algorithms for all TS over the NRMSE, inputs, hidden nodes and connections. In this way, it was taken the smallest value per TS (e.g. error) and that experiment was used to calculate the *p-values*

Table 4.4: Individual TS results for the FS$_{AD}$-EPNet evolving inputs features. Evolved NRMSE and architecture parameters for the best individual values over 30 runs with 3000 generations

Time Series	*NRMSE*				*Best individual*		
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Hidden*	*Connections*
Logistic	0.000364	0.000455	0.000022	0.002024	1	25	102
$Lo_{A}^{(ssp)}$	0.001235	0.000914	0.000239	0.003303	5	34	420
$Lo_{B1}^{(ssp)}$	0.004279	0.002777	0.001288	0.013683	7	25	263
$Lo_{B2}^{(msp)}$	0.472015	0.328557	0.049027	1.048780	8	8	47
$Lo_{B3}^{(msp)}$	1.020563	0.025913	0.976759	1.108650	4	5	41
$MG17$	0.249989	0.367239	0.032700	1.681180	7	31	297
$MG17_A$	0.041665	0.076593	0.006173	0.315323	7	11	96
$MG30$	0.105963	0.071941	0.022402	0.318639	4	19	152

against all other experiments, therefore a value of “ * ” is presented in Table in 4.5 for the best error found (NRMSE) for each TS (rows) with a specific algorithm (columns), and the same is repeated for the inputs, hidden nodes and connections.

Thus looking at statistical significance on Table 4.5 for the NRMSE, it can be seen that the error of $Lo_{B3}^{(msp)}$ TS with the EPNet was highly significant better than the FS-EPNet algorithm, it was almost significant against the FS$_{AD}$-EPNet algorithm, but no significance was found if compared with the FS$_{AD}$-EPNet fix and evolved features.

For the cases in which the FS-EPNet obtain smaller results, there was no statistical significance at all if compared against the FS$_{AD}$-EPNet algorithm in all cases, nor for $MG30$ for the other two algorithms. However it was statistical significant better on $MG17$ and $MG17_A$ against the EPNet and the FS-EPNet fixed and evolve.

Interestingly, if the EPNet and the FS-EPNet fixed and evolve algorithms are only compared, the latter improved with high statistical significance for all TS against the EPNet with the exception of *Logistic* and $Lo_{B3}^{(msp)}$ where there was not significance at all, providing insights that evolving the calculated values may be beneficial than if the fixed values are used during all evolution. The results presented from Table 4.1 to 4.4 show that, overall, evolving the inputs with the EPNet algorithms (FS-EPNet and FS$_{AD}$-EPNet) leads to

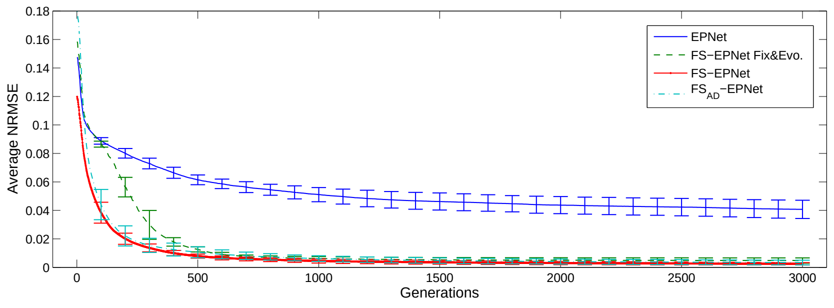
Table 4.5: *p-values* between the four algorithms tested for prediction tasks over the NRMSE, inputs, hidden nodes and connections. A mark like "*" indicate the best error found for a given TS (rows) with a selected algorithm (column)

	Time Series	EPNet	FS-EPNet *Fix&Evo.*	FS-EPNet	FS$_{AD}$-EPNet
NRMSE	*Logistic*	6.11E-11	4.17E-12	2.93E-05	*
	$Lo_A^{(ssp)}$	4.48E-10	0.423386	*	0.304104
	$Lo_{B1}^{(ssp)}$	3.80E-31	0.183278	*	0.441105
	$Lo_{B2}^{(msp)}$	2.31E-10	0.023295	0.509149	*
	$Lo_{B3}^{(msp)}$	*	0.909156	1.34E-04	0.011415
	$MG17$	1.09E-20	7.00E-04	*	0.235176
	$MG17_A$	2.59E-04	1.45E-03	*	0.338165
	$MG30$	0.493096	0.023134	*	0.323549
Inputs	*Logistic*	*	1.08E-30	9.97E-04	3.26E-01
	$Lo_A^{(ssp)}$	2.41E-19	7.08E-01	1.41E-01	*
	$Lo_{B1}^{(ssp)}$	*	1.82E-11	1.03E-15	7.72E-14
	$Lo_{B2}^{(msp)}$	*	1.29E-13	5.44E-14	1.25E-12
	$Lo_{B3}^{(msp)}$	*	2.01E-04	5.11E-19	7.49E-09
	$MG17$	*	1.26E-06	6.24E-13	1.25E-15
	$MG17_A$	*	1.61E-05	7.17E-16	1.70E-12
	$MG17_A$	1.61E-01	*	1.76E-22	6.04E-02
Hidden Nodes	*Logistic*	4.14E-01	*	2.02E-04	6.54E-14
	$Lo_A^{(ssp)}$	*	2.56E-01	1.26E-01	3.53E-01
	$Lo_{B1}^{(ssp)}$	5.71E-01	*	1.54E-06	2.41E-05
	$Lo_{B2}^{(msp)}$	4.94E-02	6.42E-01	5.63E-01	*
	$Lo_{B3}^{(msp)}$	2.70E-01	*	7.84E-01	1.91E-01
	$MG17$	*	4.03E-05	1.27E-06	3.56E-08
	$MG17_A$	*	2.47E-05	6.94E-07	8.43E-09
	$MG17_A$	6.91E-05	7.45E-04	1.81E-01	*
Connections	*Logistic*	1.61E-01	*	1.24E-03	2.21E-10
	$Lo_A^{(ssp)}$	9.58E-01	*	2.19E-01	4.17E-01
	$Lo_{B1}^{(ssp)}$	*	2.19E-01	3.28E-11	2.48E-09
	$Lo_{B2}^{(msp)}$	*	6.13E-01	4.53E-02	2.00E-01
	$Lo_{B3}^{(msp)}$	*	6.86E-01	5.04E-05	4.72E-03
	$MG17$	*	1.67E-07	3.71E-08	4.12E-08
	$MG17_A$	*	3.66E-06	3.87E-10	4.46E-10
	$MG17_A$	6.38E-05	5.28E-04	7.47E-02	*

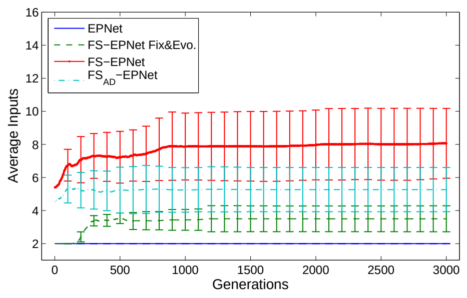
better TS prediction (i.e. lower NRMSE). In general the best algorithm that solves a TS did not find the smallest architecture, considering the number of trainable connection as the networks' size (*p-values* from Table 4.5). On the other hand, the average and best inputs and delays obtained are mixed in some cases, while in others the parameters converge to the same between algorithms, thus, one needs to look more carefully at what emerges for the individual TS. For example, the EPNet algorithm found the smallest architectures from all experiments (Table 4.5 for inputs, hidden nodes and connections), being highly statistically better than the other algorithms, but only the $Lo_{B3}^{(msp)}$ TS found the smallest average error with the EPNet, therefore, the smallest network found were not the best in all cases.

Even though there are more parameters to evolve if the inputs are not fixed, the algorithm can still obtain faster results in some cases. For example, consider Fig. 4.10 presenting the evolution of NRMSE, inputs, delays, hidden nodes and connections with the four algorithms tested in this chapter for the $Lo_{B1}^{(ssp)}$ TS. It can be seen that the evolution of inputs allows the algorithm to obtain smaller errors faster (particularly at the beginning of the evolution) as presented in Fig. 4.10a. Conversely, there were other cases where the error was reduced equally fast in any scenario, for example the *Logistic* TS (not presented here).

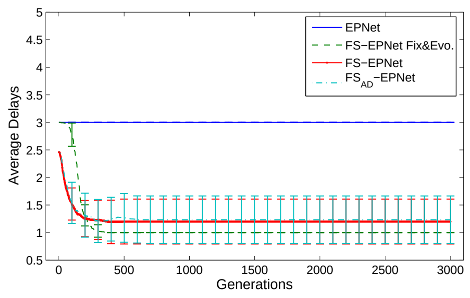
Considering the evolution of delays on Fig. 4.10c, all algorithms converge to similar values in the first 500 generations of evolution, making clear that the calculated delays were not the best, nor to maintain them during evolution (EPNet algorithm on Fig. 4.10c). For the inputs, the three algorithms that evolve them obtain bigger values during evolution than the calculated values for the EPNet algorithm. If we consider the inputs (Fig. 4.10b), hidden nodes (Fig. 4.10d) and connections (Fig. 4.10e) we may seen that the error bars present a big deviation (as previously remarked on Section 4.1). That is a clear behaviour of the algorithm, as the mutations that add neural components increase the size of the parent network, regarding whether those mutations improve the fitness of the last individual replaced. In this way, it assures that the diversity is maintained, but on the other hand the last individual in the population will be bigger every generation if it is not replaced, e.g. if bigger architectures produce better fitness (which may not be true for all cases) they will advance in the ranked population to better position, and if bigger networks continue arriving while the best individuals are of smaller sizes, that will produce to have bigger



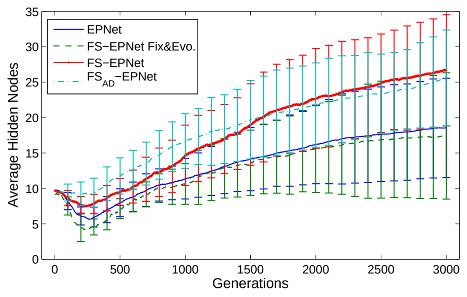
(a)



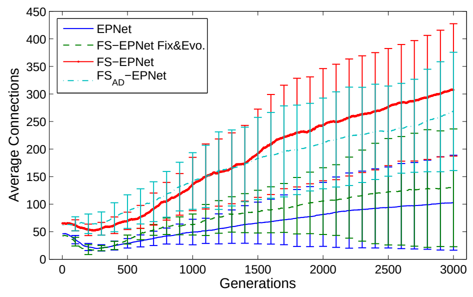
(b)



(c)



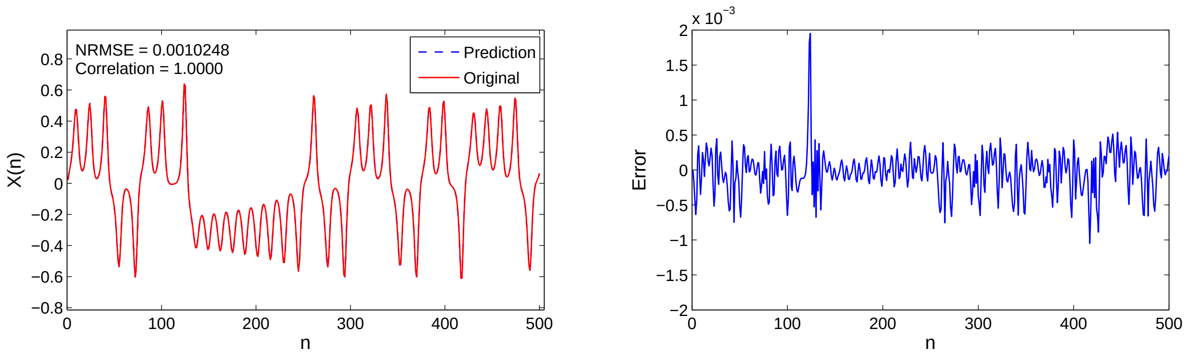
(d)



(e)

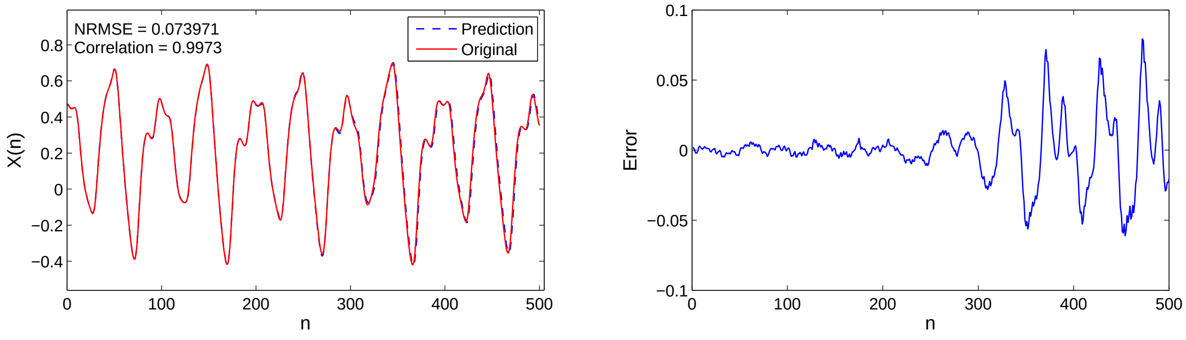
Figure 4.10: Average values per generation of different parameters evolved for the $Lo_{B1}^{(ssp)}$ TS with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms. NRMSE (Fig. 4.10a), inputs (Fig. 4.10b), delays (Fig. 4.10c), hidden nodes (Fig. 4.10d) and connections (Fig. 4.10e) evolved during 3000 generations

deviations in the hidden nodes and connections. Note that for the input case, the standard deviation is not increased as in the hidden nodes and connections, which indicate that the algorithm does not need more inputs every generation to improve the performance, corroborating that the input space is smaller (as commented in Section 4.4.2.1) than the search space formed by hidden nodes and/or connections. The best prediction found for $Lo_{B1}^{(ssp)}$ over all independent runs and over all four algorithms tested is presented in Fig.



(a) (b)

Figure 4.11: Best prediction found for $Lo_{B1}^{(ssp)}$ TS with the FS-EPNet algorithm after 3000 generations of evolution. Fig. 4.11a shows the prediction on the final test set and Fig. 4.11b presents the error in terms of $Y_i(t) - Z_i(t)$



(a) (b)

Figure 4.12: Best prediction found for $MG17$ TS with the FS-EPNet algorithm after 3000 generations of evolution. Fig. 4.12a shows the prediction on the final test set and Fig. 4.12b presents the error in terms of $Y_i(t) - Z_i(t)$

4.11 obtained with the FS-EPNet algorithm. Fig. 4.11a shows the prediction on the final test set and Fig. 4.11b presents the error in terms of $Y_i(t) - Z_i(t)$. It has also presented the correlation value to have an appreciation of the prediction error from another angle. As Fig. 4.11, the best prediction found for the $MG17$ with the FS-EPNet algorithm is displayed in Fig. 4.12.

As remarked previously, it may be important to analyze the average number of generations required until the error is no further reduced considerably. Therefore, it was repeated for all previous experiments with the condition to stop the algorithms if the best individ-

Table 4.6: Average number of generations per TS with stopping criteria: stop if the error of the best individual is not improved after 300 generations

Time Series	EPNet	FS-EPNet *Fix&Evo.*	FS-EPNet	FS$_{AD}$-EPNet
Logistic	1561±553.73	1431±502.51	2424.16±1418.20	3713.96±1254.32
$Lo_A^{(ssp)}$	2640.93±1058.99	2650.93±1000.78	2051±595.23	2657.56±1057.96
$Lo_{B1}^{(ssp)}$	2251±931.35	2774.26±1062.36	2771±828.85	3480.80±996.02
$Lo_{B2}^{(msp)}$	1531±596.62	1461±653.16	1411±631.00	1391±469.29
$Lo_{B3}^{(msp)}$	1451±557.55	1591±739.68	2021±884.11	1541±436.75
$MG17$	1331±538.29	2141±821.52	2341±920.11	2371±978.79
$MG17_A$	1411±480.19	2261±750.44	2231±750.24	2337.63±959.60
$MG30$	2157.63±939.33	2061±883.41	2267.63±1151.32	2141±720.91

ual cannot further reduce its error. The error presented in this new experiment were not statistically significant against previous results because if more generations were allowed, the predictions continue reducing the error, but not by a big amount. Table 4.6 presents the average number of generations for all TS over all algorithms used in this section with their respective standard deviation. It can be seen that in the majority of the cases, it was required on average less than 3000 generations (as done in previous experiments), while for *Logistic* and $Lo_{B1}^{(ssp)}$ with the FS$_{AD}$-EPNet algorithm it was required more than that.

4.5.2 Classification tasks

Tables 4.7 and 4.8 present the classification error for the EPNet and FS$_{AD}$-EPNet respectively for the *Breast cancer, Optical digit* and *Thyroid* data sets when the algorithms are allowed to stop as said in Section 4.1.2. For the *Breast cancer* both algorithms found similar average errors, but for the *Optical digit* set the EPNet algorithm with the fixed inputs found the smallest average error on the classification error and in the error percentage (the latter not presented in this section but can be found in Appendix A). For the *Thyroid* data set the evolution of features with the asymmetric delay implementation provided the best performance measures against the normal EPNet algorithm. Moreover, *Thyroid* data set used on average a small number of inputs (14.831±3.20) while the best individual found

Table 4.7: Classification error for the EPNet algorithm with fixed input features and stopping criteria. Evolved error rates and architecture parameters for the best individual values over 30 runs

Time Series	*Test error rate* (%)				*Best individual*		
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Hidden*	*Connections*
Breast C.	0.438096	0.466973	0	1.714290	9	3	34
Optical digit	5.325300	0.715687	4.229270	7.790760	64	25	1499
Thyroid	1.575262	0.159964	1.312720	1.925320	21	18	329

Table 4.8: Classification error for the FS$_{AD}$-EPNet algorithm with evolved input features and stopping criteria. Evolved error rates and architecture parameters for the best individual values over 30 runs

Time Series	*Test error rate* (%)				*Best individual*		
	Mean	*Std Dev*	*Min*	*Max*	*Inputs*	*Hidden*	*Connections*
Breast C.	0.438096	0.288004	0	1.142860	9	4	44
Optical digit	6.124242	0.674056	4.785750	7.234280	61	20	1309
Thyroid	1.448853	0.167548	0.933489	1.721120	9	20	409

only required 9 inputs (Table 4.8) in comparison with the 21 features the problem has (Table 4.7) as input vector.

The FS$_{AD}$-EPNet algorithm started with random initialization of inputs, which clearly did not produce the best results for the *Optical digits* data set, but for the *Breast cancer* set it was found that all inputs were used in the classification of the best individual found. It was repeated the same experiments activating all inputs before the evolution starts with the FS$_{AD}$-EPNet algorithm and then same results were obtained, indicating that for the *Optical digits* data set, the evolution of inputs were not beneficial.

Comparing statistical significance from the average classification error (Tables 4.7 and 4.8) with the T-test, for the *Breast cancer* data set there was no statistically significant differences between both algorithms ($p\text{-}value = 0.9998$). For the *Optical digit* data set, it was discovered too no statistically significance with a $p\text{-}value = 0.6213$, meanwhile the *Thyroid* data set was statistically significantly better with the FS$_{AD}$-EPNet algorithm than with

Table 4.9: Average number of generations per data set with stopping criteria: stop if the error of the best individual is not improved after 100 generations or if the classification error on the validation set is zero by the best individual

Data Set	EPNet	FS$_{AD}$-EPNet
Breast C.	49.33±39.28	47.56±37.36
Optical digit	1290.86±1003.78	644.13±688.14
Thyroid	1097.66±470.86	1484.33±436.35

the standard EPNet, obtaining a *p-value* = 0.0041. Table 4.9 shows the average number of generations required for each data set with both algorithms. Where the *Breast cancer* data set was the fastest to be solved.

As previously said, it was repeated the same experiments fixing the number of generations to have another perspective in the convergence of parameters over the entire population. There it was found that the *Breast cancer* data set generated a network able to classify all patterns in the test set with fewer number of inputs on average, and smaller number of inputs for the best individual: 4.2±1.57 and 5 inputs respectively. The inputs that were not used in that case are: 3, 4, 5 and 9 which means that just inputs 1, 2, 6, 7 and 8 where enough to correctly classify all patterns in the unseen data set.

However, the small amount of inputs in this data set required to use on average more nodes and connections on the best individual found with the FS$_{AD}$-EPNet algorithm. For the *Thyroid* data set the inputs used to perform the classification for the best network found were: 3, 6, 8, 13, 17, 19 and 21 with 3000 generations. When stopping criteria was used in the FS$_{AD}$-EPNet algorithm, the inputs used for the best individual were: 3, 7, 8, 11, 13, 15, 17, 19, 20. It is worth to said that allowing more generations, lower amount of features could be obtained without compromise the performance of the networks.

As a representative example of the evolution of parameters for classification tasks (not that each one was different), Fig. 4.13 shows the Average classification error, inputs, hidden nodes and connections for the *Optical digit* data set. The error (fitness) during evolution over the validation set was similar between both algorithm (Fig. 4.13a). For the number

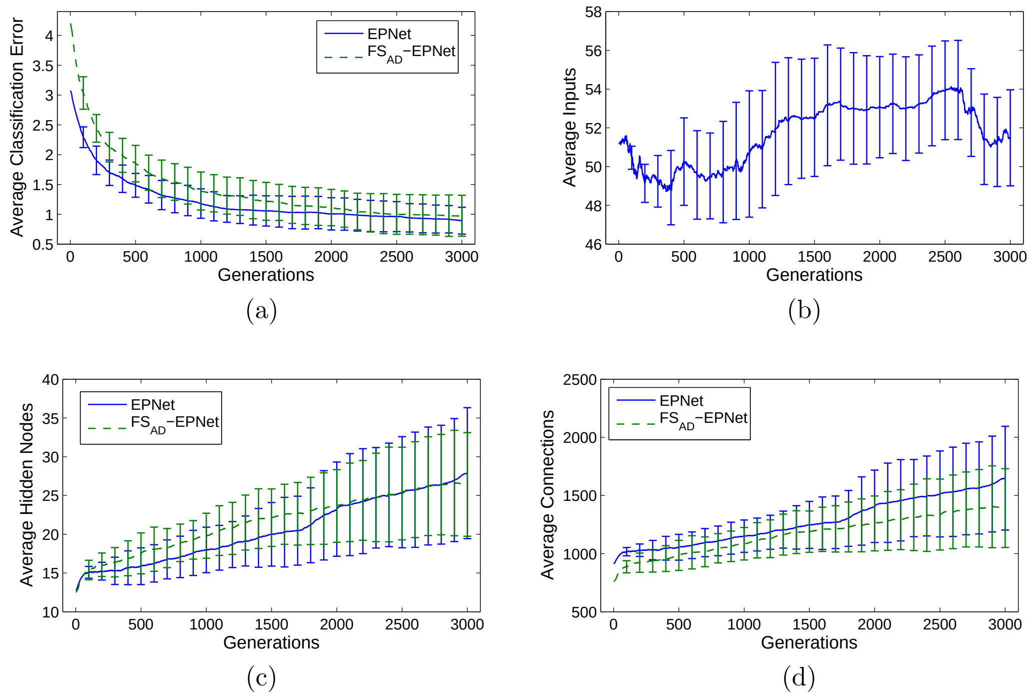


Figure 4.13: Average values per generation of different parameters evolved for the *Optical digit* data set with the EPNet and FS$_{AD}$-EPNet algorithms. Classification error rate (Fig. 4.13a), inputs (Fig. 4.13b, only the FS$_{AD}$-EPNet algorithm applied), hidden nodes (Fig. 4.13c) and connections (Fig. 4.13d) evolved during 3000 generations

of inputs (Fig. 4.13b) it is only shown the results with the FS$_{AD}$-EPNet algorithms as in the other case the inputs were fixed. There can be seen how the inputs oscillated without reaching an optimal parameter that allows better results in comparison with the fixed input case. Nevertheless, both obtain similar errors as said before. The hidden nodes and connections were increased during all evolution presenting bigger deviations as remarked in Fig. 4.10 being a characteristic behaviour of the EPNet algorithm.

4.5.3 Feature evolution discussion

This section studied the issue of feature selection for evolved ANNs. It presented a series of experiments to determine whether it is better to evolve the inputs when the EPNet algorithm is used for TS prediction and classification tasks. This was done by comparing evolving the inputs (3 algorithms) against keeping them at fixed computed values throughout the evolutionary process (standard EPNet algorithm). The fixed inputs were computed using the approaches employed in previous studies, namely False Nearest Neighbour for the number of inputs and Average Mutual Information for their delays for prediction tasks. For classification was used the number of inputs of the problem at hand.

From those experiments it was also observed that if the inputs are fixed, the evolution of hidden nodes or connections advances faster in some TS, consistent with the fact that there are fewer parameters to evolve. On the other hand, if the inputs are evolved, it was found that in some cases the algorithm found accurate networks faster, even though they required more computational processing to evolve the increased number of parameters.

Prediction tasks: it was shown that evolving the inputs gave statistically significant better prediction results for 7 of the 8 TS studied, but in 1 case was it better to keep the inputs fixed. This led to the conclusion that it is certainly not the case that using the values given by the False Nearest Neighbour and Average Mutual Information is the best way to perform input feature selection, i.e. to fix the inputs during the evolutionary process for prediction tasks.

Classification tasks: here it was different the case as in prediction, i.e. prediction required to find the optimal number of inputs and delays. In classification tasks the maximum number of input is given by the problem at hand, thus feature evolution means to look for a smaller set of inputs that can correctly classify the task given the best generalization error over the networks. Here, just 1 data set of 3 presented statistically significant results if the inputs are evolved having the lowest generalization error. However, *Breast cancer* could classify correctly all patterns in the test set with a small number of inputs for the best individual if more generations were allowed, but it was also noted that this network uses more hidden nodes than the best obtained with the fixed input case. On the other hand, both algorithms perform in a similar way for the *Breast cancer* when stopping criteria is used, presenting no statistically significant differences in terms of error or generations used.

As a general behaviour, it was noted that for any task evolved with the EPNet algorithm, the deviation on hidden nodes and connection is bigger as the generations advance. Clearly that may be a disadvantage as probably will not be possible to run the algorithm for hundreds of thousands of generations without control the growth of the last individuals in the population. One possible further modification may be the constraint of nodes and connections (the latter used in generational algorithms in [12]) as simulated for biological neurons over the human brain, however there may be the risk that it will not explore as many possible combinations as leaving those parameter free to evolve.

Since architectural modifications generally produce large changes in the evolved networks' behavior, it was expected that the addition or deletion of inputs and delays could have the same effect on the network's performance. Sometimes the addition or deletion of inputs results in significant variation in the performance of the networks; however, those variations do not usually have a major impact on the evolution, probably because after addition or deletion the network passes on to the partial training phase which could correct any undesirable deviation in the networks' learning. Again, the dynamic of the TS or data set influences the behavior of the algorithm. The most important conclusion found was that it is better to evolve the inputs and delays for prediction tasks with any of the improved versions of the EPNet algorithm. Moreover, if there is no previous domain knowledge of a given TS, the evolution of features may be the best option as the calculation of inputs with the the False Nearest Neighbour and delays with the Average Mutual Information may change if the size of the TS changes. That could be a really difficult problem for many real world scenarios, as in those cases, the number of samples is expected to grow with time.

For classification tasks it was concluded that not in all cases the best number of inputs is evolved, and probably it may be a better idea to use the standard EPNet algorithm if we are not interested in feature selection.

Thus, one can also conclude that the standard EPNet algorithm is not capable of finding the best input features in all cases either, but in the majority it can lead to the best results. Two publications derived from this section corroborate the findings over different

TS to those presented here, showing that the evolution of inputs gave significantly better prediction results in the majority of the TS studied, but in a small number of cases it was better to keep the inputs fixed [67, 131].

4.6 Comparison with previous studies

Previously it was shown the feature evolution compared between four algorithms (3 derived from the same) showing that indeed evolving the inputs of the network may produce better results in the majority of the cases. In this section will be presented the comparisons of the previous experiments against other algorithms found in the literature to determine whether results of the EPNet algorithm and its variations may compete with the results found in the literature. A general description of the methods used in the literature can be found in Section 3.1.4 for TS and in Section 3.2 for classification tasks.

4.6.1 Logistic time series prediction problem

Table 4.10 presents the comparison of different prediction methods for the *Logistic* TS. This shows that the FS$_{AD}$-EPNet algorithm has a smaller average error than the other implementation of the algorithms and previous results in the literature for this TS. It may be important to remark that the classical EPNet algorithm used only 200 generation of evolution for this TS, and here it has been used 3000 generations. Showing that the error could be further reduced if more generations are allowed.

Another fact to notice is that the four algorithm used here (EPNet, FS-EPNet fixed and evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms) found smaller average errors than the three algorithms reported in previous works as presented in Table 4.10. At the end of the evolution, the four algorithms converge to one input unit for the best individual in the population, the same number of inputs used in previous publications [65, 146].

Table 4.10: Comparison of different prediction methods for the *Logistic* TS with NRMSE. A boldface number indicates the smallest error found

Method	*Av. Connections*	*Av. NRMSE*
Classical EPNet [65]	33.03	0.0257
1-15-1 BP network [146]	-	0.0387
ERP [146]	-	0.0566
EPNet Fix	13.73	0.005393
FS-EPNet Fix&Evo.	12.33	0.005080
FS-EPNet	36.56	0.002722
FS_{AD}-EPNet	112.8	**0.00036**

Table 4.11: Comparison of different prediction methods of the $Lo_A^{(ssp)}$ TS for the best NRMSE. A boldface number indicates the smallest error found

Method	*Hidden nodes*	*Connections*	*NRMSE*
LLNF LoLiMoT [140]	20	-	**3.13E-05**
RBF OLS [140]	24	-	3.75E-05
MLP BP [140] BP	38	-	2.27E-04
EPNet Fix	23	106	1.23E-02
FS-EPNet Fix&Evo.	42	384	4.90E-05
FS-EPNet	27	247	2.48E-04
FS_{AD}-EPNet	34	420	2.39E-04

4.6.2 Lorenz time series prediction problem

Table 4.11 presents the results from the $Lo_A^{(ssp)}$ TS obtained with the four different EPNet implementation and compared against different algorithm found in [140]. For this case none of the algorithms of this study could improve the best algorithm presented in [140], however the FS-EPNet algorithm with fixed and evolved inputs improved the RBF OLS and MLP BP from [140] and it stands with a close value against the best result in [140]. Moreover, it may be important to note that a more valid comparison may be in terms of average values from different runs, but as not in all works is presented such information, here we can only compare our results against the available information shown in the literature. Table 4.12 shows the results and comparison for the $Lo_{B1}^{(ssp)}$ TS with the RMSE. Notice

Table 4.12: Comparison of different prediction methods of the $Lo_{B1}^{(ssp)}$ TS for the best RMSE. A boldface number indicates the smallest error found

Method	*Inputs*	*Delays*	*Nodes*	*Cons.*	*RMSE*
PG-RBF [139]	3	3	-	-	0.094
EPNet Fix	2	3	31	153	0.008031
FS-EPNet Fix&Evo.	4	1	20	147	0.000533
FS-EPNet	8	1	36	420	**0.00029**
FS_{AD}-EPNet	7	1	25	263	0.000369

Table 4.13: Comparison of different prediction methods for the $Lo_{B2}^{(msp)}$ ($\Delta t = 1$) and $Lo_{B3}^{(msp)}$ ($\Delta t = 50$) TS for the best NRMSE. A boldface number indicates the smallest error found

Method	$\Delta t = 1$	$\Delta t = 50$
State-space 8th order [143]	0.1822	1.0299
Regularized ANN-ARMA [143]	0.7325	**0.7652**
EPNet Fix	0.951245	0.971248
FS-EPNet Fix&Evo.	0.072598	0.965653
FS-EPNet	0.098343	0.980117
FS_{AD}-EPNet	**0.049026**	0.976758

that the FS-EPNet algorithm found the smaller error, but using more inputs, hidden nodes and connections than other algorithms. Similar to the *Logistic* TS, the four algorithms implemented found a smaller error against [139], but using in general more inputs and fewer delays than [139].

Table 4.13 shows the comparisons using the NRMSE for the $Lo_{B2}^{(msp)}$ TS in column one and for the $Lo_{B3}^{(msp)}$ TS in column two. In Table 4.13 the FS-EPNet (both) and FS$_{AD}$-EPNet algorithms found smaller errors against previous results for $Lo_{B2}^{(msp)}$ with a $\Delta t = 1$. When the prediction step was increased, it was not possible to improve the results of [143], whereas the prediction with the state-space shown in [143] was improved with all the EPNet algorithms implemented.

Table 4.14: Comparison of different prediction methods for the $MG17_A$ and $MG30$ TS with NRMSE. A boldface number indicates the smallest error found

Method	$MG17_A$		$MG30$	
	Average	*Best*	*Average*	*Best*
Boosting [151]	**0.0126 (0.004)**	0.0114	**0.021** (0.0053)	0.0202
EBPTT [151]	0.0249 (0.0086)	0.0114	0.0429 (0.029)	**0.0071**
SNC-Elman [60]	-	-	-	0.0639
EPNet Fix	0.290962 (0.3467)	0.126736	0.100739 (0.0668)	0.039384
FS-EPNetFix&Evo.	0.051479 (0.0290)	0.017288	0.131987 (0.0860)	0.036434
FS-EPNet	0.027260 (0.0269)	0.007203	0.090789 (0.0418)	0.018697
FS_{AD}-EPNet	0.041664 (0.0765)	**0.006172**	0.105963 (0.0719)	0.022401

4.6.3 Mackey-Glass TS prediction problem

Table 4.14 presents the comparisons for the $MG17_A$ (columns 2 and 3) and for the $MG30$ (columns 4 and 5) TS. The FS-EPNet and FS$_{AD}$-EPNet algorithms obtained the smallest error for the best results in Table 4.14 for the $MG17_A$ TS. However, they could not improve the average values obtained by the boosting algorithm [151]. The EPNet algorithm is characterized by an ability to maintain the diversity of the population, which means that it is not necessary for all individuals to have an accurate error. The effect of this is that several of them can drive the mean error to bigger values. As the best results were found with the FS-EPNet and FS$_{AD}$-EPNet algorithms for $MG17_A$, it is clear that if fewer individuals in the population are considered for the calculation of the mean value, this would probably result in a smaller error than [151]. Even if the average error over all the population cannot produce the best results, it can be seen that the performance of the EPNet is acceptable for this TS too.

In the case of the $MG30$ TS, it was not possible to obtain better results too, but again, the FS-EPNet algorithm obtain closer results in terms of average and best vales. Moreover, the EPNet does not uses more complicated representations like the boosting approach to perform the prediction, and even then, it can have very similar results for $MG17_A$ and $MG30$.

Table 4.15: Comparison of different prediction methods for the $MG17$ TS with RMSE. A boldface number indicates the smallest error found

Method	*Average*	*Best*
AR model [139]	-	0.19
Pseudo Gaussian RBF [139]	-	0.00287
ARMA NN [149]	-	**0.0025**
EPNet Fix	0.302249	0.159894
FS-EPNet Fix&Evo.	0.078945	0.041114
FS-EPNet	**0.048343**	0.021273
FS_{AD}-EPNet	0.071895	0.009404

Table 4.15 presents the comparison of different algorithms for the $MG17$ TS with RMSE. In this case it was possible to obtain improved performance with the FS-EPNet algorithm over the average values but not for the best results found.

4.6.4 Breast cancer data set

As for several prediction tasks, it can be seen in Table 4.16 that the EPNet algorithms with its variations provided the smallest average error for the *Breast cancer* data set.

Table 4.16: Comparison between algorithms in terms of the best classification error on the Wisconsin *Breast cancer* data set. A boldface number indicates the smallest error found

Method	*Test error rate* (%)		
	Average	*Std. Dev.*	*Best*
OC1-GA [154]	3.8	1.0	-
GANet [47]	-	-	1.06
FNNCA [155]	-	-	1.45
HDANNs [156]	-	-	1.14
Classical EPNet [35]	1.37	0.938	**0**
GSOANN [157]	-	-	0.65
EPNet	**0.4380**	0.466	**0**
FS_{AD}-EPNet	**0.4380**	0.288	**0**

Table 4.17: Comparison between algorithms in terms of the best classification error on the *Optical digits* data set. A boldface number indicates the smallest error found

Method	*Test error rate* (%)		
	Average	*Std. Dev.*	*Best*
EWLDR [158]	-	-	5.9
OC1-GA [154]	9.8	1.1	-
EPNet	**5.3252**	0.715	**4.229**
FS$_{AD}$-EPNet	6.1242	0.674	4.785

Table 4.18: Comparison between algorithms in terms of the best classification error on the *Thyroid* data set. A boldface number indicates the smallest error found

Method	*Test error rate* (%)		
	Average	*Std. Dev.*	*Best*
EWLDR [158]	-	-	5.78
MLP 20 hidden units [159]	3.61	0.78	-
MLP 10 hidden units [159]	4.26	0.34	-
Classical EPNet [35]	2.11	0.22	1.6
HDANNs [156]	-	-	1.27
EPNet	1.5752	0.159	1.312
FS$_{AD}$-EPNet	**1.4488**	0.167	**0.933**

4.6.5 Optical digit data set

The optical digit data set is one of the most difficult tasks solve in this study given the number of inputs it requires. Even that, the EPNet algorithm implemented here with the FS$_{AD}$-EPNet improvement gave smaller average error than previous studies as shown in Table 4.17 corroborating the robustness of the algorithms.

4.6.6 Thyroid data set

In the last case, it is presented the comparison for the *Thyroid* data set in Table 4.18 showing that both algorithms found the smallest error on average and over the best individuals than in previous studies.

4.7 Discussion

Through this chapter has been presented the EPNet algorithm with an analysis of the various parameters used in it to evolve ANNs for prediction and classification tasks. The algorithm was found to be less sensitive to variations of some parameters, like the population size or initial learning rate, than others, in particular the *successful training* parameter. It was shown how important it is to set this with an appropriate value, and the relationship it has with the number of epochs in the partial training and with the hybrid training algorithm (MBP and SA). It was also presented the training, validation and test sets for prediction and classification tasks and remarked why it was used an extra test set inside the evolution to obtain the fitness of the individuals for the TS forecasting with the MSP method.

Thereafter, the EPNet algorithm was extended to be able to evolve the features: inputs for classification and prediction tasks, and delays only for the later. It was found that the evolution of inputs lead in the majority of cases to smaller errors than if the fixed input case is used. Nevertheless, it was not possible to improve all the results by evolving the inputs. Clearly the mutation of inputs during evolution is a noisy process that could affect the evolution of networks. At the end it was also compared the result of the feature evolution and fixed case against previous results in the literature showing a robust performance in the EPNet and the three variations of it for feature input selection through evolution. An important aspect to remark is that existing operator to evolve the architectures have been used to evolve the inputs, avoiding complex implementations, being only added the delay mutation operator.

As the EPNet algorithm has been introduced and extended for feature evolution, its further extension for the module evolution can be considered. Note that the analysis carried out in this chapter will be required for the next chapter, i.e. to understand the basis of the algorithm and to clarify further mutation operator designed for MNNs. Moreover, this study may not be complete if we just evolve MNNs with the single EPNet algorithm, but now we have the confident of its behaviour and the basis to explain its behaviour in a new field.

Chapter 5

Evolution of Modular Neural Networks

So far, we have presented the EPNet algorithms and their enhanced versions which improve feature selection in classification and prediction tasks. Also, we saw in Chapters 1 and 2 that modularity is known to have benefits for neural systems and their evolution, and this chapter aims to improve the EPNet to take advantage of those benefits. Its improved version, M-EPNet, will be in terms of new mutation operators that favour the formation of modules (independent partition of nodes), where pure-modular architectures are expected to appear if they are beneficial for the task in hand, without restricting the emergence of homogeneous architectures.

The new modular operators aim to simulate low connectivity levels between modules during evolution (2 new operators), while they encourage the increase of intra-modular connections (1 new operator). This will help to reduce cross-talk interference during evolution, if it should occur. Therefore in this chapter, low levels of connectivity plus cross-talk interference will be the key points for the evolution of pure-modular architectures. This behaviour is a similar to that present in the human brain (Sections 1.1 and 2.3.1) which favours the development of modules.

The modularity measure presented by [17] has proved to be suitable to analyze modules during evolution for the much studied what–where task [16], so that will be used here as explained in Section 5.1. Moreover, the information provided by the modularity measure allows us to discover the node dependency on a given output partition (or module), and in terms of new contributions, this information has been analyzed differently from how this was done in previous studies. Here, it will be used to discover cross-connections between nodes from different modules, which can be translated to discover *shared nodes* and *shared connections*, i.e. neural components that contribute to more than one module or output partition. Also, the discovery of shared neural components allows the rearrangement of nodes (Section 5.1.1) into modules for a more clear graphical representation of the final modules evolved, which means a different way to analyze the evolved networks. The fact that the modularity measure was previously developed for networks with more than one output unit, leads to the second contribution of this chapter, in Section 5.1.4, where the modularity measure is extended to networks with a single output unit. Moreover, it can be applied to any neuron at any level in the network to provide a fine analysis of node partitioning.

As was stated previously, the EPNet algorithm uses a direct encoding to represent the phenotypes (networks), thus, in Section 5.2 is presented an analysis of the connectivity matrix representing different patterns from ANNs and pure-modular architectures to distinguish the harder work required to evolve modular architectures. Using the information from the modularity measure, the final contribution of this chapter is presented in Section 5.3 with the improved version of the EPNet algorithm to evolve modular neural networks through its expansion with three new mutation operators: shared node deletion, shared connection deletion and intra-module connection addition.

In Section 3.3 the idea was introduced of how two or more data sets could be combined to test the algorithms in the module formation. With this in mind, the experimental results of this chapter are divided into separate sections, each one focusing on a particular case of module formation: 1) the evolution of single tasks (e.g. *Lorenz* TS) will be presented in Section 5.4 applying the extended modularity measure for networks with one output unit, as well as investigating module formation in tasks like $A1$, which are distinguished by having 2 output units or more; 2) the evolution of two compound tasks is shown in Section 5.5, where the tasks to be solved are similar, i.e. the *A1-B1* data set, or $Lo_A^{(ssp)}$-$Lo_{B1}^{(ssp)}$ TS; 3) later, in Section 5.6 the evolution of two compound tasks is shown, where they are less similar, e.g. *Logistic*-$Lo_A^{(ssp)}$ and 4) finally the case where three tasks are evolved at the same time in Section 5.7. The end of the chapter contains the discussion and conclusion, in Section 5.8.

5.1 Modularity measure

In Section 2.3.2.3 different reasons were given for choosing the modularity measure of [17] to determine the degree of modularity of the networks during evolution. Nevertheless, the choice of measure is not crucial. Any modularity measure could be used for the same purpose (identifying shared neural components in different modules) after the required modifications. Also, it should be noted that in this study, communication between modules is allowed and the shared neural components among partitions are analyzed.

The chosen modularity measure is based on the assumption that the neural networks have to deal with completely separable problems (non-interacting subsystems) to determine the dependency of nodes against each module. Here, it is assumed that it is known a priori how to partition the given data set into modular tasks, i.e. how many modules are appropriate and which input or output units belong to each module. The algorithm was implemented in this way (as it was developed) to keep the approach as simple as possible.

For the purpose of this work, this definition of modularity is sufficient to explore the interaction between modules during evolution. Further improvements may automatically determine data partition during evolution. Nevertheless, the usage of other modularity measures may result in a different behaviour as the presented in the following sections. The algorithm is based on the idea of data partition driven by the network's input and output nodes. The data sets used in this study involve two or more tasks, so there are m modules defined by the output partition. Since there are m possible partition types (either from inputs or outputs), there are at least two ways to calculate the modularity, i.e. in a 'top-down' fashion (from output to input units) if the output partition is known, or in a 'bottom-up' fashion (from input to output nodes) in the other case. The following explanation is focused on the 'top-down' version, but its counterpart differs little (refer to [17] for the 'bottom-up' version).

Since a separable problem is assumed, the set of output nodes $(y_1, ..., y_n)$ can be partitioned into m disjoint subsets $(S_1, ..., S_m)$. Nodes that are connected directly or indirectly to one subset of outputs S_j are called *pure* nodes as they contribute only to one module (output partition). The modularity measure M is defined as the average degree of pureness of the hidden and input nodes given a m-tuple $(d_i(1), ..., d_i(m))$ for each node i. This tuple indicates the degree of dependency of each node i on the m different partition or module (S_j). The m-tuple for outputs is the first to be assigned:

$$d_i(j) = \begin{cases} 1 & y_i \in S_j \\ 0 & y_i \notin S_j \end{cases} \tag{5.1}$$

After the $d_i(j)$ are calculated for the output nodes, the m-tuple is calculated for the hidden and input nodes recursively:

$$d'_i(j) = \sum_k |w_{ik}| d_k(j) \tag{5.2}$$

$$d_i(j) = \frac{d'_i(j)}{\sum_{j'=1}^{m} d'_i(j')} \tag{5.3}$$

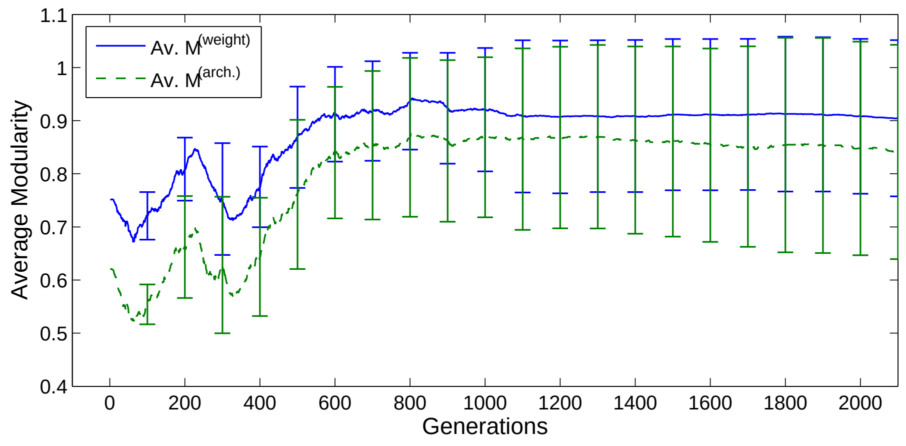
where w_{ik} is the connection weight from node k to node i, and $d'_i(j)$ is a partial processing value concerning w_{ik} with the tuple of node k at every position j. The pureness of each node i is calculated by the variance expression:

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(d_i(j) - \frac{1}{m}\right)^2 \tag{5.4}$$

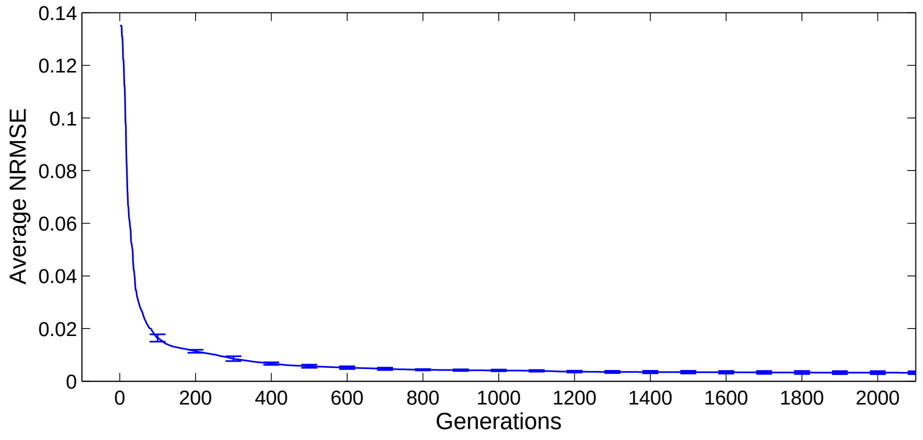
where higher σ_i^2 indicates higher pureness of node i. The maximum value of pureness possible is $\frac{m-1}{m^2}$ which corresponds to pure nodes. Finally, the modularity measure is given by the average variance of all N hidden and input nodes:

$$M^{(weight)} = \frac{m^2}{m-1}\frac{1}{N}\sum_i \sigma_i^2 \tag{5.5}$$

where $M^{(weight)}$ falls in the interval [0,1], with 1 indicating a completely separable network, and 0 a completely homogeneous network. If the weights were not included in equation 5.2, it would lead instead to a measure $M^{(arch.)}$ of the modularity only in terms of the structure of the network. Therefore, it can be seen that calculating $M^{(weight)}$ produces a finer measure of the dependency of each node against all output partitions, since the weights are involved in the calculation. At the beginning of the evolution, lower modularity values are expected because of strong interaction among modules, as well as by the type of connection scheme used. Then, as the generations advance, the interactions are expected to decrease, producing an increase in the modularity if the given task finds modular architectures beneficial, e.g. tasks with a high cross-talk interference. For example, consider Fig. 5.1 where the evolution of the $Lo_A - MG17_A^{(ssp)}$ (compound task) is presented during 2000 generation of evolution with the M-EPNet (explained in Section 5.3). Considering the differences between both tasks (higher cross-talk interference expected) and the usage of a modular algorithm, it is understandable the increase of both modularities (as the generations advance) from Fig. 5.1a. Also, it should be noticed that the error presented in Fig. 5.1b decrement as the modularity increment. There is possible to see a rise in both modularity values, mainly due by the modular algorithm and the cross-talk interference, which is expected to be high considering the difference between both tasks.



(a)



(b)

Figure 5.1: Example of modularity increase with the $Lo_A - MG17_A^{(ssp)}$ TS. Fig. 5.1a shows the evolution of both modularity measures (average over the entire population) during 2000 generations of evolution and Fig. 5.1b shows the average error considering both tasks.

Thus far, it has been assumed that the output partition is known, which implies the existence of two or more outputs. Thus the above $M^{(weight)}$ and $M^{(arch.)}$ can only be applied to networks with more than one output unit. Modifications are required for applications with a single output (see Section 5.1.4).

5.1.1 Node rearrangement

In following sections we will talk about node rearrangement, i.e. how nodes are sorted into modules. What this really means is that the actual position held by the nodes in the algorithm is never touched, i.e. a node with index n is always node number n. But in the graphical representation, it may be difficult to see which nodes contribute more to a certain module if we plot them as is conventionally done in the literature, i.e. as shown in Fig. 5.2a. However, if we plot nodes of the same module together as is done in Fig. 5.2b, it will be easy to understand the relationship of the neural components with each module or output partition, from a graphical representation. This will clearly help to understand more about modules.

5.1.2 Shared nodes

As previously remarked, shared nodes and shared connections are neural components that contribute to more than one module or output partition.

Figure 5.2 shows a neural network with two output nodes in its normal representation (Fig. 5.2a) and with the nodes rearranged into modules (Fig. 5.2b). Input nodes are represented at the bottom, hidden nodes in the middle, and output nodes at the top. Using equations 5.3 and 5.4 one can determine the extent to which a given node is connected only with a single output partition. In this case, each node will be bound (associated) with the module for which it presents the biggest dependency, i.e. the hidden nodes are sorted or rearranged into modules (Section 5.1.1).

Thus, from equation 5.3 and 5.4, nodes 1, 5, 6, 7 and 9 in Fig. 5.2a are pure nodes in the top-down fashion as they are connected directly or indirectly to only one output partition (module). The rest of the input and hidden nodes are shared to a certain degree. Note that in this case, only $M^{(arch.)}$ has been used to determine the purity of nodes, but $M^{(weight)}$ might give a finer-grained measure.

5.1.3 Shared connections

If certain weights are assigned to the connections in the network presented in Fig. 5.2a, equation 5.3 can be used to sort the nodes into modules as presented in Fig. 5.2b, where

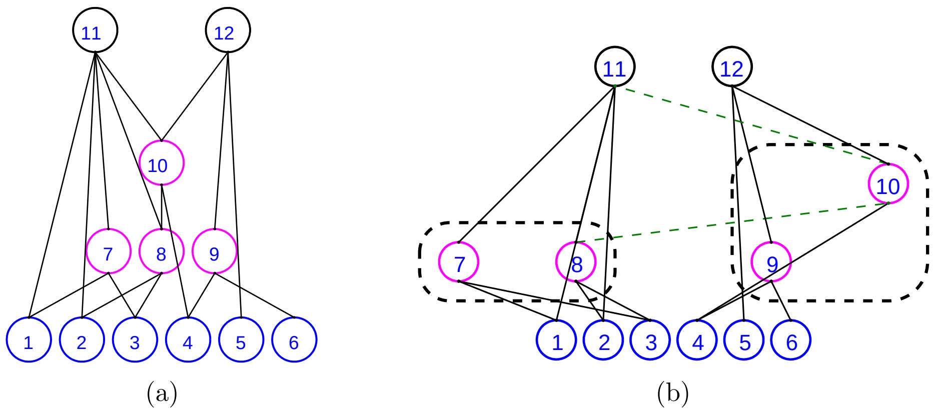


Figure 5.2: Typical neural network with two output nodes, shown as normal (Fig. 5.2a) and modular (Fig. 5.2b) representations

the shared connections are represented by dashed lines connecting nodes from different modules. The input and output nodes are not included in the graphical representation of the modules here (i.e. nodes grouped by dashed lines), but they may be included if desired.

After the nodes have been organized into modules and shared nodes detected, it is straightforward to discover the cross-connections between modules, as illustrated in Fig. 5.2b where connections $c_{8,10}$ and $c_{10,11}$ shown as dashed lines are shared by two modules. Note that if the weights take different values, such shared nodes may be assigned to different modules, e.g. node 8 could be moved into the module formed with node 12.

Note that equation 5.3, leading to $M^{(weight)}$, allows a unique rearrangement of nodes into modules, and shared nodes and connections can be found in the same modularity calculation. However, that is not possible using equation 5.3 without the weights, leading to $M^{(arch.)}$, because it measures only the architecture modularity. For example, there is no way to know if node 10 contributes more to the output partition formed by node 11 or the partition composed by node 12 without considering both weights $w_{10,11}$ and $w_{10,12}$ in Fig. 5.2a.

Summarizing, the shared components can be found in different ways. One option could be the use of $M^{(weight)}$ to sort out the nodes into modules and then identify the cross-connections between modules. Having found the shared connections, shared nodes can be discovered as they are the neurons that contain the cross-connections, or just the equations 5.4 or 5.3 can be used to find the nodes. But if the shared nodes are looked for first, i.e. a separate module is created for this kind of node, it is not possible to discover the shared connections with this information as there may be outbound connections from the shared node to more than one module (assuming that there may be at least two modules without counting the shared module).

5.1.4 Improved modularity measure

It is not straightforward to apply the above modularity measure to networks with only a single output unit. In that case, the network presented in Fig. 5.3a would have $M^{(arch.)} = M^{(weight)} = 0$ in the top-down approach, as there is only one output node and thus only one partition. However, it is possible to treat networks with one output as being comprised of a number of sub-networks (or an ensemble of sub-networks [9, 67, 68, 70]) and base the partitioning on the combined set of outputs of the constituent networks. To adapt the above modularity measure to networks with a single output unit, one needs to look for the internal connectivity patterns inside the network to discover structures similar as those in modular architectures without taking the output node into account. Thus the improved modularity measure will not consider the single output unit in the calculation of $M^{(arch.)}$ or $M^{(weight)}$, but this node and its inbound connections will still be used elsewhere.

Figure 5.3 illustrates the use of the improved modularity measure for networks with a single output node. Note that this network is the same network as in Fig. 5.2, except for the new output node 13 which relegates the former output nodes 11 and 12 to the position of hidden nodes. This simple relation will prove helpful in illustrating the improved modularity measure. Since the output node 13 will not now be considered in the modularity evaluation, it and its connections $c_{11,13}$ and $c_{12,13}$ will be referred to as *virtual nodes* and *virtual connections* for the purpose of this section, because they exist in the real network, and are used to identify modules, but do not take part in the modularity evaluation.

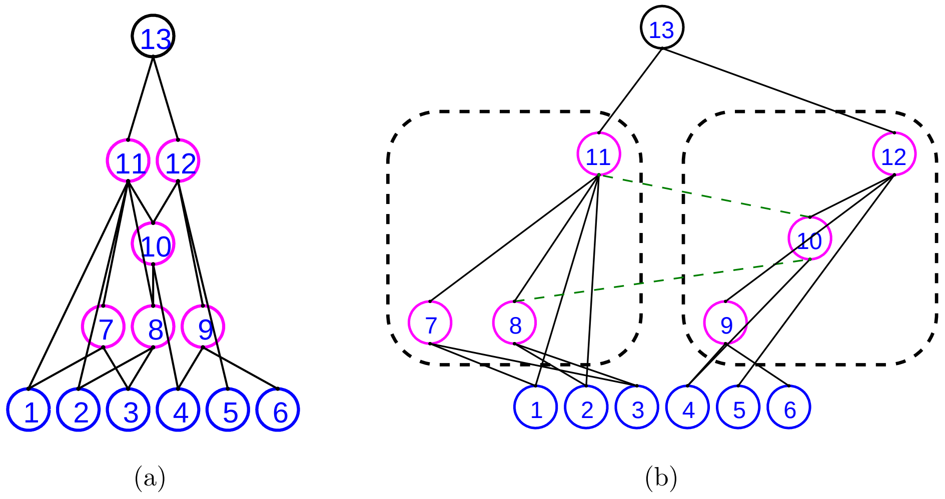


Figure 5.3: Neural network with one output unit, normal (Fig. 5.3a) and modular (Fig. 5.3b) representation

In such networks, all the inbound connections into the final output neuron come from the outputs of each module, i.e. one output per module is connected to the final output unit (virtual node 13 in Fig. 5.3a). After the final output neuron has been omitted, one can identify all the inbound connections to it and take the corresponding nodes to be potential new outputs that will lead to the structure of a multiple-output modular neural network.

However, not all of these potential new outputs should be considered as actual new outputs of the network (i.e. module outputs of the full network). For example, suppose the nodes are ordered in an incremental way over a connectivity matrix C, and nodes i and j are connected to the output unit k (i.e. there exist connections $c_{i,k}$ and $c_{j,k}$). If there exists another connection $c_{i,j}$ where $i < j < k$, the connection $c_{i,k}$ should not be considered to be the output of a possible module because one of the outputs of node i is connected to another hidden node (node j, because $c_{i,j}$ exists). In a hierarchical representation, node i is then an internal unit of a possible module formed by node j because node j needs to process the output of node i to produce its output, regardless of whether node i is directly connected to node k. For that reason, the following two restrictions need to be imposed to determine which hidden nodes will be considered as the new outputs of the network, i.e.

nodes that have a virtual connection. If H and O are the sets of hidden and output nodes respectively, then:

1. Node i is considered an internal unit inside the module formed by node j if $\exists c_{i,j}, \forall i, j \in H$ where $i < j$.

2. The output of node j $(c_{j,k}, k \in O)$ is considered the output of module k until $\exists c_{j,l}, \forall j, l \in H$ where $j < l$.

This means that a node i will be considered as the output of a new module k if and only if node i has a virtual connection and there is no other outbound connection to a subsequent node. In Fig. 5.3a, nodes 11 and 12 are the only nodes that have a virtual connection to node 13. If other hidden nodes had a connection to node 13, they could not be considered as the output of a new module because they have connections to subsequent nodes that in turn are connected to the original output node. Finally, applying equations 5.1 to 5.5 to the new partitioning leads to the improved modularity measure. Clearly, there will always be at least one hidden node in the original network that satisfies the requirements for becoming a new output node, i.e. the last hidden node in the network which was originally connected to the virtual node. If there is only one such new output node, that would again lead to $M^{(weight)} = M^{(arch.)} = 0$, as there would be no modular structures identified inside the network. However, one can then repeat the above process to identify modules that feed into that output node. This definition of modularity for top-down networks with a single output unit fits with the conventional definition of ensembles and MNNs. Each module has one output node, and all the module outputs are fed into the final node of the complete network, even if there exist other nodes that have a direct connection to the final output node.

An important aspect of this process for identifying modules is that it is not restricted to looking for modules that feed into the final output of a network, but can equally well be applied to any node (for any sub-network) in the network to produce a fine-grained measure of modularity at a local level. For example, if a detailed analysis of a single node j is required, it is possible to apply the improved modularity measure to the sub-network feeding into it. That will allow the discovery of smaller partitions of nodes in the substructure of the network. This is a similar process to that of Newman [166] where the nodes are divided into communities to create a 'dendrogram' (a tree pattern representing the connectivity clusters among nodes). However, that approach needs to search for the best partition of nodes that increases a modularity measure Q as remarked in Section 2.3.2.3. In the process defined above, there is no need for an extra optimization stage to determine the modularity, and that makes it much more efficient for using repeatedly in an extended evolutionary process.

If the nodes of the network in Fig. 5.3a are rearranged according to the above process (using the same connection weights as for Fig. 5.2b up to node 12), the modular representation shown in Fig. 5.3b is obtained. Comparing Figs. 5.2b and 5.3b shows that the process has, as required, formed the same modules with the same nodes. In the examples presented here, a graphical representation of the shared modules has not been shown, i.e. modules formed by shared nodes. If that had been done, the networks presented in Figs. 5.2b and 5.3b would have an extra module containing the nodes 8 and 10 as shown in Fig. 5.4. This is similar to [12] where the modularity was measured in terms of the number of nodes in the shared module.

The process presented here allows the analysis of neural networks to create modules using a modularity measure, identifies modules using a shared module (discovering shared nodes), and enables the analysis of cross-connections between modules over networks with one or multiple output units. Although such an automated process for identifying modules and measuring modularity can clearly be used as a very general tool for analyzing neural network structures, in this study it will simply be used within an evolutionary neural network algorithm to facilitate the evolution of neural architectures with increased modularity.

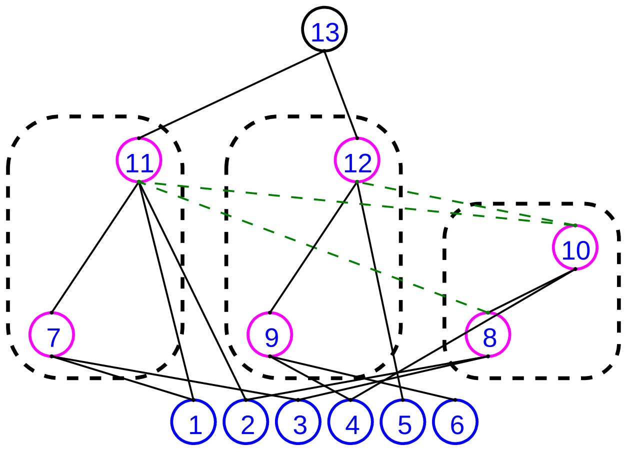


Figure 5.4: Module representation from Fig. 5.3a with a module for shared nodes

5.2 Connectivity matrix

To represent the GMLP connectivity, the EPNet algorithm uses a network connectivity matrix C as shown in Fig. 5.5. As feed-forward networks are used here, the connectivity matrix for this approach consists of five sub-matrices (IH, IO, HH, HO and OO) in the upper-right part of it, where the rows represent source nodes and columns represent destination nodes. Thus, the GMLP allows connections from inputs to hidden nodes (IH), inputs to outputs (IO) and so on until connections from output to output nodes (OO sub-matrix) are reached. This can be used to highlight the differences between MLPs and GMLPs for general neural networks and MNNs.

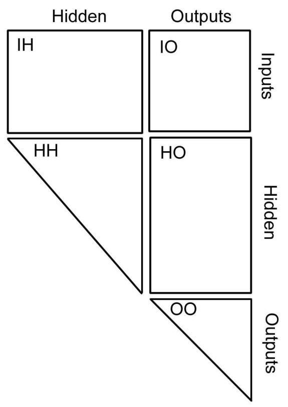


Figure 5.5: Sub-matrices in the network connectivity matrix

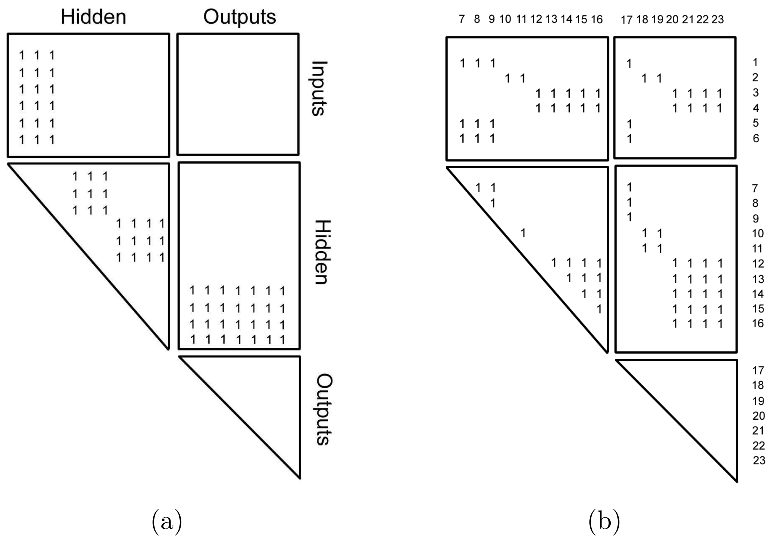


Figure 5.6: Sub-matrices for MLPs and GMLPs in the connectivity matrix. Fig. 5.6a represents a fully connected MLP whereas Fig. 5.6b shows a fully connected pure-modular architecture using GMLPs without considering OO sub-matrix

Fig. 5.6 presents the case of a fully connected MLP (Fig. 5.6a) and a fully connected pure-modular architecture with GMLP representation without considering connections between nodes in OO sub-matrix. If GMLPs are compared with MLPs, it can be seen that MLPs use IH, HH and HO when there is more than one hidden layer. For the simplest case of one hidden layer, MLPs require only the sub-matrices IH and HO (not presented here). In general, IH and HH are more restricted in MLPs as not all possible connections are allowed in those sub-matrices.

If a MLP or a pure MNN is analyzed using the connectivity matrix, a clear pattern of non-overlapping sections will be evident in each sub-matrix (as shown in Fig. 5.6). Since the GMLP is used here as the basis for evolving MNNs with the EPNet algorithm, it will require considerable effort during evolution to reach such non-overlapping patterns. However, that is the inevitable cost of allowing the evolution of more general, and potentially more powerful, architectures (rather than MLPs). This is why a biasing modification of the standard EPNet algorithm is required.

5.2.1 Connection scheme

Similar to the EPNet algorithm, the population of networks for the M-EPNet was initialized with a simplified connection scheme set using the same probability ϕ for each connection in the connectivity matrix C. For a recurrent network with n nodes, the expected number of connections would be $n^2\phi - n$, assuming there are no loops from any node to itself. In the feedforward network case studied here it is $N\phi$, where N is the maximum total number of connections allowed (the upper-right part of the connectivity matrix as shown in Fig. 5.5). Different values of ϕ were tested to determine how it affected the performance of the algorithms in this chapter.

5.3 Modular EPNet algorithm (M-EPNet)

The aim of implementing a modular version of the EPNet algorithm arises because the normal algorithm is found to be too slow to delete the appropriate neural components that increase the average modularity in the population. That is mainly because the nodes and connections are taken at random to be mutated. Therefore, it is likely that more generations of evolution will need to be used to reach the connectivity pattern required than would be needed using specific operators that delete certain neural elements to increase the modularity faster. Moreover, some tasks may be solved equally well with homogeneous or modular architectures, thus, extra pressure is needed to allow the module formation.

Thus, the required modifications of the EPNet algorithm are the addition of three new mutation operators: 1) shared nodes deletion; 2) shared connections deletion and 3) intra-module connection addition. These may allow the increase of modularity during evolution. Clearly those new mutation operators can be considered a subset of the general operators of the EPNet algorithm. Also, it may be expected that any EA will find independent partition of nodes if that is beneficial for the task in hand. The idea of shared node deletion has previously been implemented in a more direct manner [12]. In that case, there is a simple set of parameters which specify how many hidden nodes in a single hidden layer MLP connect to each subset of the outputs. If there are only two output units, deleting the shared nodes then simply corresponds to reducing a single parameter value. In this way, the evolution of modules is controlled, and the degree of modularity can be monitored, directly with that single parameter, and no additional measure of modularity is required. Clearly, the more complex connectivity patterns of the GMLP networks of interest here require a more sophisticated approach. Moreover, simply counting the number of connections does not take into account how many of the associated connection weights have been reduced to zero (or near zero) by the learning algorithm. This is where the above modularity measures are required.

In [17] it is noted that a learning algorithm may reduce the weights of the cross-connections to a value close to zero if required by the task at hand, i.e. if a modular architecture is suitable for the task, rather than a homogeneous network. For that reason it is usually found that $M^{(weight)}$ achieves bigger modularity value than $M^{(arch.)}$, as shown in Fig. 5.1. Given this information, the specific deletion of a shared connection with an operator designed for that purpose should not have any negative effect on the fitness of the network. Instead, if the shared connection selected was causing some interference in the learning process, its deletion may produce a reduction of error as well as the increase in both modularities ($M^{(arch.)}$ and $M^{(weight)}$).

The mutations of the M-EPNet algorithm are presented in Fig. 5.7, with the same overall procedure as shown in Fig. 2.2a. The modification, of direct relevance to modularity, lies in the introduction of three new mutation operators (shared node and shared connection deletions, and intra-module connection addition) as the first architectural mutations. If these are not successful (i.e. they do not produce a better individual than the least fit one in the existing population), then the general node and connection deletion mutations are followed as in the original algorithm. At the end come the standard addition mutations. If the task in hand does not require a modular network, then the operators designed to

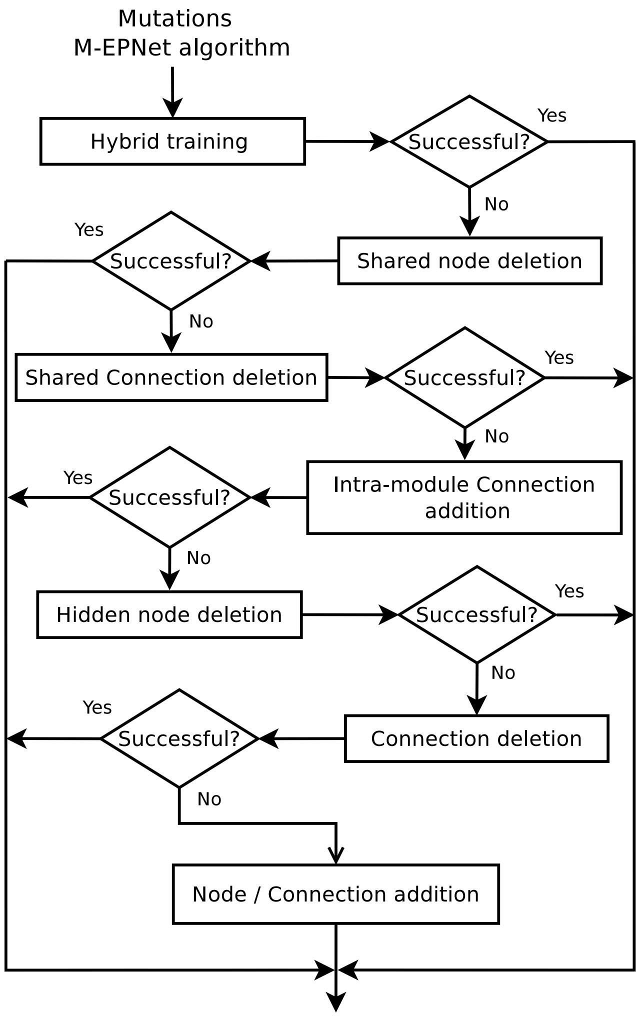


Figure 5.7: The Modular EPNet mutations

increase the modularity may not be used. However, if the task can be performed better or equally well by a modular network, then it may be expected that those mutations will be used more than the other mutations in the algorithm.

If a shared node is deleted, it is likely to decrease the number of shared connections in the network. Similarly, a deletion in a shared connection can cause a shared node to become a pure node. Also, every time the weights are updated, there is the possibility that a node will switch to another module, i.e. while the learning procedure does not affect $M^{(arch.)}$, it can change $M^{(weight)}$.

5.3.1 Feature selection

If feature selection is required (as previously seen in Chapter 4), the modification of this algorithm may be straightforward, with the introduction of the new operators after the modular ones, i.e. after the intra-module connection addition. For prediction tasks the configuration of the FS-EPNet algorithm was used, while for classification tasks the FS$_{AD}$-EPNet algorithm was used to evolve the features in this chapter, i.e. all the experiments designed for two or more compound tasks presented below use the evolution of input features.

Even though there are different rearrangements of mutation operators in both algorithms, we will hereafter just make reference to the M-EPNet algorithm. However, it may be advantageous to bear in mind that it may use feature selection mutations from the FS-EPNet algorithm for prediction tasks, or the input addition/deletion mutation from the FS$_{AD}$-EPNet algorithm for classification tasks.

5.3.2 Operators and their similarity to biological processes

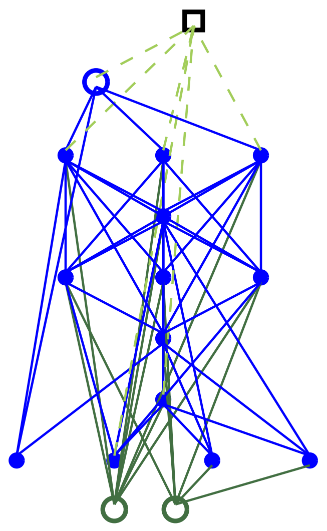
Considering how little information we have about the human brain, and taking into account that we do not know exactly how modules appeared within it, we simply have to assume that there exist different processes in the brain to deliver module formation. However, since the brain is constrained by neuro-biological factors, there may be some operations (in terms of connectivity) that may have a higher probability of being applied than others. That insight is used here to constrain connectivity during evolution, giving greater probability of module being formed during evolution through the new mutation operators introduced in the M-EPNet. Nevertheless, as was seen, the M-EPNet continues to have the previous operators of the standard algorithm, so that it too can evolve normal network architectures for a single task.

5.4 Evolution of single tasks looking for module formation

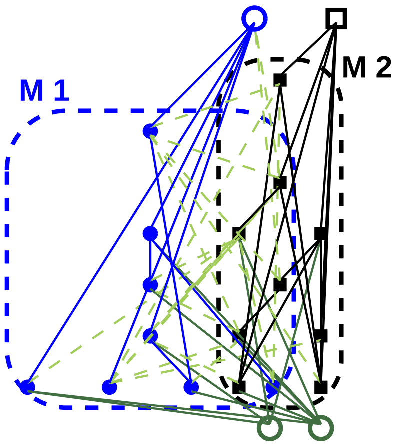
Now that the M-EPNet algorithm has been presented, as well as the modularity measure with its extension, the next thing is to test the EPNet and M-EPNet, looking at the modularity value to discover whether the mutation operators of the M-EPNet really does help to increase the modularity of networks. To corroborate that, the first series of experiments will be presented in this section, where single tasks are used to evolve modules inside networks.

Thus, the aim is to investigate if single tasks could form some kind of module that lets us know more about each task. At the same time, the improved modularity measure will be evaluated for networks with single output units, e.g. *Lorenz* TS. Therefore, in this section the EPNet and M-EPNet algorithms will be tested, with the TS and data sets used in Chapter 4 plus a representative sample of all artificial data sets from Fig. 3.3 in Chapter 3. In the case of the EPNet algorithm, we can still use the modularity measure during evolution even if no operator exists to encourage the module formation as in the M-EPNet, so we can still have a way to measure whether the normal EPNet algorithm found networks (and modules inside them) similar to those found by the M-EPNet algorithm.

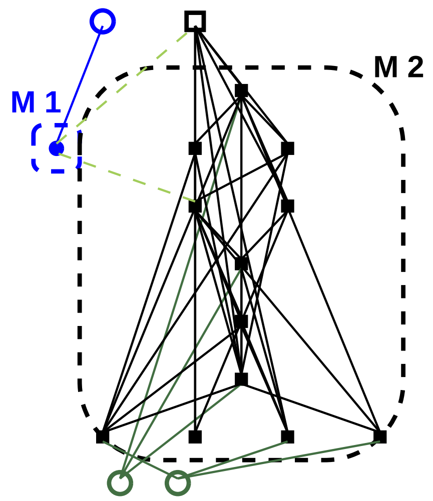
The first experiment tested data sets *A1, B1, C1* and *D1*, where 1000 values were used for training, 200 for validation and 200 for the test set. Fig. 5.8 presents the best network found after 30 independent trials for the $A1$ and $B1$ data sets with the EPNet and M-EPNet algorithms. As the standard format for the networks (figures displaying networks' architectures) presented in this thesis, the input nodes will be plotted at the bottom of the figure, hidden nodes in the middle and output nodes at the top. Input and output nodes will also be distinguished by different shapes (circles, squares, etc.) without infilling. Hidden nodes will have infilled shapes and all nodes of the same module will have the same shape and colour. Where each module has at least one hidden node, a dashed line will enclose all the nodes (as Fig. 5.8c), to better show the nodes that form the module. As



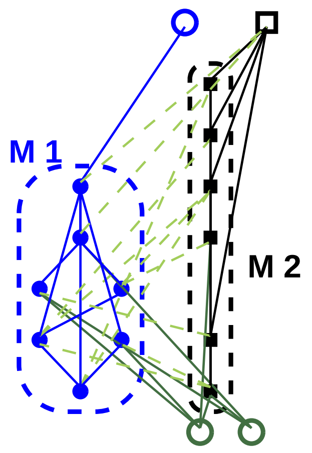
(a) *A1* - EPNet



(b) *B1* - EPNet



(c) *A1* - M-EPNet



(d) *B1* - M-EPNet

Figure 5.8: Best individual evolved for the *A1* and *B1* data sets with the EPNet and M-EPNet. Data set *A1* with the EPNet (Fig. 5.8a) and with the M-EPNet (Fig. 5.8c). Data set *B1* with the EPNet (Fig. 5.8b) and with the M-EPNet (Fig. 5.8d)

previously, dashed lines will represent shared connections, connecting nodes from different modules. The data sets presented in Fig. 5.8 contain a combined input, where the same two inputs can represent two different classification tasks, each one with two classes to classify. Thus, it is understandable that these inputs can be connected to any node of the network because they are not exclusive of one data partition. However, it may be the case that the inputs of different tasks are not mixed into a single set of inputs as shown in Section 5.5.1.1. Therefore, there is the option whether to consider the inputs in the modularity calculation. If they are not considered, all inputs will have a standard shape (circles without infilling) and solid lines representing connections to hidden and output nodes. If they are considered, inputs will have the same shape as the module to which they belong, and because shared connections may yet be discovered, they will also be present from input to hidden nodes (if that is the case).

As can be seen, in Fig. 5.8a the majority of the nodes contribute to class 1 (first output node from left to right) in data set *A1*, which corresponds to the circle class in Fig. 3.3a from Chapter 3. In Fig. 5.8c almost all nodes focus on class 2 (module $M2$) of data set *A1*, which is the class outside the circle from Fig. 3.3a. This is an interesting result because in one experiment (or independent run) all nodes collaborate to solve the class represented by the first output, while in a different experiment, all nodes contribute to solve the complement of the problem, having a bigger contribution in the second output node. Also we observe in Fig. 5.8 the differences in use between the EPNet and M-EPNet algorithms in tasks *A1* and *B1* when solved separately, where the M-EPNet obtained a lower quantity of shared connections than was obtained with the EPNet algorithm. In all cases, the networks were able to classify correctly all patterns of the test set. Note that here we have represented the classification task from Figs. 3.3a and 3.3d with two output nodes, which could also be done with just a single output.

Table 5.1 shows the modularity values for all tasks from the previous chapter plus the new artificial data sets (*A1, B1, C1* and *D1*) with the EPNet and M-EPNet algorithms. There it can be seen how the M-EPNet allows, on average bigger modularity values, indicating that it was beneficial to deliver more modular architectures. It may be noted that there were no statistically significant differences in the error obtained between both algorithms for all cases, which also indicated that the modularity can be increased without damaging the performance of the networks. That is mainly because the M-EPNet algorithm does

Table 5.1: Architectural modularity value results from the EPNet and M-EPNet algorithms over different prediction and classification tasks evolved independently

TS/Data set	*EPNet*				*M-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Logistic	0.253	0.328	0.000	1.000	0.480	0.440	0.000	1.000
$Lo_{A}^{(ssp)}$	0.266	0.352	0.000	1.000	0.461	0.363	0.000	1.000
$Lo_{B1}^{(ssp)}$	0.193	0.264	0.000	0.976	0.361	0.310	0.000	0.960
$Lo_{B2}^{(msp)}$	0.104	0.208	0.000	0.767	0.164	0.233	0.000	0.775
$Lo_{B3}^{(msp)}$	0.177	0.263	0.000	0.813	0.358	0.378	0.000	1.000
$MG17$	0.160	0.272	0.000	1.000	0.314	0.344	0.000	1.000
$MG17_A$	0.134	0.214	0.000	0.913	0.240	0.294	0.000	0.909
$MG17_A$	0.196	0.290	0.000	1.000	0.399	0.401	0.000	1.000
Breast cancer	0.265	0.285	0.000	1.000	0.501	0.374	0.000	1.000
Optical digit	0.066	0.017	0.042	0.098	0.092	0.033	0.041	0.147
Thyroid	0.157	0.134	0.028	0.551	0.483	0.243	0.119	0.963
$A1$	0.383	0.259	0.034	0.889	0.595	0.273	0.066	1.000
$B1$	0.360	0.190	0.085	0.829	0.409	0.132	0.180	0.811
$C1$	0.307	0.129	0.059	0.624	0.385	0.142	0.145	0.714
$D1$	0.320	0.121	0.079	0.483	0.481	0.211	0.156	0.758

not perform a mutation if that action produces worst results than the last fit individual (without counting the last addition mutations), similar to the EPNet algorithm.

In Table 5.1 the algorithms' stopping criteria were set as in previous chapters, and it will clearly deliver different results if the number of generations is instead fixed at some larger value. To test that, the classification tasks (*Breast, Optical digits* and *Thyroid*) data sets were evolved again without stopping the algorithms. As expected, the level of modularity increased with more generations. That indicates that even where a network is able to classify correctly all patterns of the data set, if its evolution is not stopped it is probable that different architectures may appear with bigger modularity values with the M-EPNet algorithm. The average error in prediction and classification tasks showed no statistical significance between both algorithms. Thus, the networks continue having similar performance, which means that they were not over-trained, mainly because they are mutated before partially trained, i.e. the hybrid training is not used if the successful training parameter is not set as success (which means that the error could be further reduced), on the other hand, the network is mutated before being partially trained.

It may be worth to say that bigger modularity values are desired because it will allow to identify modules that could be reused in later implementations. Moreover, the aim is the automatic emergence of modules during evolution (as in nature), instead of forcing them to be modular [15, 100, 118], i.e. finding the rules that help the module formation will be more beneficial for further studies than studying algorithms that always produce modular architectures.

From this series of experiments in this section, it can be concluded that even if we do not know how to subdivide a single task into sub-tasks, we can still use the insights presented here to discover what kind of modules emerge, which clearly will be quite useful to discover sub-tasks in this kind of problem. Also, it was shown that the modularity extension for networks with a single output unit can be used to deliver a measure of modularity during evolution. Here only an overview of the potential of using the M-EPNet algorithm for single tasks was presented, and more extensive research should be considered in the future. However, the rest of this thesis will be focused solely on the combination of two or more tasks, since this is where the benefits of modularity are likely to prove most useful.

5.5 Evolution of similar compound tasks

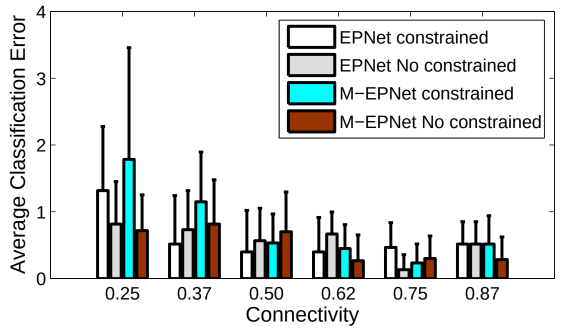
The objective of this section is to investigate whether two compound tasks, where they are similar, can produce independent partition of modules during evolution. A second objective is to investigate to what degree cross-talk interference could produce independent partitions of nodes, or if just having several connections in the network can absorb such interference, leading to pure-homogeneous networks during evolution. Here prediction and classification tasks are evaluated, and the results from this are presented in separate sub-sections as was previously done in Chapter 4. The compound tasks studied here are *A1-B1* and *C1-D1* (with 2 inputs and 4 outputs and with 4 inputs and 4 outputs), Lo_A-$Lo_{B1}^{(ssp)}$ TS where SSP is used to forecast both TS simultaneously and $Lo_A^{(ssp)}$ TS predicting $\Delta = 1$ and $\Delta = 5$ simultaneously. Thus, it can be seen that all of them have two sub-tasks to be solved by the algorithms.

The same stopping criteria presented in Section 4.1.2 were used here, where the number of generations is not fixed. Four different experiments were tested, with different connectivity values in the following way: the EPNet algorithm with constrained connections during the whole evolutionary process, including during the random initialization (EPNet constrained); the EPNet algorithm with constrained connections only in the random initialization of the individuals, i.e. the connections are not constrained during evolution ('EPNet Not constrained' during evolution); and the other two experiments were the same as the previous two but with the M-EPNet algorithm.

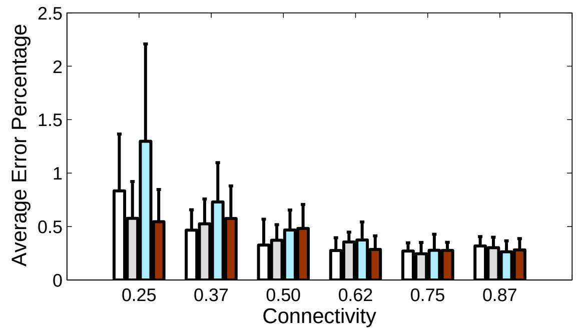
5.5.1 Two compound classification tasks

Figs. 5.9 and 5.10 show some average values obtained after the data set *A1-B1* was evolved with 2 inputs and 4 outputs. The generalization performance (final test set) with the average classification error is presented in Fig. 5.9a, which includes the legend for all sub-figures in Fig. 5.9.

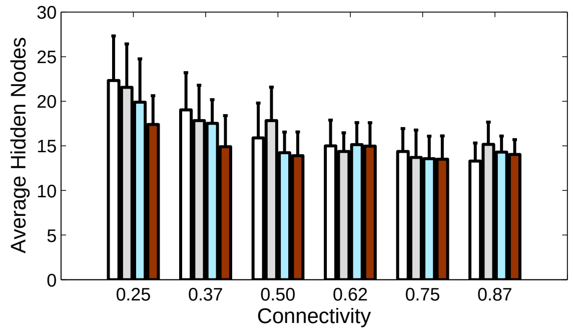
If we look at the performance case (Figs. 5.9a and 5.9b), it can be seen that as the level of connectivity is increased, the error is reduced for this data set, indicating that a greater number of connections allows a better solution of the problem. But it can also be observed that the greater the number of connections (Fig. 5.9d) the smaller the number of generations required (Fig. 5.9e) and the lower the modularity obtained (Figs. 5.10a and 5.10b). If we focus only on experiments that use a connectivity value of 0.25, it can be seen that the M-EPNet algorithm, without constrained connections during evolution, obtained a smaller error with bigger modularity values than the standard EPNet algorithm, corroborating



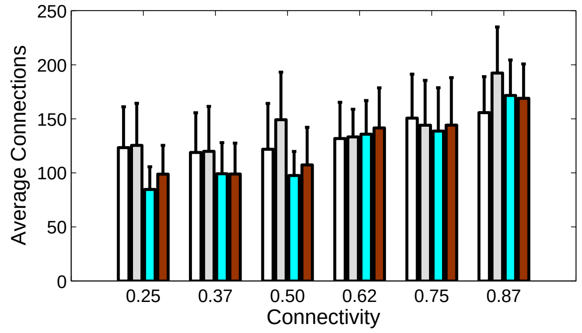
(a)



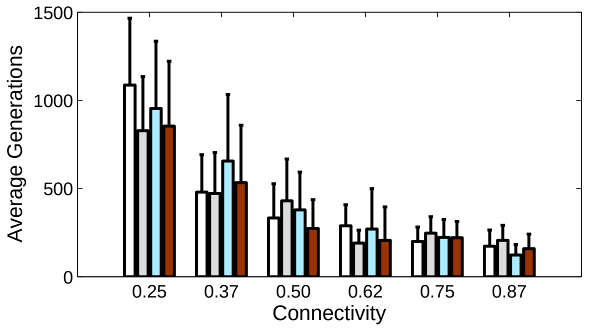
(b)



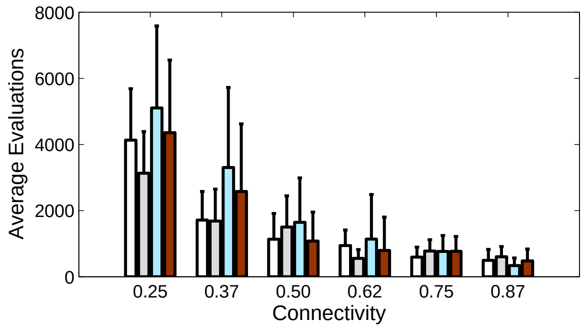
(c)



(d)



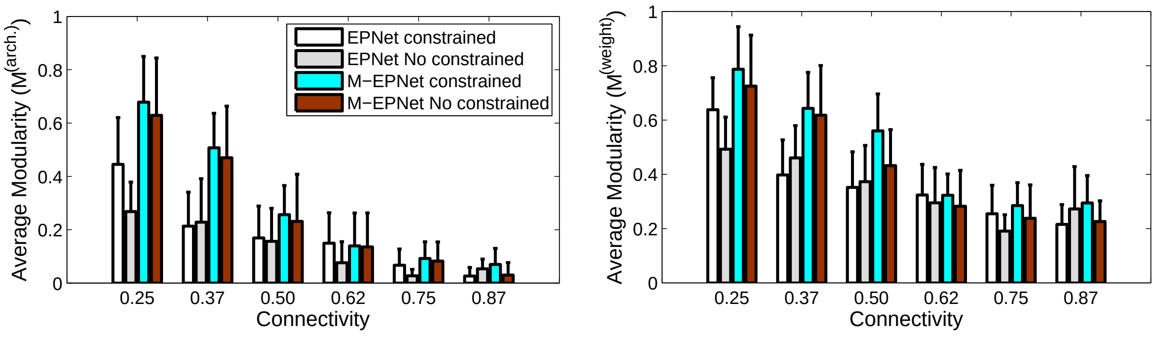
(e)



(f)

Figure 5.9: Evolved parameter values for *A1-B1* data set with 2 inputs and 4 outputs at different connectivity values with the EPNet and M-EPNet and constrained connections, or not, during evolution. Generalization performance with the classification error (Fig. 5.9a) and with the error percentage (Fig. 5.9b); hidden nodes and connections are shown in Figs. 5.9c and 5.9d respectively, while the average number of generations is presented in Fig. 5.9e and the number of evaluations is shown in Fig. 5.9f

the findings that, at lower connectivity values, the bigger the modularity, the better the performance. Also, it may be noticed that it obtained a smaller number of nodes and connections than the EPNet algorithm with and without constrained connections. However, as there are more operators in the modular algorithm, the number of evaluations was larger



(a) (b)

Figure 5.10: Architectural and weighted modularity values for *A1-B1* data set with 2 inputs and 4 outputs at different connectivity values with the EPNet and M-EPNet and constrained connections, or not, during evolution. The architecture is shown in Fig. 5.10a while the weighted modularity is presented in Fig. 5.10b

in general with the M-EPNet than with the standard algorithm. When the connections are constrained in the random initialization and during evolution (EPNet and M-EPNet constrained), the classification error and error percentage were larger than when connections are not constrained during evolution at low connectivity values. Clearly, constraining the connections during evolution did not provide any benefit in this case.

In different sections (e.g. Sections 2.4.1 or 5.3) the work of Bullinaria [12] was discussed, where a fixed number of nodes was used to solve this problem, and it was pointed out that this kind of task can easily make the EA increase the number of nodes and connections, as the more resources, the faster the solution of the problem. Here it has been corroborated that bigger connectivity values solve the problem with a smaller number of generations. The important finding in this part, is that the EPNet and M-EPNet algorithms could be initialized at any connectivity value and they will ensure that the average number of nodes and connections will not increase without limit.

In [12] a simulated evolution of three modules was also used, one for each task and a third one representing the shared nodes. The optimum number of nodes per module was evolved, where each node or module had full connectivity with the output layer. Then, a modification in the number of nodes can drastically change the modularity of the network. In a work derived from this thesis [161], the M-EPNet algorithm was used without the intra-module connection addition operator and it was noticed that lower modularity values (with statistical significant difference) were obtained in comparison with Fig. 5.10a.

This shows that mutating the networks in the right direction (increasing modularity) as may happen in [12] can favour the formation of networks with bigger modularity values, which allows a smaller error as shown in all the sub-figures from Fig. 5.9 with a connectivity of 0.25. Thus, the effectiveness of using the three new mutations for the modular case was more beneficial than just using the share nodes and shared connection deletion operators at low connectivity values. Considering this kind of task (*A1, B1, ...*), the data set *C1-D1* was also evaluated giving similar results to the *A1-B1* data set where at lower connectivity values the M-EPNet algorithm gave a better generalization performance.

5.5.1.1 Separate inputs

The data set presented in the previous section was tested with a combined input to classify different tasks. This presents an analogy with the brain, where the visual system can be seen as a compound input (through the eyes) that feeds a flow of information to the brain, and where it may be expected that different modules process different regions/objects of the input. This is similar to the *what–where* data set [16], where the input vector cannot be partitioned into the different sub-tasks because all of them use the information at the same time. On the other hand, there may be other flows of information into the brain that are not combined inputs, e.g. a visual task and a hearing task, which provide different channels of information to the brain; or from another angle, different inputs into the same system, in which the inputs could be partitioned and distinguished from each sub-task.

In this case, the *A1-B1* data set was investigated with separate inputs per task, i.e. 4 inputs instead of 2, where the first two inputs belong to data set $A1$ and the other two to $B1$. The aim here is to investigate whether it was easier for the networks to find the modules and to see if a given set of inputs could be correctly associated within the module to which they belong.

Fig. 5.11 shows the average classification errors on the final test set and the modularity obtained in the population at the end of the evolution with different connectivity values. As there were more inputs involved to solve the tasks, bigger errors were found with the constrained experiments during evolution at lower connectivity values than if the combined inputs were used as in Fig. 5.9a. On the other hand, it was easier for the M-EPNet to find networks with bigger modularity values when the inputs were separated, which seems to demonstrate that it was easier for the M-EPNet to separate both tasks during evolution. This behaviour was also observed when the data set *C1-D1* was tested in these conditions. Also, the best individual over all independent runs could classify correctly all patterns in the test set, showing that generalization performance is not lost with more modular architectures in this case.

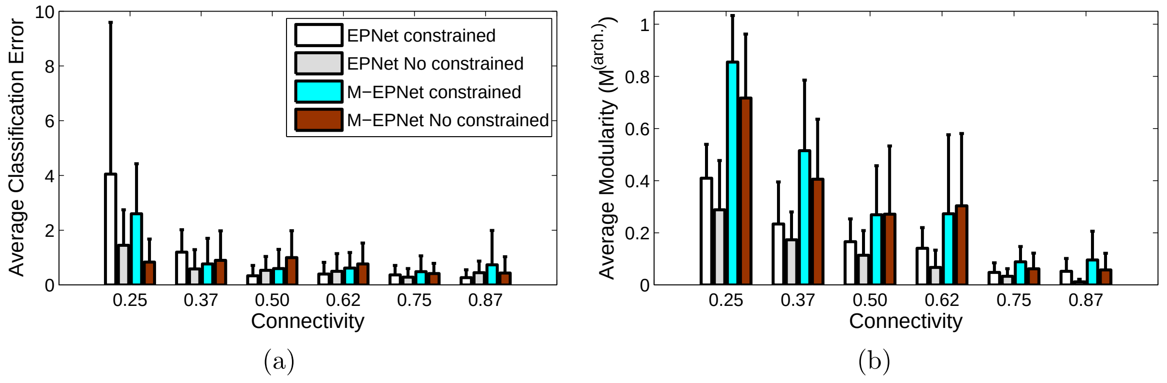


Figure 5.11: Average classification error and architectural modularity values for *A1-B1* data set with 4 inputs and 4 outputs at different connectivity values with the EPNet and M-EPNet and with constrained connections, or not, during evolution. The classification error is presented in Fig. 5.11a and the architectural modularity in Fig. 5.11b

5.5.1.2 Network sizes

It may be important to evaluate how different the evolved networks are when the data sets are evolved independently and when they are put together to evolve modules.

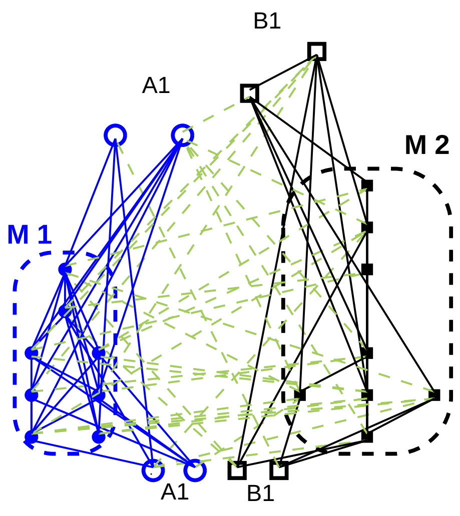
Data set *A1* obtained on average 13.9±2.74 hidden nodes with 54.03±17.03 connections, and data set *B1* 15.36±2.69 hidden nodes with 57.4±13.91 connections when they were evolved in Section 5.4. When they are combined (*A1-B1* with 4 inputs) figures of 14.86±3.29 hidden nodes and 72.80±19.85 connections were obtained, using the M-EPNet at 0.25 of connectivity in the initialization and no constraint during evolution. Those values are of moderate size considering that both tasks are solved at the same time. When 2 inputs are used in this combined data set, the average number of nodes and connections increased slightly, showing that it was more difficult to separate both tasks with this configuration. The same experiments was repeated for data set *C1-D1* and similar results were obtained.

Besides the comparison of average values, it is interesting to compare the best networks evolved when the data sets are combined (Fig. 5.12) and when they are evolved separately (Fig. 5.8). Fig. 5.12 shows the evolution of data set *A1-B1* with 2 and 4 inputs with the EPNet and M-EPNet, only in Fig. 5.12a it is indicated which input and output nodes belong to each module. There, clearly the M-EPNet algorithm reduces considerably the amount of shared connections, making the modules more easily distinguishable in the figures. Also it can be seen that similar structures, in terms of size, were obtained if they were compared with the evolution of the tasks separately in Fig. 5.8. Note that in Figs. 5.12a and 5.12d the M-EPNet algorithm correctly associates the inputs of task $A1$ with its module and the same is shown for task $B1$.

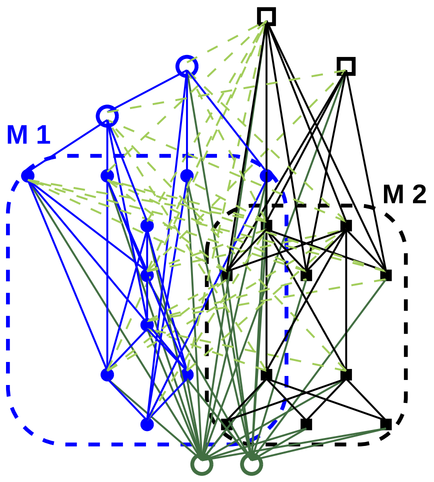
A more detailed analysis could be performed if one looked at the average number of nodes per module. As an example, Fig. 5.13 shows the number of nodes obtained with the M-EPNet algorithm at 0.25 connectivity in the initialization and no constraint during evolution. In this case the harder task *A1* represented by Module 1 in Fig. 5.13 required on average more hidden nodes than task *B1* (Module 2), and the same behaviour is present with the EPNet algorithm (not shown here).

5.5.1.3 Performance

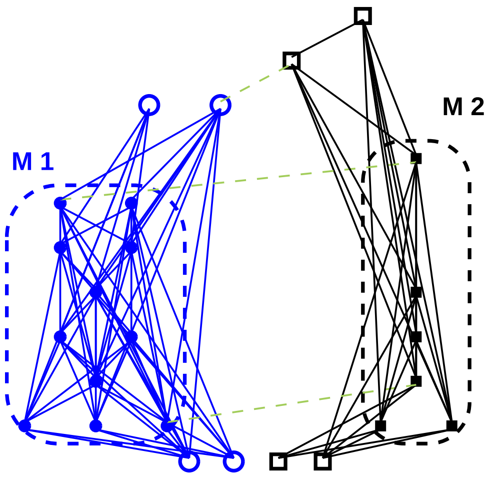
The separate evolution of task $A1$ and $B1$ required significantly smaller number of generations than when they are combined. That is because *A1-B1* data set is a bigger problem



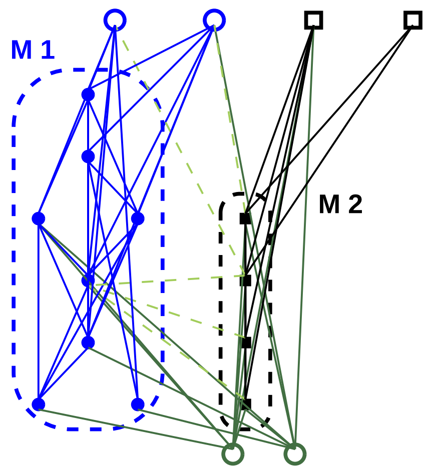
(a) EPNet



(b) EPNet



(c) M-EPNet



(d) M-EPNet

Figure 5.12: Best individual evolved for the *A1-B1* data sets with the EPNet and M-EPNet and with 2 and 4 inputs. *A1-B1* data set with the EPNet and 4 inputs (Fig. 5.12a) and with the M-EPNet and 4 inputs (Fig. 5.12c). *A1-B1* data set with the EPNet and 2 inputs (Fig. 5.12b) and with the M-EPNet and 2 inputs (Fig. 5.12d)

to solve, and it is expected to use more mutations than in the separate case. Nevertheless, remember that one of the advantages of evolving modules is their reuse, avoiding the evolution of the same task from scratch (Chapter 6).

The average values shown in Fig. 5.9 were obtained from the best individual of each independent run. Clearly not all of them correctly classify all patterns of the test set, but

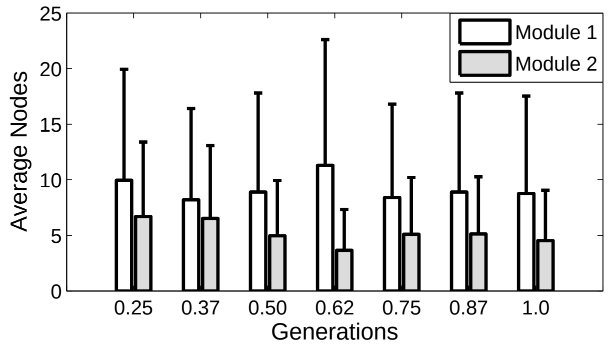


Figure 5.13: Average number of nodes per module for *A1-B1* data set with the M-EPNet algorithm at 0.25 percentage of connectivity in the random initialization and no constrain during evolution. Modules 1 and 2 represent data set *A1* and *B1* respectively

Table 5.2: *A1-B1* data set results with four input units and with the M-EPNet algorithm with 0.25 connectivity in the random initialization and no connection constrain during evolution

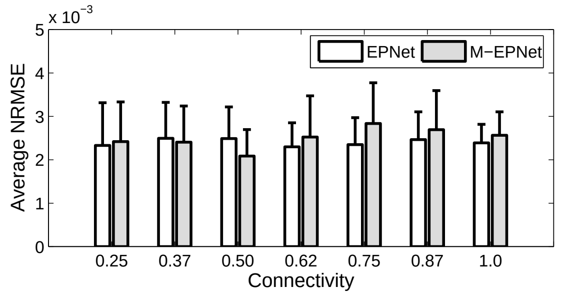
Parameter	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	4	0	4	4
Number of Hidden Nodes	15	3.298	9	21
Number of Connections	72.800	19.86	40	108
Error percentage Validation Set	0.964	0.400	0.380	1.699
Classification error Validation Set	1.033	0.472	0.500	2.000
Error percentage Test Set	0.815	0.406	0.197	1.782
Classification error Test Set	0.833	0.844	0	2.500

the best individual of the best independent run classifies all patterns correctly as previously stated and shown in Table 5.2.

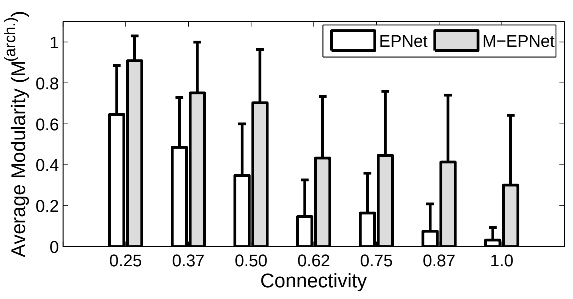
5.5.2 Two compound prediction tasks

It was shown above that connectivity constraint during evolution was not beneficial for the algorithms at low connectivity values, providing bigger generalization errors. Testing combined prediction tasks, showed the same behaviour, thus the constrained connections during evolution will no longer be considered in the remainder of this study.

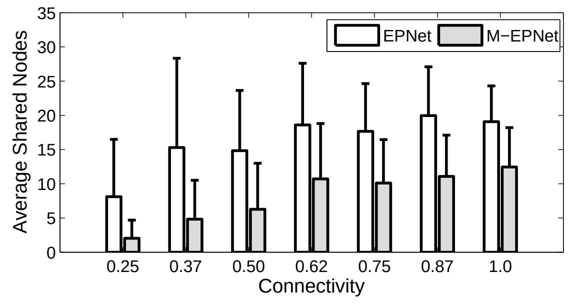
Fig. 5.14 shows the average parameters of the best network over 30 independent runs of the combined TS $Lo_A^{(ssp)}$-$Lo_{B1}^{(ssp)}$ (henceforth Lo_A-$Lo_{B1}^{(ssp)}$). Fig. 5.14a presents the average



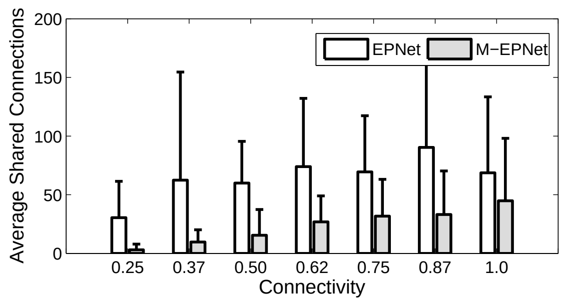
(a)



(b)



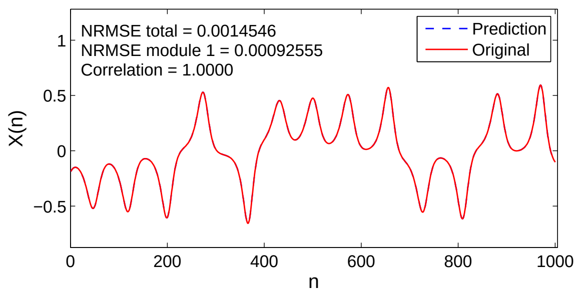
(c)



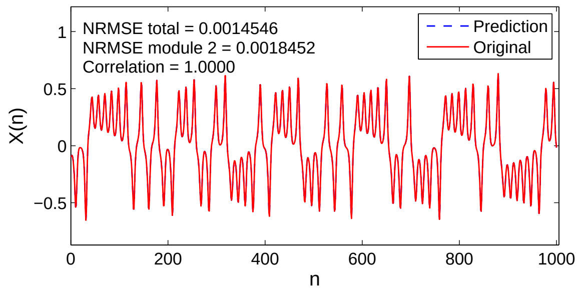
(d)

Figure 5.14: Average evolved values for the Lo_A-$Lo_{B1}^{(ssp)}$ TS with the EPNet and M-EPNet algorithms. Average NRMSE on the final test set (Fig. 5.14a), $M^{(arch.)}$ (Fig. 5.14c), shared nodes (Fig. 5.14b) and shared connections (Fig. 5.14d)

NRMSE in the final test set (generalization performance from both tasks) where it can be noticed that M-EPNet at 0.50 of connectivity obtained the smallest error. Fig. 5.14b shows the architectural modularity, where its increment is corroborated by the shared nodes and shared connections decrement, Figs. 5.14c and 5.14d respectively. Different to the *A1-B1* data set, the Lo_A-$Lo_{B1}^{(ssp)}$ TS produces bigger modularity values with the EPNet and M-EPNet. This combined TS was configured as $Lo_A^{(ssp)}$ TS, i.e. 1500 values to train and 1000 values to test. Fig. 5.15 shows the best prediction found over 30 independent runs for the M-EPNet algorithm at 0.25 connectivity for both modules. The NRMSE displayed in Fig. 5.15a was slightly bigger than the best error found when $Lo_A^{(ssp)}$ and $Lo_{B1}^{(ssp)}$ TS were evolved separately in the previous chapter: 4.86E-05 and 1.02E-03 respectively for the best found over 30 trials. In Figs. 5.15a and 5.15b it can be seen that the best error (of each module) is slightly bigger than when the tasks are solved separately. This may be due to the interference of both tasks during evolution, and it is quite probable that if more



(a)



(b)

Figure 5.15: Best prediction found in 30 independent run for both sub-tasks in Lo_A-$Lo_{B1}^{(ssp)}$ TS with the M-EPNet algorithm at 0.25 connectivity. Fig. 5.15a shows the first module of the task, represented by TS $Lo_A^{(ssp)}$ and Fig. 5.15b represents the second module, being $Lo_{B1}^{(ssp)}$ TS

generations were allowed the networks will reach similar error values in the combined TS. Note that $Lo_{B1}^{(ssp)}$ TS was evaluated with different sizes of data sets: 500 values to train and 500 to test. Nevertheless, the evolution of those combined TS allows accurate prediction where it cannot distinguish between predicted and real data as seen in Fig. 5.15. In all tasks tested in this chapter, it was noticed that usually there is one task more difficult to solve than the others, needing more hidden nodes and having bigger errors during evolution. For example, in Fig. 5.16 can be seen how the error per module (Fig. 5.16a) in Lo_A-$Lo_{B1}^{(ssp)}$ TS is decrement when the M-EPNet algorithm is used at 0.25 level of connectivity in the random initialization, and how the second module ($Lo_{B1}^{(ssp)}$) was more difficult to predict, requiring more hidden nodes than the first module (Fig. 5.16b).

Looking at the best network's architecture, Fig. 5.17 shows the best individual found for the Lo_A-$Lo_{B1}^{(ssp)}$ with the M-EPNet at 0.25 connectivity, where circle shapes represent the first module ($Lo_A^{(ssp)}$), and square shapes the second module ($Lo_{B1}^{(ssp)}$). There it can be seen that a pure modular architecture emerges, without consider shared connections between output nodes. Also note that the input nodes from both tasks were intercalated, i.e. input one from sub-task 1, input 2 from sub-task 2, input 3 from sub-task 1 and so on. That was done to test the harder case where the inputs of the same task are not together. The M-EPNet was able to find the correct partition for this combined TS in terms of hidden and input nodes. The network with the EPNet algorithm is not presented as it provided similar results as previously seen where several cross-connections are presented.

5.5.3 Same time series at different prediction lapses

Continuing with the evolution of two similar tasks, next was tested a single TS to predict different steps ahead at the same time, i.e. $Lo_A^{(ssp)}$ predicting $\Delta = 1$ and $\Delta = 5$. In that case, in terms of modularity values similar results were obtained as those of Lo_A-$Lo_{B1}^{(ssp)}$ (Fig. 5.14b). That demonstrates again that different connectivity values can solve this kind of TS with similar accuracy, and independent partitions appear when low connectivity values are used in the random initialization. The same experiments was carried out for the $MG30$ TS, obtaining similar results as the presented in this section, which let us assume that this behaviour will be maintained for other TS.

5.5.4 Discussion of evolving two similar tasks

In [12] it was shown that modular architectures emerge at low connectivity values for data set *A1-B1*, and when the fitness was evaluated differently. During this chapter, it has been shown the conventional evolution of those tasks where the fitness is in terms of the performance error, as commonly in EANNs. Whether 2 or 4 inputs are used, neural

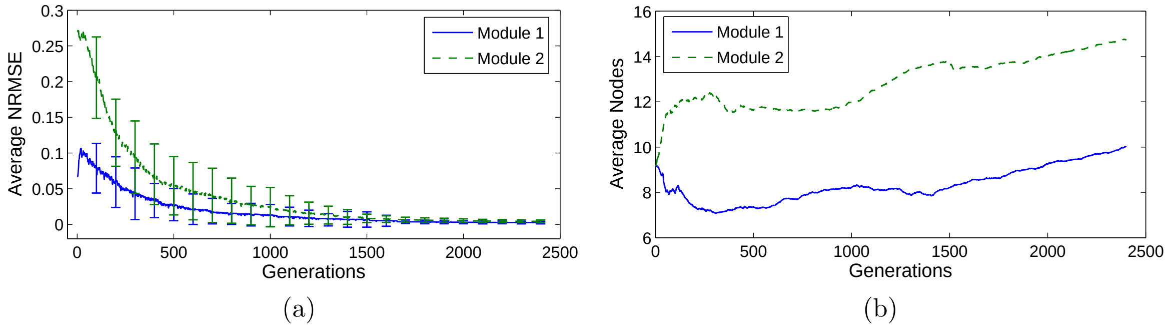


Figure 5.16: Average error and hidden nodes per module during 2400 generations for the Lo_A-$Lo_{B1}^{(ssp)}$ TS with the M-EPNet algorithm at 0.25 connectivity. Error is presented in Fig. 5.16a and hidden nodes in Fig. 5.16b

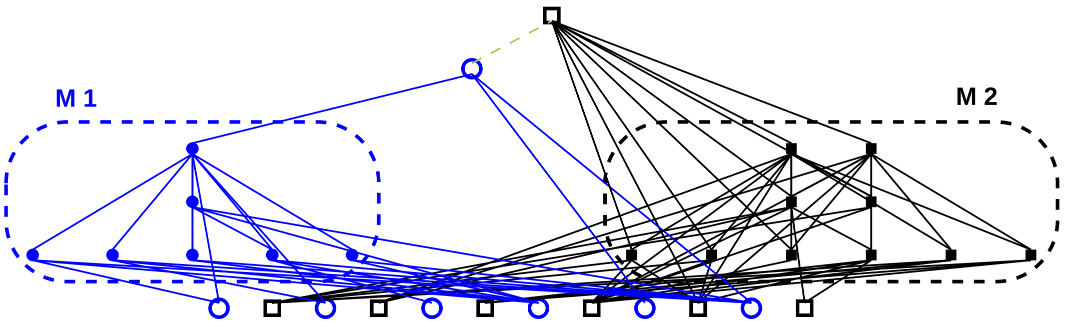


Figure 5.17: Best network found for the Lo_A-$Lo_{B1}^{(ssp)}$ TS with the M-EPNet algorithm at 0.25 connectivity. Where module 1 represents $Lo_A^{(ssp)}$ TS and module 2 $Lo_{B1}^{(ssp)}$ TS

networks with different degrees of modularity can solve equally well data sets *A1-B1* and *C1-D1*. Moreover, it has been demonstrated that the M-EPNet is able to obtain bigger modularity values without compromise the performance of the tasks.

When prediction tasks were introduced, the modularity increased in general, showing that these task presented more cross-talk interference, giving bigger modularity values in both algorithms. In any case, the M-EPNet algorithm may be the best option if bigger modularity is desired.

The prediction of the Lorenz and Mackey-Glass TS is presented in [100, 118], looking for modular architectures that could correctly separate (and predict) the compound task and associate each input to their partition it belongs, similar to Fig. 5.17. Nevertheless and as remarked in Section 2.4.3, in [100, 118] it is used a co-evolutionary algorithm with two containers, one for modules and one for systems, producing in all cases modular architectures (given the configuration of the method). That is completely different to the M-EPNet algorithm, because in this thesis any node could be connected to any partition of nodes, making more difficult to find independent modules that solve a particular sub-task, i.e. the aim here is to find in one evolutionary stage the modules and the interaction between them, while [100, 118] evolve modules and systems separately, requiring a credit assignment strategy.

Thus, it can be concluded that similar tasks as presented here can indeed evolve pure-modular architectures when the connectivity is constrained in the random initialization of the population. Before commenting on the cross-talk interference, the next section will provide more information about this aspect, using less similar tasks to create a compound data set.

5.6 Evolving less similar tasks

Since the use of similar tasks in the compound data set did not produce pure modular networks in all cases tested, the aim in this section is to investigate whether the use of less similar tasks could introduce more cross-talk interference leading to pure modular architectures when high connectivity levels are used.

In this section, classification tasks *Breast-D2* and *Breast-Thyroid* will be tested; and prediction tasks Lo_A-$MG17^{(ssp)}$ and Lo_A-$MG17^{(msp)}$ TS will be also evaluated. All of them have been configured in the same way as the experiments of the previous section, without fixing the number of generation to a given value. In all cases, the experiments were configured with the parameters of the first sub-task presented, e.g. for *Breast-D2*, the experiment was configured as stated in Section 3.2.1 for the *Breast cancer* data set. Note that using these kinds of tasks is expected to result in a high amount of cross-talk interference as the tasks do not have anything in common.

Looking at the results for the *Breast-D2* data set, it was noticed that similar results were obtained as those with *A1-B1*, where only lower connectivity values produce almost pure-modular architectures with the M-EPNet algorithm. These results suggest that the interference between two task could be easily absorbed by the large amount of connections if one of the combined tasks is easy to solve (e.g. $D2$), leading to non-modular architectures. The previous assumption was corroborated when the *Breast-Thyroid*, Lo_A-$MG17^{(ssp)}$ and Lo_A-$MG17^{(msp)}$ task were tested, producing bigger modularity values. This demonstrates that, indeed, the more different two tasks are, the more cross-talk interference in the networks and the bigger modularity obtained. Nevertheless, non-pure modular architecture are still found on average at connectivity values from 0.62 to 1.0, suggesting again that both algorithms (EPNet and M-EPNet) cannot delete the exceeding amount of connections because they seem to absorb the interference each inflicts on the other with the learning algorithm. The previous experiments allowed the evolution to finish when the error was no further improved or when all patterns of the validation set were correctly classified; the effect of that was explored by testing again the *Breast-Thyroid* data set with fixed number of generations (3000, being bigger than the maximum used in previous experiments for this task). As was expected, the modularity was increased slightly, without statistically significance, over the previous case. For example, the final architectural modularity values at 1.0 connectivity was $M^{(arch.)} = 0.1926 \pm 0.22$, which is bigger than the previous cases, but still lower to deliver pure-modular architectures.

It can be concluded that only modular architectures are found at evolving two different tasks simultaneously, if the networks are initialized with lower connectivity values before the evolution starts. If high connectivity values are used, the tasks can be solve with a similar performance, but in a lower number of generations and lower modularity values ($M^{(arch.)}$), which indicate that the large number of connections in the networks can absorb the high cross-talk interference expected for these tasks. It was also demonstrated that MSP tasks can be performed with a modular algorithm, i.e. use a compound data set to evolve modules. Similar results may be expected if more similar TS are tested with the MSP method.

5.7 Evolving three tasks simultaneously

Even though evolving more than 2 task may be more challenging for the algorithms, here this scenario will be investigated by evolving 3 tasks simultaneously, expecting that more tasks will introduce more interference and that will result in bigger modularity values.

Fig. 5.18 shows the average parameters evolved for the *A1-B1-C2* data set. Clearly, the modularity values were reduced in comparison to when two tasks were used, but it is important to note that the M-EPNet algorithm obtained the smallest classification error on average at 0.5 connectivity.

Moreover, even when the architectural modularity is low, it can be seen that the weighted modularity is bigger in all cases with the EPNet and M-EPNet algorithms (Figs. 5.18e and 5.18f), which (as remarked by [17]) is a signal that the algorithm is able to deal with the interferences between tasks. Even though this aspect was not discussed for the previous experiments, the same behavior was noticed in all experiments concerning 2 or more compound tasks. It is worth commenting that the bigger the connectivity used, the more generations required to stop the algorithm (Fig. 5.18b), which is a completely different behaviour to that seen with two compound tasks, and mainly due to the algorithms spending more time dealing with the cross-talk interference.

Fig. 5.19 shows the best network evolved for *A1-B1-C2* with the M-EPNet algorithm at 0.25 connectivity. As previously noted, here it was more difficult to reduce the number of shared connections and that is reflected in Fig. 5.19 where several cross-connections are presented. Nevertheless, the network was able to correctly classify all patterns of the final test set. Also it may be noted that all inputs where correctly assigned into the module they should belong, as the first 2 inputs from left to right belongs to $A1$, the next 2 to $B1$ and the last 2 inputs to $C2$ sub-task (and the same applies for the output nodes).

A1-B1-C3 data set was also tested, producing similar results as those obtained for *A1-B1-C2* data set. Clearly, evolving more than 2 sub-tasks simultaneously produce networks with lower modularity values, which may be directly related to the fact that there are more parameters to evolve and the overlap between classes in each sub-task.

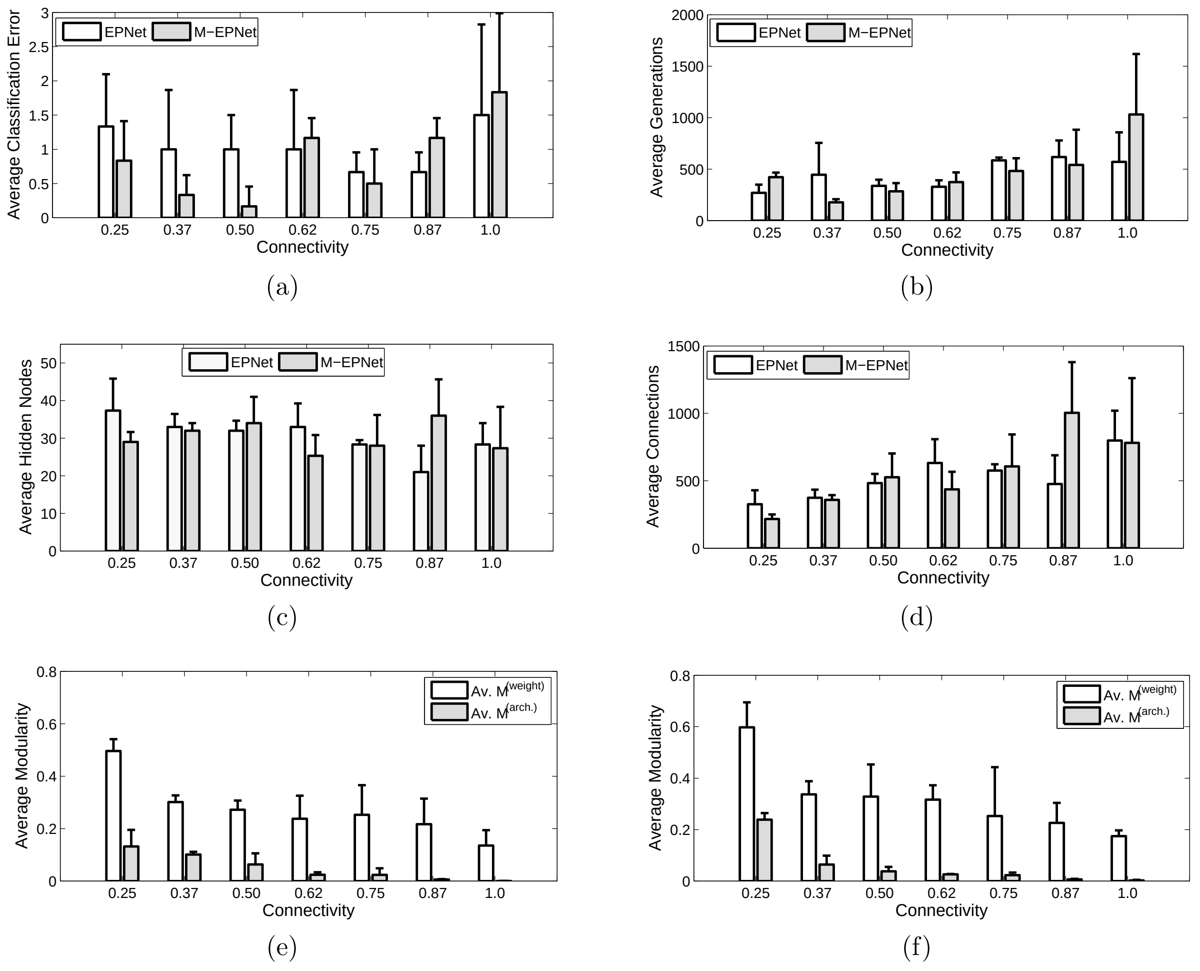


Figure 5.18: Evolved parameter values for *A1-B1-C2* data set with 2 inputs and 2 outputs per task at different connectivity values with the EPNet and M-EPNet. Generalization performance is shown in Fig. 5.18a, generations in Fig. 5.9b, hidden nodes and connections are shown in Figs. 5.18c and 5.18d respectively. $M^{(arch.)}$ and $M^{(weight)}$ with the EPNet algorithm in Fig. 5.18e and with the M-EPNet in Fig. 5.18f

5.8 Discussion

This chapter has explored how the EPNet algorithm can be extended to produce more modular networks. In terms of novel contributions, it has been shown how a modularity measure can be used to identify specific neural components, which lead to the rearrangement of nodes in the graphical representation of networks and the introduction of new mutation operators, thus increasing the degree of modularity arising through evolution. It is worth pointing out that even a different modularity measure could be used for the same purpose as presented here, that could result in a different behaviour of the EA. Also note,

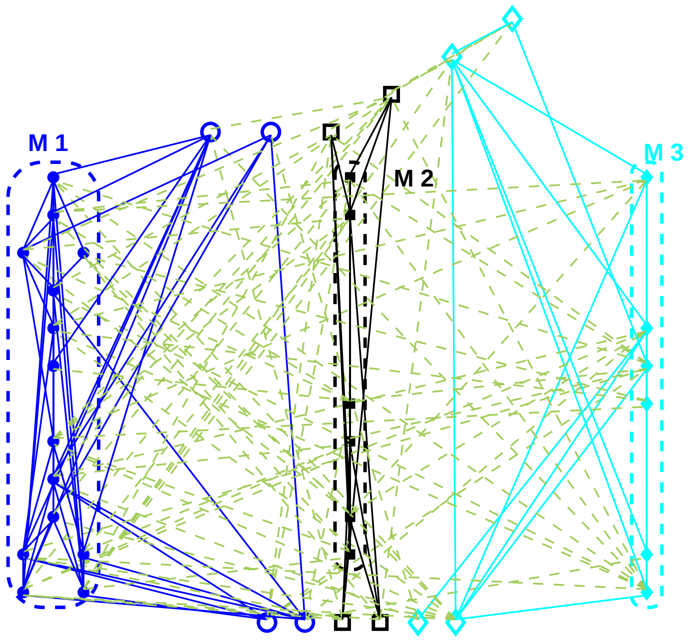


Figure 5.19: Best network evolved for *A1-B1-C2* data set with the M-EPNet algorithm at 0.25 percentage of connectivity. Modules 1, 2 and 3 represent tasks *A1*, *B1* and *C2* respectively

that the aim is to increase modularity because that will lead to more independent modules, and that may facilitate the reuse of them in subsequent implementations, i.e. in homogeneous networks there are a strong coupling among nodes because of the large number of connections expected, which may be translated to a harder task of module reuse.

The same modularity measure was also improved to work for networks with a single output unit, and to allow a finer-grained analysis of the internal partitions of nodes inside a network. Although this approach has assumed prior knowledge about the task data partitions to measure the modularity, there is scope for dynamically adjusting the output partition into modules for networks with more than one output unit.

The information provided by the modularity measure allowed the implementation of a new Modular EPNet algorithm (M-EPNet) that biases the evolution of modularity compared to the classical EPNet algorithm. Comparing the results of this chapter with a derived work of this study [161] showed that using just the shared node and shared connection deletion is not as beneficial as using them with the intra-module connection addition to increase the modularity in the networks. Moreover, the EPNet and M-EPNet algorithms ensure that networks will not add neural components without limit, which means that no extra constraints are required to maintain the networks with moderate size.

Whether or not 2 tasks are similar, they only evolve pure modular architectures if lower connectivity values are used. When 3 task are evolved simultaneously, there are more parameters to evolve and lower modularity values are found in general with lower connectivity values. However, the interference is more clearly presented with three tasks, and more generations were required to reach similar performances values. It was also discovered that tasks requiring the MSP method can be performed with evolved modular architectures, which means that more difficult task can be tackle with the M-EPNet algorithm.

The general conclusion is that the rearrangement of nodes into modules using the modularity measure enables the discovery of shared neural components through modules, and that information can be used to evolve almost pure modular neural networks with the M-EPNet algorithm at lower connectivity values without compromising the performance of each individual sub-task. The new mutation operators are not special to the M-EPNet algorithm - they can easily be applied to bias towards modularity in other evolutionary neural network approaches. Moreover, they are not restricted in any way to the test tasks chosen for this chapter, but can be applied to any classification, prediction or regression task.

What remains to be done is to test to what extend module reuse is facilitated with neural networks using the M-EPNet algorithm. This matter will be explored in the next chapter.

Chapter 6

Module reuse

A practical problem for real world applications is that the advantages of modularity (particularly module reuse) will only become apparent (and hence result in a tendency for modularity to emerge through evolution) if sufficiently extensive patterns of successful module reuse occur in the evolutionary simulations. In practice, most simulations are far too small-scale for such modularity and its advantages for module reuse to occur. For that reason, Chapter 5 explored how to help modularity (and its advantages emerge) with the M-EPNet algorithm, without the need for such computationally infeasible evolutionary simulations.

Considering the fact that no previous study has demonstrated empirically that module reuse is possible (e.g. modular NEAT [15]), this chapter provides a range of experimental configurations to demonstrate module reuse. The experiments of this chapter will be divided into two main stages: 1) two combined tasks will be evolved for module discovery as done in Chapter 5; 2) thereafter, the arrival of a new sub-task to the network will be simulated, where it can be the same (Section 6.2) or similar (Section 6.3) to the one previously evolved; with the aim to reuse a module or neural resources from previously evolved modules. Since it will be beneficial to have a way to quantify the degree of module reuse, Section 6.1 introduces a measure that could be implemented (using the dependency measure from equation 5.3) to quantify the number of nodes reused to solve the current task.

6.1 Module reuse metric

Since no previous work has been found to quantify module reuse, here will be introduced a metric to calculate the nodes reused from other modules to solve the given task. In a further work, the same idea can be implemented to measure the number of modules reused to solve a bigger task.

To quantify this, one can simply count the number of nodes from other modules connected directly or indirectly to a given module k. To avoid complicated algorithms for performing that, the dependency $d_i(j)$ (equation 5.3) can be used from the weighted modularity $M^{(weight)}$, i.e. counting all occurrences where $d_i(j) > 0$ and $j \neq k$.

This measure is less computationally expensive than using a conventional algorithm for the same purpose (e.g. a recursive algorithm), and that is achieved by taking advantage of the dependency values already calculated. However, it may give misleading values if the EPNet algorithm is used, because of the high number of cross-connections between modules.

6.2 Module reuse: same task

As remarked in Chapter 2, no study has demonstrated empirically that effective module reuse is possible in ANNs. Hence, the aim of this section is to investigate whether a module previously evolved, or internal nodes from it, can be reused.

First the M-EPNet algorithm will be used to find modules for the *A1-B1* data set. It will be assumed for the rest of this chapter that low connectivity values will be used (0.25) to initialize the network before the evolution starts, because this provides bigger modularity values as shown in Chapter 5. After the M-EPNet has finished, all individuals in the population will be saved, and sub-task $A1$ will be introduced again, giving an *A1-B1-A1* data set. Where the two inputs and outputs of the first $A1$ sub-task (from left to right) are the same as the last $A1$ sub-task, i.e. the patterns representing $A1$ are the same for both $A1$ sub-tasks (at the end of this section will be presented the case in which the patterns of the last $A1$ sub-task are randomly reorganized to test module reuse). Thereafter, all sub-tasks will be allowed to continue their evolution, possibly reusing the previous modules evolved for *A1-B1* data set. With this configuration, it will be expected that the module evolved for the $A1$ sub-task in the first stage will be reused to solve the second $A1$ sub-task in the second stage.

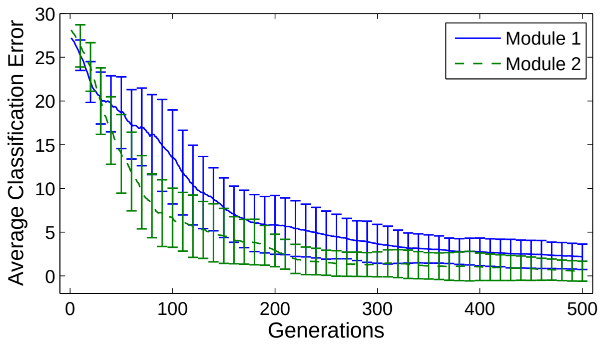
Note that, alternative configurations are possible, e.g. fixing previous modules and just evolving the new one introduced. However, for the purpose of this chapter, it will be sufficient to leave all parameters to evolve.

Fig. 6.1 shows the average classification error per module, modularity values and best network found for the *A1-B1* data set in the first stage of the experiment. Where, the first 2 inputs and outputs from left to right belong to module 1 ($A1$) and the others inputs and outputs correspond to the second module ($B1$). Note that all inputs have been correctly assigned in the module they should belong and just one cross-connection remains there, as found in the previous chapter for the same data set (Fig. 5.12c).

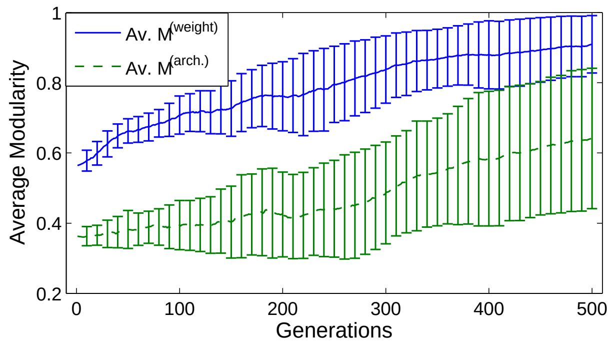
When the EPNet and M-EPNet are compared, as remarked in Chapter 5, there is no statistical significance during all the evolution if consider the average classification error of the population (Fig. 6.2a), nevertheless, the $M^{(arch.)}$ between both algorithms (Fig. 6.2b) shows statistical significance at the end of the evolutionary process.

In the second stage of the experiment, $A1$ sub-task is introduced at the end of the previous data set, leading to an *A1-B1-A1* data set. The evolved classification error, number of nodes reused and best network found for the module reuse is presented in Fig. 6.3.

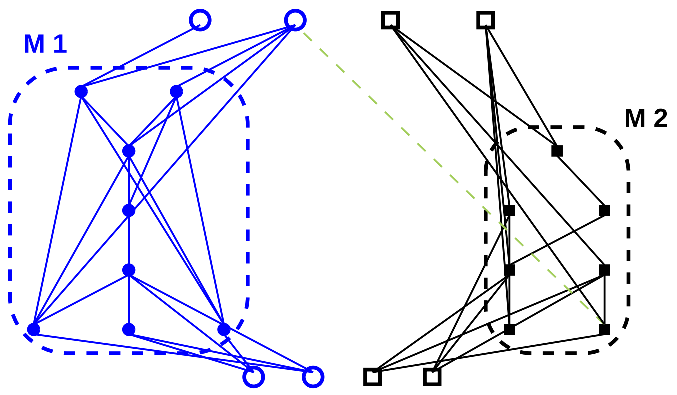
As can be seen in Fig. 6.3a, the error of the third module (new $A1$ sub-task, nodes with diamond shape) in the early generations is higher, because at the beginning there were no neural resources assigned to it. But as the generations advance, the last module starts to reuse nodes from other modules (Fig. 6.3b) while the average classification error of the population is decreased (not presented here), as well the average error of the new task (Fig. 6.3a). At the end of evolution, it can be seen in Fig. 6.3c the cross-connections from module



(a)

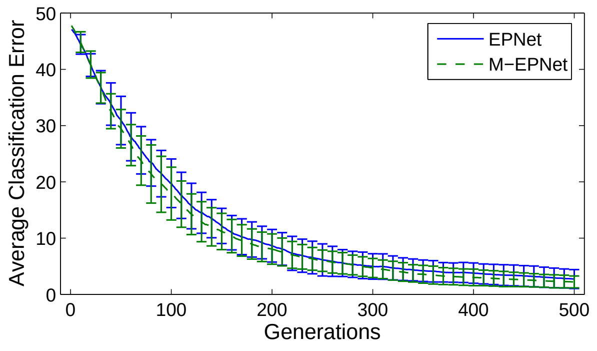


(b)

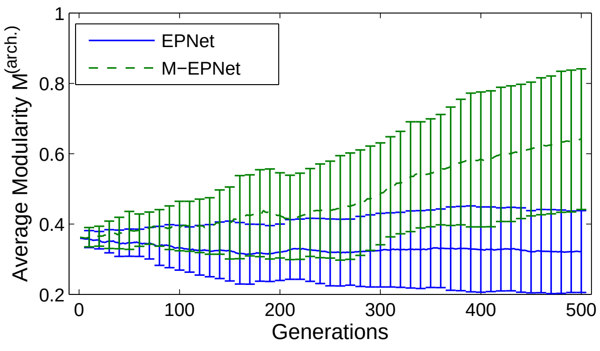


(c)

Figure 6.1: *A1-B1* data set evolved with the M-EPNet algorithm for stage 1. Module 1 ($M1$) corresponds to $A1$ sub-task and module 2 to $B1$ sub-task. Average classification error on the validation set is shown in Fig. 6.1a, modularity values in Fig. 6.1b and best network found in Fig. 6.1c

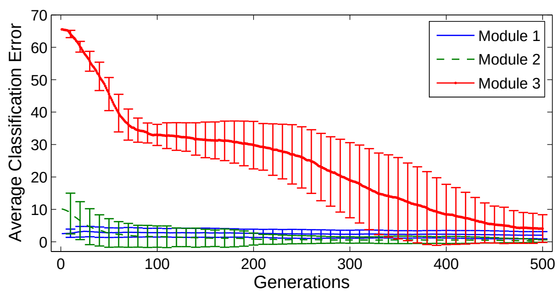


(a)

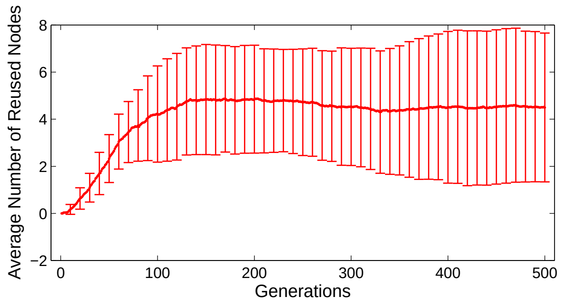


(b)

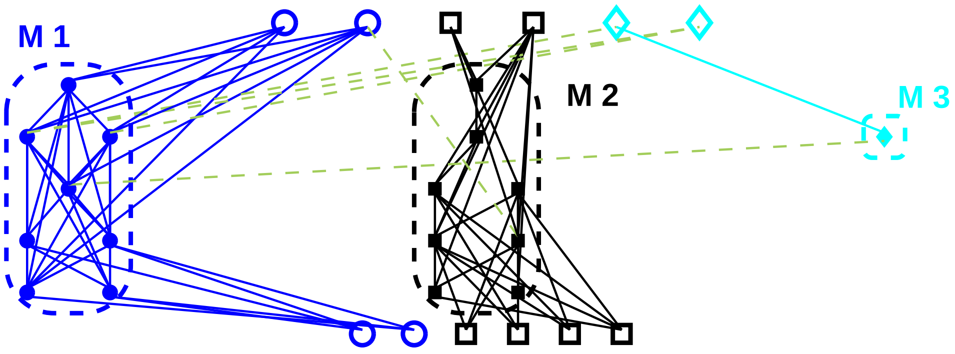
Figure 6.2: *A1-B1* data set evolved with the EPNet and M-EPNet algorithm for stage 1. Fig. 6.2a shows the average classification error considering both tasks, while Fig. 6.2b presents the average modularity $M^{(arch.)}$



(a)



(b)



(c)

Figure 6.3: Module reuse for *A1-B1-A1* data set with the M-EPNet algorithm. Modules 1, 2 and 3 correspond to sub-tasks $A1$, $B1$ and $A1$ respectively, where the first two were previously evolved in stage 1 of the experiment and allowed to continue evolving here. Average classification error on the validation set is shown in Fig. 6.3a, average number of reused nodes from module 3 in Fig. 6.3b and best network found in Fig. 6.3c

1 and module 3, suggesting that nodes reused in Fig. 6.3b belong to module 1. The new incoming task from the best evolved network, reused almost all pure nodes from module 1 (7 nodes) as seen in Fig. 6.3c. Also note that module 1 maintains the same number of nodes but higher intra-modular connections and module 2 increase the number of nodes and number of intra-modular connections.

Nevertheless, the average error was not improved nor worsened as seen in Fig. 6.3a for the first two modules. On the other hand, inputs of module 3 were not correctly assigned to module 1 nor 3, and they were connected to module 2. That was because the first 2 inputs carry the same information as the last 2, thus, outputs of sub-task 3 take the information provided by the first 2 inputs.

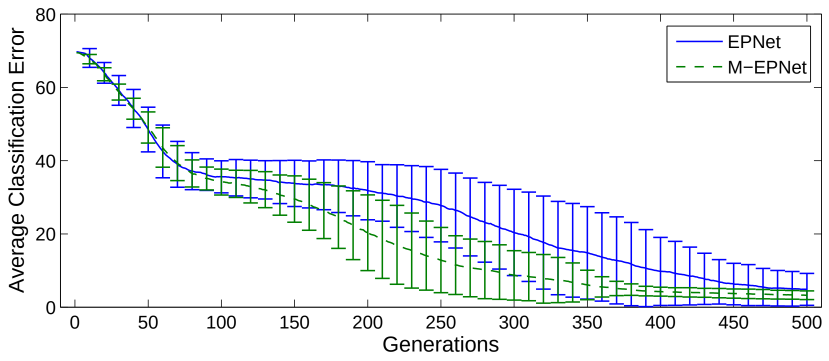


Figure 6.4: Module reuse for *A1-B1-A1* data set with the EPNet and M-EPNet algorithms for the average classification error. The three sub-tasks involved are considering in the error of each algorithm

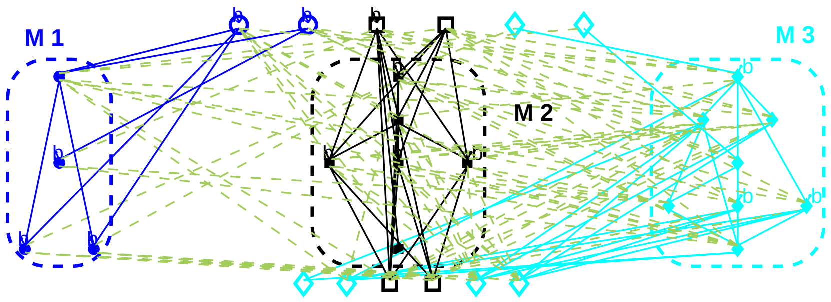


Figure 6.5: Best network obtained for module reuse for *A1-B1-A1* data set with the EPNet algorithm

Similar as stage one, Fig. 6.4 shows the EPNet and M-EPNet algorithms for the average classification error from the three sub-tasks, i.e. the general classification error from each network. Even there is no statistically significance at the end of the evolution between both algorithms (as in stage one), it can be seen that the modular algorithm was beneficial to reduce more the error around the generation 100, being the same interval where the modular algorithm started to reuse more neurons from other modules (Fig. 6.3b).

Even both algorithms have a similar performance at the end of the second stage of the experiment, it can be seen in Fig. 6.5 that the best network evolved for the EPNet algorithm has several cross-connections between modules. Moreover, it presents a bigger number of connections than the required for the best network found with the M-EPNet (Fig. 6.3c).

When the new $A1$ data set is randomly reorganized (or new patterns generated), a new module was observed to emerge during evolution for sub-task 3, where modules 1 and 3 did not communicate with each other (no module reuse), even they solve the same sub-task. That is because the first 2 inputs represent one class of the $A1$ sub-task, but at the same time, the last two inputs could represent another classification boundary of $A1$, and a single module is not able to solve this at the same time: two different input vectors for two different target vectors, because the module was designed to solve one input vector (the first 2 inputs) for one output vector (the first 2 outputs) in the modular architecture. Therefore, the only way to deal with such differences was evolving new hidden nodes (module 3) for the last sub-task.

In this experiment, the second stage starts without any neural resource assigned to the last module, demonstrating that module reuse is possible because the last module started to evolve connections to other modules to reduce its error, and the overall error of the networks. The same experiment was repeated evolving all three tasks at the same time from scratch to see whether a single module emerge for sub-tasks 1 and 3. Different to the experiment of Section 5.7, here the last sub-tasks (last 2 inputs and outputs) were not connected to any other node in the network (in the random initialization) to simulate the same scenario as when the module were reused in Fig. 6.3.

The results showed that both sub-tasks (1 and 3 for $A1$) developed nodes and shared connections between them, indicating that both tasks have nodes in common and they were bounded through different cross-connections. In this case, for the best network found, module 1 reused 2 nodes from module 3, and module 3 reused 12 nodes from module 1.

Note that $M^{(arch.)}$ and $M^{(weight)}$ have not been discussed for module reuse (second stage) as they will provide lower modularity values than stage one, because of the expected shared connections between modules 1 and 3.

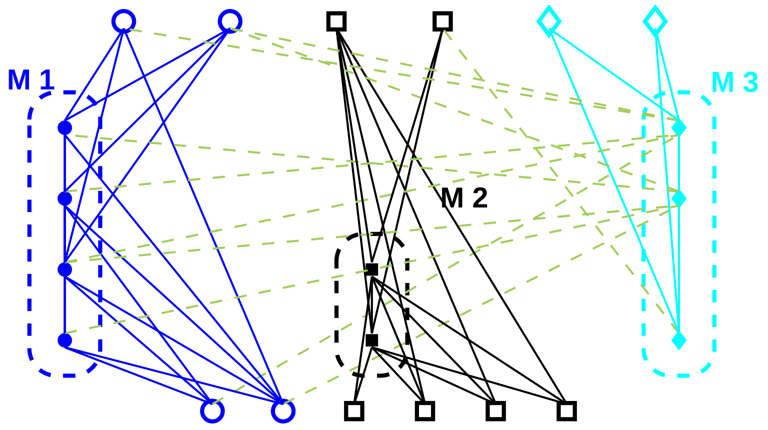


Figure 6.6: Best network found for module reuse for *A1-B1-A1* data set with the M-EPNet algorithm using stopping criteria during evolution

6.2.1 Evolving populations with stopping criteria

The same experiments were repeated allowing the algorithms to stop at any moment using any of the stopping criteria previously employed, and similar results were obtained as shown in Fig. 6.6, where more shared connections between module 1 and 3 were found. Furthermore, there were fewer nodes in modules 1 and 2, but module 3 evolved 3 hidden nodes for its own sub-task. In terms of number of generations, the experiments required on average fewer generations than in the previous sections, i.e. the algorithm finished on average before 500 generations of evolution.

6.2.2 Allowing connections between output nodes

As we do not know how modules are interconnected in the brain, nor how they are internally organized, it is difficult to have a precise experiment simulating exactly the same conditions that could help to improve our algorithms. In the previous two sections connections in the OO sub-matrix of the connectivity matrix were not allowed. Therefore, this section will investigate how module reuse is carried out when connections between output nodes are allowed. Hence, the same experiments of previous sections were repeated allowing

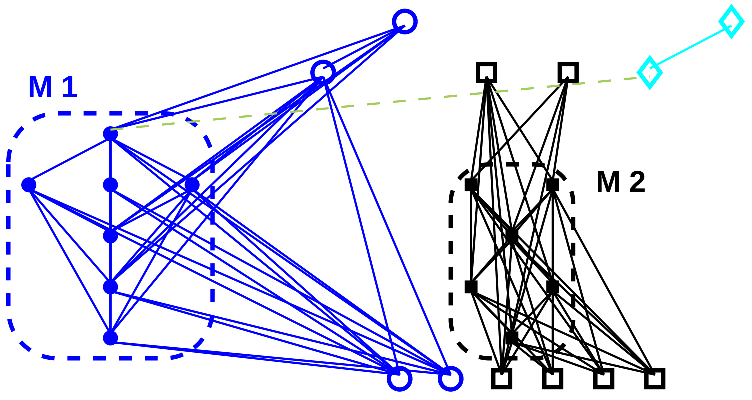


Figure 6.7: Best network found for module reuse for *A1-B1-A1* data set with the M-EPNet algorithm using stopping criteria during evolution and allowing connections in the OO sub-matrix

connections anywhere in the connectivity matrix. In Fig. 6.7 is presented the best network evolved when output to output connections are allowed. Interestingly, only one connection communicates the last hidden node from module 1 and the fist output node from the third sub-task, giving the required information to solve the new $A1$ sub-task.

6.3 Module reuse with similar tasks

In this case, the aim is to investigate whether module reuse could appear if the third task introduced is different but similar to the first one. To deliver that, the *A1-B1-A2* data set was tested with the same configuration previously used, i.e. using both stages. The results show that during evolution of the second stage (module reuse), a module was evolved for sub-task 3 (pure nodes of sub-task 3), but different cross-connections between module 1 and 3 where found. That demonstrates that both modules communicate because both classification tasks are similar in the geometrical shape of their boundaries, but different as they are in different position in the input space. This leads to the investigation of other different data sets, to test whether similar behaviours are obtained. Hence, *C1-C2-D1* and *C1-D1-C2* were tested in a similar way to the previous experiments, expecting the reuse of

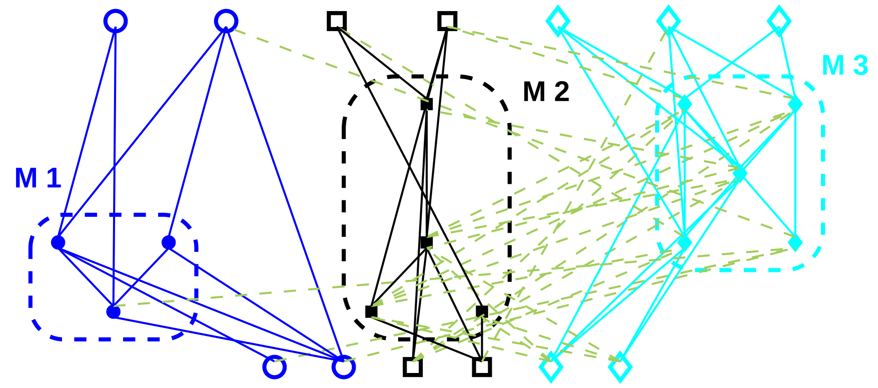


Figure 6.8: Best network found for module reuse for *C1-C2-D1* data set with the M-EPNet algorithm using stopping criteria during evolution

modules (or sub-neural structures) between sub-tasks 2 and 3, or new modules emerging for the incoming task with some shared connection between nodes from module 2. Similar to *A1-B1-A2*, on average *C1-C2-D1* and *C1-D1-C2* data sets evolved new hidden nodes for the new sub-task, but they were also interconnected through shared connections with the second module as expected. For example, Fig. 6.8 shows the best network found for the *C1-C2-D1* data set. Note that these results were obtained when the stopping criteria were used in the algorithms. When the experiments were repeated with fixed number of generations, it was found that the shared connections between the $D1$ and $C2$ sub-tasks reduce significantly, as the M-EPNet algorithm gives more priority to delete them. This again supports the observation that fixing the number of generations may lead to different evolved networks.

6.3.1 Constraining the number of nodes

As it is known that the brain does not have unlimited resources, the experiments with the *C1-C2-D1* and *C1-D1-C2* data sets were repeated fixing the number of hidden nodes in the second stage of the process, forcing the reuse of existing nodes from the second module into the third one. Thus, the aim is to discover if specific parts of the second module could be reused in the last module without increasing the average size of the networks (in terms of hidden nodes). Interestingly, in this new experiment it was found that pure nodes from the second module changed to the third one (spreading the nodes between modules) and more connections appeared (cross-connections and intra-modular connections), reducing the error of the last module and maintaining the performance of the first two. The behavior obtained by *C1-C2-D1* and *C1-D1-C2* is not maintained for all other classification tasks, because when the *A1-B1-A1* data set was tested in the same way (fixing the nodes in the second stage), sub-task 3 ($A1$) reused almost all nodes of sub-task 1 in a similar way as in Section 6.2. That seems to indicate that the arrangement of the classification tasks in the two dimensional input space plus the overlapping between them lead to complex relationships, but most importantly, some insights of module reuse (or substructures reuse) can be achieved.

6.4 Discussion

In this chapter, module reuse was proved to be possible through the evolution of neural networks with the M-EPNet algorithm. Also, the effect of reusing the same or similar tasks has been explored, showing that some neural structures are reused to solve the new incoming task, as no independent partition of nodes emerge for it. A simple metric was also introduced that helps to quantify node reuse from previous modules. That was based on the dependency measurement from the modularity measure, avoiding unnecessarily complex algorithms (e.g. recursive functions).

If in this chapter module communication was not allowed as in previous studies (and connections between output nodes), the networks will inevitably increase their average size (nodes and connections) because the last task introduced will evolve its own neural resources to improve its performance. In [100] is used a similar method where an incremental task is evolved, e.g. during 1000 generations is solved the first task, later it is introduced a compound task containing the first one and continuing the evolution, thus, reusing the previous structure evolved, similar as done in this chapter. The main difference between [100] and this study, is the fact that each module can interact with any other node of the network (more natural way to evolve modules), it was shown how neural reuse is increased during evolution and it was presented evidence of the best modular architectures found. Another discovery was the similarity in performance between the EPNet and M-EPNet algorithms for module evolution and reuse, with the difference that the modular algorithm increases the modularity considerably than with EPNet algorithm, i.e. homogeneous networks can absorb the cross-talk interference produced by similar and different tasks. However, if one considers bigger implementations, the M-EPNet algorithm should be the best option to evolve more independent modules, making more easy the module reuse than monolithic ANNs. Therefore, as a general conclusion it can be said that neural reuse proved to be possible and advantageous in this area using the M-EPNet algorithm. The next step may be an automatic method that reuses the same module in different contexts, like the structural and functional organization of the brain.

Chapter 7

Conclusions and future work

In this thesis, the EPNet algorithm was extended to deliver more modular networks (Chapter 5) for prediction and classifications tasks, and experiments were performed showing empirically that module reuse is possible (Chapter 6). The key contributions, in the order they appear in this thesis, are:

- Feature input evolution extension of the EPNet algorithm performed with single mutation operators. It was also shown that the False Nearest Neighbour and Average Mutual Information may not provide the best possible values to evolve networks for time series forecasting.

- Modularity measure improvement for networks with a single output unit, i.e. the modularity measure was extended and applied for networks containing a single output unit, like prediction tasks. It was proved that this modification fits in the actual definition of Ensembles and MNNs, and could be used for a fine-grain modularity analysis.

- Clearer graphical representation and analysis of modules provided by node reorganization, using the dependency value from the modularity measure.

- Introduction of the M-EPNet algorithm supported by a modularity measure, which borrows the basis of the EPNet algorithm and ideas that mimic some constraint aspects of the brain. The M-EPNet may have a similar or better performance in comparison with the EPNet, and it could increase significantly the average modularity of the evolved population. Moreover, homogeneous networks may absorb the cross-talk interference, i.e. homogeneous and MNNs could have a similar performance for some tasks.

- Module reuse proved to be possible through a simple experimental set-up. Also, evidence was provided showing which neural substructures were reused for a new incoming task, something missing in previous studies. In this context, the M-EPNet algorithm was capable to deliver bigger modularity values which helped to reuse previous modules evolved.

Even after these contributions to the knowledge, there remains several avenues to improve different aspects of the computational techniques. The following sections present the final conclusions (Section 7.1) and further research paths (Section 7.2) of this study.

7.1 Conclusions

In this thesis, more suitable modular architectures have been evolved with the M-EPNet algorithm in terms of having fewer cross-connections between modules and bigger intra-modular connectivity, which was helpful to demonstrate empirically that module reuse is possible in different scenarios.

To deliver this, firstly, the basis for its development has been described through the EPNet algorithm for finding suitable networks for prediction and classification tasks. Furthermore, it was shown that an extra data set was required for MSP tasks, as the fitness of the predictions needs to be evaluated in the same terms as the final performance evaluation when the evolution finished. Also, an analysis of the various parameters used in the EPNet algorithm to evolve ANNs was presented. The algorithm was found to not be as sensitive to variations of some parameters, like the population size or initial learning rate, as others, in particular the *successful training* parameter. It was shown how important it is to set this with an appropriate value.

Thereafter, Feature Selection EPNet (FS-EPNet) and Feature Selection with asymmetric delays EPNet (FS$_{AD}$-EPNet) algorithms were introduced to tackle input feature selection during evolution, where they were mainly implemented for predictions and classification tasks respectively. It was shown that calculating the Average Mutual Information for the time delay and False Nearest Neighbour for the embedded dimension was not the best option for all TS. Moreover, these method may give different values if the amount of information in the data set changes, something common in real world scenarios, where more information from the phenomenon is accumulated through time.

In classification tasks, the *Thyroid* and *Breast cancer* data sets used fewer inputs with the FS$_{AD}$-EPNet algorithm, however, the latter only achieved lower number of inputs through evolution, if the generations were fixed at some large value. This is an important point to be considered in real world applications, i.e. whether to terminate the evolution when the error shows no improvement, or when the number of features have been minimized. Thus, the main experiments presented in this study were performed evolving the input feature selection, where existing operators to evolve the architectures have been used to evolve the inputs, avoiding complex implementations, by only adding the delay mutation operator. The results of the feature evolution and fixed case against previous results in the literature showed a robust performance by EPNet, which may be important, as mutations are only used to carry out the evolution of networks with a steady-state algorithm that uses Lamarckian inheritance.

After understanding and improving the EPNet algorithm for feature selection, it was extended using a modularity measure to reorganize nodes into modules (allowing a clearer graphical representation of networks), and to introduce new mutation operators that simulate constraints in the brain (shared node and shared connections deletion, and intra-modular connection addition), which led to the M-EPNet algorithm. However, even a different modularity measure could be used for the same purpose as in this study, that could result in a different behaviour of the EA, and that should bear in mind when another modularity measure is adopted.

The introduction of the M-EPNet algorithm allowed a more natural way to evolve modules than previous studies, where all the networks' parameters were allowed to evolve simultaneously without restrictions like in previous studies, e.g. evolving only the hidden nodes [12], or forcing pure-modular architectures [13, 118]. The modular architectures resulting from the M-EPNet did not present significantly worse results than those evolved with the standard EPNet algorithm. However, the M-EPNet produces significantly larger modularity values when the networks were initialized with low connectivity values before the evolution starts.

Even though larger modularity values could be obtained with the M-EPNet algorithm, it was found that initializing the population with full connectivity, the networks required fewer generations to solve the tasks. This indicates that one should bear in mind the final application of the evolved networks, i.e. to investigate related aspects of modularity in the brain, or real world applications.

Moreover, one should consider that having full connectivity could be more computational expensive for the learning algorithm, and if we are dealing with finite resources (e.g. the battery and memory in a robot), the M-EPNet algorithm may be the best option to deliver significantly lower number of connections, with the advantage to discover and reuse modules (compact representations are expected), and having similar performance to fully connected networks.

Overall it was noted that cross-talk interference indeed leads to increased modularity, i.e. the more different two tasks the bigger the modularity obtained, however, real differences in the increase of modularity are found when the networks are initialized with low connectivity levels as may happen in the brain. In the case of networks with full connectivity, the increase of the average architectural modularity was low during evolution, indicating that the learning algorithm could deal with the cross-talk interference in that case.

Whether the tasks are solved separately (EPNet) or together (M-EPNet), similar structures and performances were found, indicating that modular architectures are beneficial to solve combined classification or prediction tasks. Note that no previous work has been found that forecast TS with modular architectures as done in this study. Moreover, it has been shown that SSPs and MSPs methods can be used to predict a given combined TS during evolution. Finally, it was shown that module reuse is possible over ANNs (i.e. MNNs) through a series of experiments performed in different scenarios.

7.2 Further research paths

There are a number of different topics that were not possible to tackle during this study, but clearly they need further attention for the EPNet, M-EPNet algorithms and module reuse.

EPNet algorithm: one aspect to consider of the EPNet algorithm is the evolution of a learning rate parameter per node, which may not be straightforward considering just mutation operators with Lamarckian inheritance, but clearly, that could be helpful to improve the convergence and overall performance of the networks as shown in previous studies for generational algorithms [12].

Module Evolution: The modularity measure assumes the output partition is known in advance, thus, an improvement would be to adjust dynamically this aspect during evolution, or the usage of other modularity measures as the presented in [97]. Also, fine grain measures like the the clustering coefficient could be employed in deeper studies of modularity to understand more about the flow of information between modules, output partitions and module reuse. However, it may bear in mind, that the usage of another modularity measure may produce different results as the presented in this study.

The non-convergent method [71] mentioned in Sections 2.2.1.1 and 2.2.1.3 helps to eliminate connections according to their importance to the task. However, a more detailed analysis is required when modules are evolved, i.e. when the importance of connections are not only determined by a single task. This kind of study may help the modularity increase, and probably by a faster evolution of modules considering the cross-talk interference between sub-tasks. That is supported by Section 5.7, where it was shown that the weighted modularity ($M^{(weight)}$) was bigger than the architectural one ($M^{(arch.)}$). Therefore, some similarities may be discovered between the dependency values from the modularity measure and the non-convergent method of [71].

Module reuse: besides the prediction and classification tasks used in this study, one of the more promising areas that needs further research is module reuse, more generally, in dynamic environments, and in simulating the structural and functional organization of the brain. Therefore, it remains the usage of the M-EPNet algorithm in bigger representations to better appreciate the advantages of modularity.

Chapter 8

Appendix A

This section presents all the evolved parameters from prediction and classification with the EPNet algorithms (EPNet, FS-EPNet Fix&evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms) from Chapter 4.

Tables 8.1 - 8.8 present the prediction tasks, where the first part of them (top) shows the EPNet and FS-EPNet Fix&evolve algorithms, and the second part (bottom) shows the FS-EPNet and FS$_{AD}$-EPNet implementations. The four experiments are presented in a single table to facilitate the comparisons.

Table 8.9 shows the evolved parameters for the *Breast, Optical digit* and *Thyroid* data sets with the EPNet and FS$_{AD}$-EPNet algorithms.

Table 8.1: *Logistic* TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	1	0	1	1	1	0	1	1
Number of Delays	12	0	12	12	12	0	12	12
Number of Hidden Nodes	4.700	1.579	2	9	4	1.221	3	7
Number of Connections	13.733	4.464	7	27	12.300	3.250	8	20
NRMSE Validation Error	6.02E-03	0.003	1.62E-03	1.38E-02	5.80E-03	0.003	1.70E-03	9.58E-03
NRMSE Test Set	5.39E-03	0.003	1.21E-03	1.16E-02	5.08E-03	0.002	1.53E-03	8.65E-03
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	2.067	1.596	1	5	1	0	1	1
Number of Delays	2.900	1.269	1	7	2.467	1.137	1	4
Number of Hidden Nodes	8.567	5.302	3	28	18.000	5.760	7	33
Number of Connections	36.567	37.058	8	185	112.800	58.170	41	293
NRMSE Validation Error	2.80E-03	0.003	1.60E-04	8.66E-03	2.96E-04	0.000	2.18E-05	1.40E-03
NRMSE Test Set	2.72E-03	0.003	1.73E-04	7.72E-03	3.64E-04	0.000	2.19E-05	2.02E-03

Table 8.2: $Lo_A^{(ssp)}$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	10	0	10	10	4.167	1.877	2	8
Number of Delays	15	0	15	15	1.767	0.971	1	4
Number of Hidden Nodes	17.267	8.670	2	35	19.833	8.659	9	42
Number of Connections	136.467	92.491	15.000	411.000	135.133	103.313	26.000	454
NRMSE Validation Error	1.18E-02	0.007	5.34E-03	3.67E-02	4.60E-04	0.000	3.33E-05	9.07E-04
NRMSE Test Set	2.75E-02	0.016	1.23E-02	8.08E-02	1.19E-03	0.001	4.86E-05	3.16E-03
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	4.567	1.406	2	8	4.500	1.634	3	8
Number of Delays	1.933	0.828	1	4	2.133	1.074	1	4
Number of Hidden Nodes	20.667	8.285	3	34	19.300	8.159	7	37
Number of Connections	168.067	102.033	20	401	156.833	102.201	37	420
NRMSE Validation Error	4.31E-04	0.0003	1.56E-04	0.001	5.82E-04	0.000	1.42E-04	1.49E-03
NRMSE Test Set	9.90E-04	0.001	2.48E-04	0.003	1.24E-03	0.001	2.39E-04	3.30E-03

Table 8.3: $Lo_{B1}^{(ssp)}$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	2	0	2	2	3.500	0.777	3	6
Number of Delays	3	0	3	3	1.000	0.000	1	1
Number of Hidden Nodes	18.800	7.151	6	36	17.633	8.640	10	36
Number of Connections	101.867	87.613	19	350	131.600	97.410	50	431
NRMSE Validation Error	4.27E-02	0.010	2.85E-02	7.16E-02	4.82E-03	0.002	2.27E-03	7.07E-03
NRMSE Test Set	3.56E-02	0.005	2.80E-02	4.40E-02	4.59E-03	0.002	1.86E-03	6.59E-03
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	8.067	2.116	4	13	5.266	1.337	3	8
Number of Delays	1.200	0.407	1	2	1.233	0.430	1	2
Number of Hidden Nodes	28.667	7.198	17	45	26.867	6.740	14	44
Number of Connections	332.567	123.180	126	594	284.033	109.769	60	491
NRMSE Validation Error	2.52E-03	0.001	9.54E-04	4.76E-03	3.38E-03	0.002	1.31E-03	9.38E-03
NRMSE Test Set	3.69E-03	0.003	1.02E-03	1.50E-02	4.28E-03	0.003	1.29E-03	1.37E-02

Table 8.4: $Lo_{B2}^{(msp)}$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	2	0	2	2	4.133	0.900	3	6
Number of Delays	3	0	3	3	2.600	0.621	1	4
Number of Hidden Nodes	14.733	3.991	7	22	13.300	5.484	5	26
Number of Connections	83.267	36.888	28	198	88.967	49.099	32	210
NRMSE Validation Error	5.43E-02	0.021	1.42E-02	9.59E-02	2.41E-02	0.017	7.66E-03	8.82E-02
NRMSE Test Set	1.03E+00	0.052	9.51E-01	1.13E+00	6.64E-01	0.310	7.26E-02	1.110
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	6.700	1.915	3	10	5.633	1.607	3	8
Number of Delays	2.467	0.937	1	4	2.333	0.884	1	4
Number of Hidden Nodes	13.333	4.253	5	26	12.733	3.723	7	23
Number of Connections	108.467	56.199	39	351	95.967	39.011	35	196
NRMSE Validation Error	1.66E-02	0.008	6.76E-03	3.48E-02	1.96E-02	0.009	5.56E-03	4.71E-02
NRMSE Test Set	5.23E-01	0.262	9.83E-02	9.26E-01	4.72E-01	0.329	4.90E-02	1.050

Table 8.5: $Lo_{B3}^{(msp)}$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	2	0	2	2	3.600	2.061	1	10
Number of Delays	3	0	3	3	4.300	1.915	1	8
Number of Hidden Nodes	10.400	4.280	1	20	9.167	4.300	1	17
Number of Connections	44.567	24.438	1	114	47.300	27.571	1	108
NRMSE Validation Error	8.10E-01	0.061	7.09E-01	1.002	6.47E-01	0.140	3.36E-01	1.002
NRMSE Test Set	1.005	0.019	9.71E-01	1.067	1.006	0.019	9.66E-01	1.079
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	8.467	1.697	4	12	4.200	1.730	1	7
Number of Delays	3.600	1.276	1	7	2.800	1.031	1	4
Number of Hidden Nodes	9.433	3.081	4	20	10.600	4.082	4	22
Number of Connections	72.867	25.593	14	120	74.567	49.367	18	247
NRMSE Validation Error	3.93E-01	0.155	2.23E-01	6.95E-01	5.27E-01	0.170	2.50E-01	7.66E-01
NRMSE Test Set	1.033	0.031	9.80E-01	1.099	1.02E+00	0.026	9.77E-01	1.109

Table 8.6: $MG17$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	3	0	3	3	5.300	2.070	3	12
Number of Delays	10	0	10	10	7.667	2.746	5	13
Number of Hidden Nodes	9.033	2.785	4	17	15.200	6.738	6	31
Number of Connections	45.500	18.792	20	117	109.633	51.224	28	237
NRMSE Validation Error	3.78E-02	0.029	1.77E-02	1.85E-01	6.78E-03	0.004	3.05E-03	1.93E-02
NRMSE Test Set	1.05E+00	0.216	5.56E-01	1.71E+00	2.75E-01	0.150	1.43E-01	6.67E-01
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	8.333	2.397	5	13	5.866	1.357	4	8
Number of Delays	3.567	0.858	2	6	3.500	0.731	2	4
Number of Hidden Nodes	16.067	6.175	6	31	15.800	4.874	8	31
Number of Connections	150.600	78.150	52	376	133.467	65.764	55	297
NRMSE Validation Error	3.30E-03	0.002	1.19E-03	7.06E-03	3.84E-03	0.001	1.68E-03	6.18E-03
NRMSE Test Set	1.68E-01	0.046	7.40E-02	3.10E-01	2.50E-01	0.367	3.27E-02	1.681

Table 8.7: $MG17_A$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	3	0	3	3	5.033	2.157	3	11
Number of Delays	10	0	10	10	8.600	2.978	4	13
Number of Hidden Nodes	11.133	4.862	6	27	17.533	5.871	10	33
Number of Connections	65.667	44.320	27	234	139.700	64.409	47	321
NRMSE Validation Error	2.85E-02	0.008	1.30E-02	4.72E-02	6.13E-03	0.003	3.43E-03	1.48E-02
NRMSE Test Set	2.91E-01	0.347	1.27E-01	2.07E+00	5.15E-02	0.029	1.73E-02	1.34E-01
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	9.033	2.076	5	14	5.833	1.147	4	8
Number of Delays	3.267	0.980	2	6	3.200	0.805	2	4
Number of Hidden Nodes	19.167	6.165	8	30	20.200	5.530	10	35
Number of Connections	192.867	76.289	60	359	190.833	75.370	90	447
NRMSE Validation Error	3.00E-03	0.001	1.61E-03	7.52E-03	2.86E-03	9.65E-04	1.70E-03	6.82E-03
NRMSE Test Set	2.73E-02	0.027	7.20E-03	1.18E-01	4.17E-02	7.66E-02	6.17E-03	3.15E-01

Table 8.8: $MG30$ TS results with the EPNet, FS-EPNet Fix&Evolve, FS-EPNet and FS$_{AD}$-EPNet algorithms

Parameter	*EPNet*				*FS-EPNet Fix&Evolve*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	6	0	6	6	5.867	0.507	5	7
Number of Delays	14	0	14	14	13.867	0.346	13	14
Number of Hidden Nodes	20.067	5.705	9	30	19.000	5.502	10	32
Number of Connections	183.667	71.513	72	324	170.433	66.538	73	294
NRMSE Validation Error	6.54E-03	0.005	1.66E-03	1.87E-02	5.79E-03	0.004	2.14E-03	1.71E-02
NRMSE Test Set	1.01E-01	0.067	3.94E-02	2.86E-01	1.32E-01	0.086	3.64E-02	3.23E-01
	FS-EPNet				*FS$_{AD}$-EPNet*			
	Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Number of Inputs	8.333	0.661	8	11	6.466	1.136	4	8
Number of Delays	4.200	0.484	3	5	3.967	0.183	3	4
Number of Hidden Nodes	16.400	6.558	5	31	14.467	4.240	6	23
Number of Connections	146.600	81.082	37	353	115.333	47.409	34	213
NRMSE Validation Error	5.99E-03	0.004	2.71E-03	2.38E-02	1.10E-02	1.21E-02	2.39E-03	4.40E-02
NRMSE Test Set	9.08E-02	0.042	1.87E-02	1.59E-01	1.06E-01	7.19E-02	2.24E-02	3.19E-01

Table 8.9: *Breast, Optical digit* and *Thyroid* data sets results with the the EPNet, and FS$_{AD}$-EPNet algorithms

Data Set	*Parameter*	*EPNet*				*FS$_{AD}$-EPNet*			
		Mean	*Std Dev*	*Min*	*Max*	*Mean*	*Std Dev*	*Min*	*Max*
Breast cancer	Number of Inputs	9	0	9	9	9	0	9	9
	Number of Hidden Nodes	3.067	0.740	2	4	3.900	1.213	1	6
	Number of Connections	34.133	6.647	23	47	38.733	7.670	21	54
	Error percentage Validation Set	1.393	0.330	0.746	1.762	1.465	0.261	1.042	1.824
	Classification error Validation Set	2.438	0.732	1.493	2.985	2.289	0.757	1.493	2.985
	Error percentage Test Set	0.455	0.121	0.322	0.806	0.412	0.091	0.314	0.734
	Classification error Test Set	0.438	0.467	0	1.714	0.438	0.288	0	1.143
Optical digit	Number of Inputs	51	24	9	64	42.667	26.109	9	64
	Number of Hidden Nodes	13.133	7.533	3	25	9.100	5.416	2	21
	Number of Connections	800.100	482.212	29	1499	567.500	437.624	24	1309
	Error percentage Validation Set	0.957	0.301	0.660	1.619	1.141	0.298	0.627	1.588
	Classification error Validation Set	4.378	0.714	3.000	5.500	4.837	1.120	3.000	7.100
	Error percentage Test Set	1.092	0.285	0.772	1.773	1.254	0.215	0.897	1.689
	Classification error Test Set	5.325	0.716	4.229	7.791	6.124	0.674	4.786	7.234
Thyroid	Number of Inputs	21	0	21	21	14.833	3.206	8	20
	Number of Hidden Nodes	14.800	4.326	8	29	18.000	6.747	7	37
	Number of Connections	269.900	88.088	143	604	337.933	160.908	107	825
	Error percentage Validation Set	0.585	0.096	0.413	0.786	0.485	0.089	0.298	0.651
	Classification error Validation Set	0.986	0.201	0.556	1.351	0.771	0.177	0.477	1.113
	Error percentage Test Set	0.852	0.072	0.743	1.033	0.812	0.091	0.545	0.983
	Classification error Test Set	1.575	0.160	1.313	1.925	1.449	0.168	0.933	1.721

Bibliography

[1] J. H. Kaas, “Why is brain size so important: Design problems and solutions as neocortex gets bigger or smaller,” *Brain and Mind*, vol. 1, pp. 7–23, 2000.

[2] D. B. Chklovskii, T. Schikorski, and C. F. Stevens, “Wiring optimization in cortical circuits,” *Neuron*, vol. 34, no. 3, pp. 341–347, 2002.

[3] D. Meunier, R. Lambiotte, and E. T. Bullmore, “Modular and hierarchically modular organization of brain networks,” *Frontiers in Neuroscience*, vol. 4, no. 0, 2010.

[4] M. L. Anderson, “Neural reuse: A fundamental organizational principle of the brain,” *Behavioral and Brain Sciences*, vol. 33, pp. 245 – 313, 2010.

[5] D. S. Bassett and M. S. Gazzaniga, “Understanding complexity in the human brain,” *Trends in Cognitive Sciences*, vol. 15, no. 5, pp. 200 – 209, 2011.

[6] J. A. Fodor, *The Modularity of Mind: an Essay on Faculty Psychology.* Cambridge, MA: The MIT Press, April 1983.

[7] B. Seok, “Diversity and unity of modularity,” *Cognitive Science*, vol. 30, pp. 347 – 380, 2006.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel distributed processing: explorations in the microstructure of cognition*, vol. 1: Foundations, ch. Learning internal representations by error propagation, pp. 318–362. Cambridge, MA, USA: MIT Press, 1986.

[9] J. A. Bullinaria, “Using evolution to improve neural network learning: pitfalls and solutions,” *Neural Computing and Applications*, vol. 16, no. 3, pp. 209–226, 2007.

[10] X. Yao, “Evolving artificial neural networks,” *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[11] J. A. Bullinaria, “To modularize or not to modularize?,” in *Proceedings of the 2002 U.K. Workshop on Computational Intelligence (UKCI 2002)* (J. Bullinaria, ed.), (Birmingham, UK: The University of Birmingham), pp. 3–10, 2002.

[12] J. A. Bullinaria, “Understanding the emergence of modularity in neural systems.,” *Cognitive Science*, vol. 31, no. 4, pp. 673–695, 2007.

[13] V. R. Khare, X. Yao, B. Sendhoff, Y. Jin, and H. Wersing, “Co-evolutionary modular neural networks for automatic problem decomposition,” in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (D. Corne, Z. Michalewicz, B. McKay, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, G. Raidl, K. C. Tan, and A. Zalzala, eds.), vol. 3, (Edinburgh, Scotland, UK), pp. 2691–2698, IEEE Press, 2-5 Sept. 2005.

[14] V. Gordon and J. Crouson, “Self-splitting modular neural network - domain partitioning at boundaries of trained regions,” in *IEEE International Joint Conference on Neural Networks. IEEE World Congress on Computational Intelligence*, IJCNN 2008, pp. 1085–1091, june 2008.

[15] J. Reisinger, K. O.Stanley, and R. Miikkulainen, “Evolving reusable neural modules,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, (New York, UK), pp. 68–81, Springer-Verlag, 2004.

[16] J. G. Rueckl, K. R. Cave, and S. M. Kosslyn, “Why are what and where processed by separate cortical visual systems? a computational investigation,” *Journal of Cognitive Neuroscience*, vol. 1, no. 2, pp. 171–186, 1989.

[17] M. Hüsken, C. Igel, and M. Toussaint, “Task-dependent evolution of modularity in neural networks,” *Connection Science*, vol. 14, 2002.

[18] J. A. Bullinaria, “Understanding the advantages of modularity in neural systems,” in *Proceedings of the Twenty-eighth Annual Conference of the Cognitive Science Society*, (Mahwah), pp. 119–124, NJ: Lawrence Erlbaum Associates, 2006.

[19] N. Kashtan and U. Alon, “Spontaneous evolution of modularity and network motifs,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102, pp. 13773 – 13778, Sep. 2005.

[20] F. Chang, “Symbolically speaking: a connectionist model of sentence production,” *Cognitive Science*, vol. 26, pp. 609–651, 2002.

[21] B. L. M. Happel and J. M. J. Murre, “The design and evolution of modular neural network architectures,” *Neural Networks*, vol. 7, pp. 985–1004, 1994.

[22] H. Lipson, J. B. Pollack, and N. P. Suh, “On the origin of modular variation,” *Evolution*, vol. 56, no. 8, pp. 1549–1556, 2002.

[23] N. Kashtan, A. E. Mayo, T. Kalisky, and U. Alon, “An analytically solvable model for rapid evolution of modular structure,” *PLoS Comput Biol*, vol. 5, p. e1000355, 04 2009.

[24] E. Schlessinger, P. J. Bentley, and R. B. Lotto, “Modular thinking: evolving modular neural networks for visual guidance of agents,” in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO ’06, (New York, NY, USA), pp. 215–222, ACM, 2006.

[25] J. Clune, B. E. Beckmann, P. K. McKinley, and C. Ofria, “Investigating whether hyperneat produces modular neural networks,” in *Proceedings of the 12th annual*

conference on Genetic and evolutionary computation, GECCO '10, (New York, NY, USA), pp. 635–642, ACM, 2010.

[26] G. P. Wagner, "Homologues, natural kinds, and the evolution of modularity," *American Zoologist*, vol. 36, pp. 36–43, 1996.

[27] E. Hemberg, C. Gilligan, M. ONeill, and A. Brabazon, "A grammatical genetic programming approach to modularity in genetic algorithms," in *Genetic Programming* (M. Ebner, M. ONeill, A. Ekárt, L. Vanneschi, and A. Esparcia-Alcázar, eds.), vol. 4445 of *Lecture Notes in Computer Science*, pp. 1–11, Springer Berlin, Heidelberg, 2007.

[28] I. Jonyer and A. Himes, *Improving Modularity in Genetic Programming Using Graph-Based Data Mining*, pp. 556–561. American Association for Artificial Intelligence, 2006.

[29] K. Krawiec and B. Wieloch, "Functional modularity for genetic programming," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, (New York, NY, USA), pp. 995–1002, ACM, 2009.

[30] N. NourAshrafoddin, A. R. Vahdat, and M. M. Ebadzadeh, *Automatic Design of Modular Neural Networks Using Genetic Programming*, vol. 4668, pp. 788–798. Springer, 2007.

[31] J. R. Woodward, "Modularity in genetic programming," in *In Genetic Programming, Proceedings of EuroGP 2003*, pp. 14–16, Springer-Verlag, 2003.

[32] G. P. W. Gerhard Schlosser, *Modularity in Development and Evolution.* University of Chicago Press, 2004.

[33] N. J. Radcliffe, "Genetic set recombination and its application to neural network

topology optimisation," *Neural Computing and Applications*, vol. 1, no. 1, pp. 67–90, 1993.

[34] J. A. Bullinaria, "Evolved dual weight neural architectures to facilitate incremental learning," in *Proceedings of the International Joint Conference on Computational Intelligence (IJCCI 2009)*, (Portugal: INSTICC), pp. 427–434, 2009.

[35] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 694–713, 1997.

[36] R. A. Jacobs, M. I. Jordan, and A. G. Barto, "Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks," *Cognitive Science: A Multidisciplinary Journal*, vol. 15, pp. 219–250, 1991.

[37] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence Through Simulated Evolution*. New York: Wiley, 1966.

[38] D. B. Fogel, *Evolutionary computation: toward a new philosophy of machine intelligence*. Piscataway, NJ, USA: IEEE Press, 1995.

[39] N. Saravanan and D. Fogel, "Evolving neural control systems," *IEEE Expert*, vol. 10, pp. 23–27, jun 1995.

[40] H.-P. Schwefel, *Numerical Optimization of Computer Models*. New York, NY, USA: John Wiley & Sons, Inc., 1981.

[41] H.-P. P. Schwefel, *Evolution and Optimum Seeking: The Sixth Generation*. New York, NY, USA: John Wiley & Sons, Inc., 1993.

[42] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: Ann Arbor, MI: University of Michigan Press, 1975.

[43] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[44] J. Schaffer, D. Whitley, and L. Eshelman, “Combinations of genetic algorithms and neural networks: a survey of the state of the art,” in *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)* (D. Whitley and J. Schaffer, eds.), (Piscataway, New Jersey), pp. 1–37, IEEE Press, jun 1992.

[45] F. Gruau, D. Whitley, and L. Pyeatt, “A comparison between cellular encoding and direct encoding for genetic neural networks,” in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. Koza, D. Goldberg, D. Fogel, and R. Riolo, eds.), (CA, USA), pp. 81–89, MIT Press, 1996.

[46] L. D. Whitley, V. S. Gordon, and K. E. Mathias, “Lamarckian evolution, the baldwin effect and function optimization,” in *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, PPSN III, (London, UK), pp. 6–15, Springer-Verlag, 1994.

[47] E. Cantu-Paz and C. Kamath, “An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems,” *IEEE Transactions on Systems, Man and Cybernetics, Part B*, vol. 35, pp. 915–927, Oct. 2005.

[48] G. E. Hinton and S. J. Nowlan, “How Learning Can Guide Evolution,” *Complex Systems*, vol. 1, pp. 495–502, 1987.

[49] D. C. Heath, Company, L. M. Bierer, and V. F. Lien, *Heath life science.* D.C. Heath, 1984.

[50] J. M. Baldwin, “A new factor in evolution,” *The American Naturalist*, vol. 30, no. 354, pp. 441–451, 1896.

[51] C. M. Bishop, *Neural Networks for Pattern Recognition.* Oxford University Press, November 1995.

[52] L. Prechelt, “Early stopping-but when?,” in *Neural Networks: Tricks of the Trade*, vol. 1524 of *Lecture Notes in Computer Science*, (London, UK), pp. 55–69, Springer-Verlag, 1998.

[53] D. E. Moriarty and R. Miikkulainen, “Forming neural networks through efficient and adaptive coevolution,” *Evolutionary Computation*, vol. 5, no. 4, pp. 373–399, 1997.

[54] V. K. Valsalam and R. Miikkulainen, “Evolving symmetric and modular neural networks for distributed control,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO ’09, (New York, NY, USA), pp. 731–738, ACM, 2009.

[55] F. Gomez, J. Schmidhuber, and R. Miikkulainen, “Accelerated neural evolution through cooperatively coevolved synapses,” *The Journal of Machine Learning Research*, vol. 9, pp. 937–965, 2008.

[56] M. Dash and H. Liu, “Feature selection for classification,” *Intelligent Data Analysis*, vol. 1, pp. 131–156, 1997.

[57] G. Zhang, B. E. Patuwo, and M. Y. Hu, “Forecasting with artificial neural networks: The state of the art,” *International Journal of Forecasting*, vol. 14, pp. 35–62, March 1998.

[58] A. Kehagias and V. Petridis, “Predictive modular neural networks for time series classification,” *Neural Networks*, vol. 10, no. 1, pp. 31–49, 1997.

[59] C. Chatfield, “Time series forecasting with neural networks,” *Proceedings of the 1998 IEEE Signal Processing Society Workshop Neural Networks for Signal Processing VIII, 1998.*, pp. 419–427, 31 Aug-2 Sep 1998.

[60] T. J. Cholewo and J. M. Zurada, “Sequential network construction for time series

prediction," *International Conference on Neural Networks*, vol. 4, pp. 2034–2038, Jun 1997.

[61] Y. LeCun, "Efficient learning and second-order methods, a tutorial," *In Advances in neural information processing, Denver, CO*, vol. 6, 1993.

[62] J. A. Bullinaria, "Evolving neural networks: Is it really worth the effort?," in *Proceedings of the European Symposium on Artificial Neural Networks*, pp. 267–272, Evere, Belgium: d-side, 2005.

[63] H. Braun and P. Zagorski, "ENZO-M - A hybrid approach for optimizing neural networks by evolution and learning," in *Parallel Problem Solving from Nature – PPSN III*, (Berlin), pp. 440–451, Springer, 1994.

[64] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," tech. rep., Amherst, MA, USA, 1987.

[65] X. Yao and Y. Liu, "EPNet for chaotic time-series prediction," in *SEAL'96: Selected papers from the First Asia-Pacific Conference on Simulated Evolution and Learning*, (London, UK), pp. 146–156, Springer-Verlag, 1997.

[66] P. J. Werbos, *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting.* New York, NY, USA: Wiley-Interscience, 1994.

[67] V. Landassuri-Moreno and J. A. Bullinaria, "Neural network ensembles for time series forecasting," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, (New York, NY, USA), pp. 1235–1242, ACM, 2009.

[68] X. Yao and Y. Liu, "Making use of population information in evolutionary artificial neural networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 28, pp. 417–425, Jun. 1998.

[69] X. Yao and Y. Liu, "Ensemble structure of evolutionary artificial neural networks," *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 659–664, 1996.

[70] X. Yao and M. Islam, "Evolving artificial neural network ensembles," *Computational Intelligence Magazine, IEEE*, vol. 3, pp. 31–42, Feb. 2008.

[71] W. Finnoff, F. Hergert, and H. G. Zimmermann, "Improving model selection by nonconvergent methods," *Neural Networks*, vol. 6, no. 6, pp. 771–783, 1993.

[72] S. V. Odri, D. P. Petrovacki, and G. A. Krstonosic, "Evolutional development of a multilevel neural network," *Neural Networks*, vol. 6, no. 4, pp. 583–595, 1993.

[73] F. E. H. Tay and L. J. Cao, "ϵ-descending support vector machines for financial time series forecasting," *Neural Processing Letters*, vol. 15, no. 2, pp. 179–195, 2002.

[74] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl, "Automatic feature selection in neuroevolution," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, (New York, NY, USA), pp. 1225–1232, ACM, 2005.

[75] D. W. Opitz, "Feature selection for ensembles," *Proceedings of 16 th International Conference on Artificial Intelligence*, pp. 379–384, 1999.

[76] E. Schlessinger, P. J. Bentley, and B. R. Lotto, "Analysing the evolvability of neural network agents through structural mutations," in *Proceeding of European Conference on Artificial Life (ECAL 2005)*, (Canterbury, UK), Springer-Verlag Berlin Heidelberg, September 5-9 2005.

[77] H. A. Mayer and R. Schwaiger, "Evolutionary and coevolutionary approaches to time series prediction using generalized multi-layer perceptrons," in *Proceedings of the 1999*

Congress on Evolutionary Computation (P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, eds.), vol. 1, (Mayflower Hotel, Washington D.C., USA), pp. 275–280, IEEE Press, 6-9 July 1999.

[78] I. Harvey, "Species adaption genetic algorithms: A basis for a continuing SAGA," in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life.* (F. J. Varela and P. Bourgine, eds.), pp. 346–354, MIT Press, 1992.

[79] L. Bull, "Coevolutionary species adaptation genetic algorithms: growth and mutation on coupled fitness landscapes," in *The 2005 IEEE Congress on Evolutionary Computation*, vol. 1, pp. 559–564, IEEE Press, sept. 2005.

[80] D. B. D'Ambrosio and K. O. Stanley, "A novel generative encoding for exploiting neural network sensor and output geometry," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, no. 8 in GECCO '07, (New York, NY, USA), pp. 974–981, ACM, 2007.

[81] R. V. Solé and S. Valverde, "Spontaneous emergence of modularity in cellular networks," *Journal of The Royal Society Interface*, vol. 5, pp. 129–133, January 2008.

[82] G. Palla, I. Derenyi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, pp. 814–818, 2005.

[83] O. Sporns, D. Chialvo, M. Kaiser, and C. Hilgetag, "Organization, development and function of complex brain networks," *Trends in Cognitive Sciences*, vol. 8, pp. 418–425, Sept. 2004.

[84] C. F. Stevens, "How cortical interconnectedness varies with network size," *Neural Computation*, vol. 1, pp. 473–479, December 1989.

[85] P. MacLean, *The triune brain in evolution: role in paleocerebral functions.* Plenum Press, 1990.

[86] H. A. Simon, "The architecture of complexity," in *Proceedings of the American Philosophical Society*, vol. 106, pp. 467–482, 1962.

[87] M. L. Anderson, "Circuit sharing and the implementation of intelligent systems," *Connection Science*, vol. 20, pp. 239–251, December 2008.

[88] V. Gallese and G. Lakoff, "The brain's concepts: The role of the sensory-motor system in reason and language," *Cognitive Neuropsychology*, vol. 22, no. 3–4, pp. 455–479, 2005.

[89] S. L. Hurley, "The shared circuits hypothesis: A unified functional architecture for control, imitation and simulation," in *Perspectives on imitation: From neuroscience to social science* (S. Hurley and N., eds.), vol. 1, pp. 76–95, MIT Press, 2005.

[90] S. Hurley, "The shared circuits model (SCM): How control, mirroring, and simulation can enable imitation, deliberation, and mindreading," *Behavioral and Brain Sciences*, vol. 31, no. 1, pp. 1–22, 2008.

[91] S. Dehaene, "Evolution of human cortical circuits for reading and arithmetic: The "neuronal recycling" hypothesis," in *From monkey brain to human brain* (S. Dehaene, J. R. Duhamel, M. Hauser, and G. Rizzolatti, eds.), pp. 133–157, Cambridge, Massachusetts, MIT Press, 2004.

[92] S. Dehaene and L. Cohen, "Cultural recycling of cortical maps," *Neuron*, vol. 56, no. 2, pp. 384–398, 2007.

[93] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. Pages 128–135, 1 April 1999.

[94] A. Robins, “Catastrophic forgetting, rehearsal and pseudorehearsal,” *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995.

[95] D. J. Watts and S. H. Strogatz, “Collective dynamics of ’small-world’ networks,” *Nature*, vol. 393, pp. 440–442, 1998.

[96] M. E. J. Newman and M. Girvan, “Finding and evaluating community structure in networks,” *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004.

[97] M. E. J. Newman, “Finding community structure in networks using the eigenvectors of matrices,” *Phys. Rev. E*, vol. 74, no. 3, p. 36104, 2006.

[98] R. Guimerá and L. A. Nunes Amaral, “Functional cartography of complex metabolic networks,” *Nature*, vol. 433, pp. 895–900, 2005.

[99] E. Ronco and P. J. Gawthrop, “Modular neural networks: State of the art,” Tech. Rep. CSC-95026, Centre for System and Control. University of Glasgow, Glasgow, UK, May 1995.

[100] V. R. Khare, X. Yao, and B. Sendhoff, “Multi-network evolutionary systems and automatic problem decomposition,” *International Journal of General Systems*, vol. 35, no. 3, pp. 259–274, 2006.

[101] M. Hüsken, J. E. Gayko, and B. Sendhoff, “Optimization for problem classes - neural networks that learn to learn,” in *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, pp. 98–109, IEEE Press, 2000.

[102] G. P. Zhang and V. L. Berardi, “Time series forecasting with neural network ensembles: An application for exchange rate prediction,” *The Journal of the Operational Research Society*, vol. 52, no. 6, pp. 652–664, 2001.

[103] Y. Liu and X. Yao, “Evolving modular neural networks which generalise well,” in *IEEE International Conference on Evolutionary Computation*, pp. 605–610, apr 1997.

[104] L. K. Hansen and P. Salamon, "Neural network ensembles," *Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 10, pp. 993–1001, 1990.

[105] M. Islam, X. Yao, S. Shahriar Nirjon, M. Islam, and K. Murase, "Bagging and boosting negatively correlated neural networks," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 38, pp. 771–784, june 2008.

[106] S. D. Bay, "Nearest neighbor classification from multiple feature subsets," *Intelligent Data Analysis*, vol. 3, no. 3, pp. 191–209, 1999.

[107] R. Avnimelech and N. Intrator, "Boosting regression estimators," *Neural Computation*, vol. 11, no. 2, pp. 499–520, 1999.

[108] R. Meir, "Bias, variance and the combination of estimators; the case of linear least squares," in *Advances in Neural Information Processing Systems 7*, Morgan Kaufmann, 1995.

[109] R. E. Schapire, "The strength of weak learnability," *Mach. Learn.*, vol. 5, pp. 197–227, July 1990.

[110] R. Avnimelech and N. Intrator, "Boosted mixture of experts: an ensemble learning scheme," *Neural Computation*, vol. 11, no. 2, pp. 483–497, 1999.

[111] M. H. Nguyen, H. A. Abbass, and R. I. Mckay, "A novel mixture of experts model based on cooperative coevolution," *Neurocomputing*, vol. 70, no. 1-3, pp. 155–163, 2006.

[112] E. D. Ubeyli, "Wavelet/mixture of experts network structure for eeg signals classification," *Expert Systems with Applications*, vol. 34, no. 3, pp. 1954–1962, 2008.

[113] G. Auda, M. Kamel, and H. Raafat, "Modular neural network architectures for classification," *IEEE International Conference on Neural Networks*, vol. 2, pp. 1279–1284, 3-6 Jun 1996.

[114] T. Caelli, L. Guan, and W. Wen, “Modularity in neural computing,” *Proceedings of the IEEE*, vol. 87, pp. 1497–1518, sep 1999.

[115] B.-L. Lu and M. Ito, “Task decomposition and module combination based on class relations: a modular neural network for pattern classification,” *IEEE Transactions on Neural Networks*, vol. 10, no. 5, pp. 1244–1256, Sep 1999.

[116] J. Holland, “Adaptation,” in *Progress in Theoretical Biology 4* (R. Rosen and F. Snell, eds.), pp. 263–293, New York: Academic Press, 1976.

[117] T. O’Hara and L. Bull, “Building anticipations in an accuracy-based learning classifier system by use of an artificial neural network,” in *The 2005 IEEE Congress on Evolutionary Computation, 2005.*, vol. 3, pp. 2046– 2052, sept 2005.

[118] V. R. Khare, *Automatic Problem Decomposition using Co-evolution and Modular Neural Networks*. PhD thesis, The University of Birmingham, 2006.

[119] C. MacLeod, G. Maxwell, and S. Muthuraman, “Incremental growth in modular neural networks,” *Engineering Applications of Artificial Intelligence*, vol. 22, no. 4-5, pp. 660–666, 2009.

[120] J. Molina-Vilaplana, J. Feliu-Batlle, and J. Lpez-Coronado, “A modular neural network architecture for step-wise learning of grasping tasks,” *Neural Networks*, vol. 20, no. 5, pp. 631–645, 2007.

[121] N. García-Pedrajas and D. Ortiz-Boyer, “A cooperative constructive method for neural networks for pattern recognition,” *Pattern Recognition*, vol. 40, no. 1, pp. 80–98, 2007.

[122] T. Drabe, W. Bressgott, and E. Bartscht, “Genetic task clustering for modular neural networks,” *Neural Networks for Identification, Control, and Robotics, International Workshop*, pp. 339–347, 1996.

[123] J. Santos, L. Alexandre, and J. de Sa, "Modular neural network task decomposition via entropic clustering," in *Sixth International Conference on Intelligent Systems Design and Applications, 2006. ISDA '06.*, vol. 1, pp. 62–67, oct. 2006.

[124] V. Petridis and A. Kehagias, *Predictive Modular Neural Networks: Applications to Time Series.* Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[125] J. A. Bullinaria, "The importance of neurophysiological constraints for modelling the emergence of modularity," in *Computational Modelling in Behavioural Neuroscience: Closing the Gap Between Neurophysiology and Behaviour* (D. Heinke and E. Mavritsaki, eds.), (Hove, UK), pp. 187–208, Psychology Press, 2009.

[126] S. Nolfi, "Using emergent modularity to develop control systems for mobile robots," *Adaptive Behavior*, vol. 5, pp. 343–363, 1997.

[127] R. Calabretta, S. Nolfi, D. Parisi, and G. P. Wagner, "A case study of the evolution of modularity: towards a bridge between evolutionary biology, artificial life, neuro- and cognitive science," in *Proceedings of the sixth international conference on Artificial life*, ALIFE, (Cambridge, MA, USA), pp. 275–284, MIT Press, 1998.

[128] R. A. Watson and J. B. Pollack, "Symbiotic composition and evolvability," in *Proceedings of the 6th European Conference on Advances in Artificial Life* (S. P. Kelemen, J., ed.), ECAL '01, (London, UK), pp. 480–490, Springer-Verlag, 2001.

[129] R. J. Frank, N. Davey, and S. Hunt., "Time series prediction and neural networks," *Journal of Intelligent and Robotic Systems*, vol. 31, pp. 91–103, 2001.

[130] H. Takenaga, S. Abe, M. Takatoo, M. Kayama, T. Kitamura, and Y. Okuyama, "Optimal input selection of neural networks by sensitivity analysis and its application to image recognition," *Proceedings of the MVA '90. IAPR Workshop on Machine Vision Applications*, 1990.

[131] V. Landassuri-Moreno and J. A. Bullinaria, "Feature selection in evolved artificial neural networks using the evolutionary algorithm EPNet," in *Proceedings of the 2009 UK Workshop on Computational Intelligence*, UKCI '2009, (Nottingham, UK: University of Nottingham.), July 2009.

[132] N. A. Gershenfeld and A. S. Weigend, "The future of time series: Learning and understanding," in *Time series prediction. Forecasting the future and understanding the past. Proceedings of the NATO Advanced Research Workshop on Comparative Time Series Analysis* (A. S. Weigend and N. A. Gershenfeld, eds.), vol. XV, pp. 1–70, 1992.

[133] D. P. Mandic and J. A. Chambers, *Recurrent neural networks for prediction: learning algorithms, architectures and stability.* New York: John Wiley & Sons, 2001.

[134] W. Sarle, "Neural network faq, part 3 of 7: Generalization, periodic posting to the usenet newsgroup comp.ai.neural-nets," 1997.

[135] S. Makridakis and S. C. Wheelwright, *Forecasting Methods & Applications.* John Wiley & Son, 1978.

[136] J. Belaire-Franch and D. Contreras, "Recurrence plots in nonlinear time series analysis: Free software," *Journal of Statistical Software*, vol. 7, no. 9, 2002.

[137] F. Takens, "Detecting strange attractors in turbulence," in *Dynamical Systems and Turbulence, Warwick 1980*, vol. 898, (Berlin), pp. 366–381, Springer, 1981.

[138] I. D. Falco, A. Iazzetta, P. Natale, and E. Tarantino, "Evolutionary neural networks for nonlinear dynamics modeling," in *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature* (A. E. Eiben, T. Back, M. Schoenauer, and H.-P. Schwefel, eds.), (London, UK), pp. 593–602, Springer-Verlag, 1998.

[139] I. Rojas, H. Pomares, J. L. Bernier, J. Ortega, B. Pino, F. J. Pelayo, and A. Prieto, "Time series analysis using normalized pg-rbf network with regression weights," *Neurocomputing*, vol. 42, no. 1-4, pp. 267–285, 2002.

[140] A. Gholipour, B. N. Araabi, and C. Lucas, "Predicting chaotic time series using neural and neurofuzzy models: A comparative study," *Neural Processing Letters*, vol. 24, no. 3, pp. 217–239, 2006.

[141] K.-R. Müller, A. J. Smola, G. Rätsch, B. Schökopf, J. Kohlmorgen, and V. Vapnik, "Using support vector machines for time series prediction," pp. 243–253, 1999.

[142] K.-R. Müller, A. J. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik, "Predicting time series with support vector machines," in *ICANN '97: Proceedings of the 7th International Conference on Artificial Neural Networks*, (London, UK), pp. 999–1004, Springer-Verlag, 1997.

[143] S. V. Dudul, "Prediction of a lorenz chaotic attractor using two-layer perceptron neural network," *Applied Soft Computing*, vol. 5, no. 4, pp. 333–355, 2005.

[144] F. A. Guerra and L. dos S. Coelho, "Multi-step ahead nonlinear identification of lorenz's chaotic system using radial basis neural network with learning by clustering and particle swarm optimization," *Chaos, Solitons & Fractals*, vol. 35, no. 5, pp. 967 – 979, 2008.

[145] R. Mikolajczak and J. Mandziuk, "Comparative study of logistic map series prediction using feed-forward, partially recurrent and general regression networks," *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP '02.*, vol. 5, pp. 2364–2368, Nov. 2002.

[146] J. Mcdonnell and D. Waagen, "Evolving recurrent perceptrons for time-series modeling," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 24–38, 1994.

[147] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of atmospheric Science*, vol. 20, pp. 130–141, 1963.

[148] C. Igel and M. Hsken, "Empirical evaluation of the improved Rprop learning algorithms," *Neurocomputing*, vol. 50, pp. 105–123, 2003.

[149] I. Rojas, O. Valenzuela, F. Rojas, A. Guillen, L. Herrera, H. Pomares, L. Marquez, and M. Pasadas, "Soft-computing techniques and arma model for time series prediction," *Neurocomputing*, vol. 71, no. 4-6, pp. 519–537, 2008.

[150] M. Ardalani-Farsa and S. Zolfaghari, "Chaotic time series prediction with residual analysis method using hybrid elman-narx neural networks," *Neurocomputing*, vol. 73, no. 13-15, pp. 2540–2553, 2010.

[151] M. Assaad, R. Bon, and H. Cardot, "A new boosting algorithm for improved time-series forecasting with recurrent neural networks," *Information Fusion*, vol. 9, no. 1, pp. 41–55, 2008. Special Issue on Applications of Ensemble Methods.

[152] D.-T. Lin, J. E. Dayhoff, and P. A. Ligomenides, "Trajectory production with the adaptive time-delay neural network," *Neural Netw.*, vol. 8, pp. 447–461, May 1995.

[153] R. S. Crowder, "Predicting the mackey-glass time series with cascade-correlation learning," in *Connectionist Models: Proceedings of the 1990 Summer School* (D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, eds.), (San Mateo, CA.), pp. 524–532, Morgan Kaufmann, 1990.

[154] E. Cantú-Paz and C. Kamath, "Using evolutionary algorithms to induce oblique decision trees," in *GECCO*, pp. 1053–1060, 2000.

[155] R. Setiono and L. Hui, "Use of a quasi-newton method in a feedforward neural network construction algorithm," *IEEE Transactions on Neural Networks*, vol. 6, pp. 273 –277, jan 1995.

[156] L. Prechelt, "PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms," Tech. Rep. 21/94, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, Sept. 1994.

[157] S. He, Q. Wu, and J. Saunders, "Group search optimizer: An optimization algorithm inspired by animal searching behavior," *IEEE Transactions on Evolutionary Computation*, vol. 13, pp. 973–990, Oct. 2009.

[158] E. Tang, P. Suganthan, X. Yao, and A. Qin, "Linear dimensionality reduction using relevance weighted lda," *Pattern Recognition*, vol. 38, no. 4, pp. 485 – 493, 2005.

[159] E. Alpaydin, "Combined 5 x 2 cv f test for comparing supervised classification learning algorithms.," *Neural computation*, vol. 11, pp. 1885–1892, November 1999.

[160] T. Masters, *Practical Neural Network Recipes in C++*. San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[161] V. M. Landassuri-Moreno and J. A. Bullinaria, "Biasing the evolution of modular neural networks," in *Proceedings of the 2011 IEEE Congress on Evolutionary Computation* (A. E. Smith, ed.), (New Orleans, USA), pp. 1952–1959, IEEE Computational Intelligence Society, IEEE Press, 5-8 June 2011.

[162] N. A. Gershenfeld and A. S. Weigend, *The Future of Time Series*. In Time series prediction: Forecasting the future and understanding the past, Gershenfeld A. N. and A. S. Weigen, eds, pp 1-70., 1993.

[163] C.-L. Huang and C.-Y. Tsai, "A hybrid SOFM-SVR with a filter-based feature selection for stock market forecasting," *Expert Systems with Applications*, vol. 36, no. 2, Part 1, pp. 1529–1539, 2009.

[164] R. G. Pajares, J. M. Benítez, and G. S. Palmero, "Feature selection for time series

forecasting: A case study," *Proceedings of the International Conference on Hybrid Intelligent Systems*, pp. 555–560, 2008.

[165] H. Yoon and K. Yang, "Feature subset selection and feature ranking for multivariate time series," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 9, pp. 1186–1198, 2005.

[166] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E*, vol. 69, p. 066133, Jun. 2004.