Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Acceleration of a bacterial metabolic simulation using neural networks with optimization techniques

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial

AUTOR: Javier París Uhryn

TUTOR/ES: Dr. Alfonso Rodríguez-Patón Aradas

2019

Acknowledgment

Me gustaría expresar mi agradecimiento hacia todo el equipo del Laboratorio de Inteligencia Artificial de la UPM, donde me han acogido, apoyado y ayudado durante toda la duración de este proyecto.
Y en especial a Elena, por la idea inicial del proyecto y por haberme ayudado en tantas ocasiones.

Abstract

Computational biology is a leading field that requires the combination of biological knowledge as well as a high level of technology and computing, which promotes a joint work of both areas for its correct evolution.

A branch of this discipline is based on the use of simulators, which have been boosted in the last decades thanks to hardware improvements, allowing to create more complex, truthful and efficient systems. One kind of these simulators are the agent-based models, which work with individual cells and their interactions to form tissues and colonies. Since a few years ago, software advances in the field of Artificial Intelligence are opening new doors to these simulators to allow improvements in the analysis of the data produced as well as in their internal execution.

GRO is a multicell colony simulator that represents each cell on a bacterial colony with their own space and internal data. This simulator is thought to represent huge microbial communities as the *microbiota*, taking into account the spatial distribution and the individual behaviour of each cell. To simulate this, it is necessary to take into account the metabolism of each cell and the relations between them individually, with colonies of more than ten thousand individuals.

In this work we try to find a system based on Artificial Intelligence and optimization methods to approximate a biological calculation such as the metabolism of cells, understanding metabolism as the different internal chemical reactions of a cell that consume and secrete various metabolites to obtain biomass. This metabolism has several theoretical models, among which we find the model *FBA* or *Flux Balance Analysis*. This model allows to calculate in few time the metabolism of a cell. The execution time of the FBA is still too high to be used in an agent-based simulator, where thousands of cell metabolisms would be simulated in every step.

The solution proposed in this project is to use *Artificial Neural Networks* to simulate the execution of a FBA model, and its optimization. This solution will allow in a limited space of time, to obtain a quickly and accurately approximation for the results of the metabolism.

As future work, it is contemplated the use of these optimization systems on neural networks to approximate a metabolism model different from the FBA, getting to use, for example, other metabolic models or even real experimental data to model the individual metabolism of each cell.

Key words
Computational biology, Neural Networks, Deep Learning, Heuristic algorithms, Op-

timization algorithms, Flux Balance Analysis, FBA, GRO, metabolism, microbiota, Ant Colony Algorithm, Evolutionary Algorithm, Simulated Annealing Algorithm.

Resumen

La biología computacional es un campo puntero que requiere de la combinación de conocimientos biológicos a la vez que un alto nivel tecnológico e informático, lo que promueve un trabajo conjunto de ambas áreas para su correcta evolución.

Una rama de esta disciplina se basa en el uso de simuladores, que se han visto impulsados en las últimas décadas gracias a las mejoras del hardware, permitiendo crear sistemas más complejos, veraces y eficientes. Un tipo de estos simuladores son aquellos basados en agentes, los cuales representan las células individualmente y sus interacciones a la hora de crear tejidos y colonias. Desde hace pocos años los avances de software en el ámbito de la Inteligencia Artificial están abriendo nuevas puertas a estos simuladores para permitir mejoras tanto en el análisis de los datos producidos como en su ejecución interna.

GRO es un simulador que representa individualmente cada célula de una colonia bacteriana con su propio espacio e información interna. En un futuro se quiere que este simulador logre representar grandes comunidades microbianas como la microbiota, considerando la distribución espacial de cada célula así como su comportamiento. Para simular esto, es necesario considerar el metabolismo de cada célula, trabajando con colonias de decenas de miles de individuos.

En este trabajo se intenta encontrar un sistema basado en Inteligencia Artificial y métodos de optimización para aproximar un cálculo biológico como es el metabolismo de las células, entendiendo como metabolismo las distintas reacciones químicas internas de una célula que consumen y segregan diversos metabolitos para obtener biomasa. Este metabolismo cuenta con diversos modelos teóricos, entre los que encontramos el modelo *FBA* o *Análisis de Balance de Flujo*, que permite obtener medidas de forma rápida para el metabolismo de ciertas células. El tiempo de ejecución de dicho sistema es aún excesivamente alto como para poder usarse en un simulador basado en agentes, donde miles de metabolismos bacterianos tienen que ser simulados en cada iteración.

La solución alcanzada en este proyecto es el uso de *Redes Neuronales Artificiales* para simular la ejecución de un modelo FBA, y la optimización de dichas redes, permitiendo en un acotado espacio de tiempo obtener de forma rápida y precisa los resultados del metabolismo.

Como trabajo futuro se contempla el uso de estos sistemas de optimización sobre redes neuronales para poder aproximar un modelo de metabolismo diferente al FBA, llegando a usar otros sistemas metabólicos o incluso datos experimentales reales para modelar el metabolismo individual de cada célula.

Palabras clave
Biología computacional, Redes de Neuronas, Aprendizaje Profundo, algoritmos basados en heurísticas, algoritmos de optimización, Análisis de balance de flujo, FBA, GRO, metabolismo, microbiota, algoritmo de colonia de hormigas, algoritmos evolutivos, algoritmo de recocido simulado

Contents

Acknowledgment **iii**

Abstract **v**

Resumen **vii**

1 Introduction **1**
1.1 Project introduction . . . 1
1.1.1 Motivation . . . 1
1.1.2 Idea . . . 2
1.1.3 Work done . . . 2
1.2 Document structure . . . 3
1.3 Concepts . . . 3
1.3.1 Biology concepts . . . 3
1.3.2 Computer science concepts . . . 4

2 State-Of-The-Art **7**
2.1 FBA as a metabolic model . . . 7
2.2 Biology Simulators . . . 9
2.2.1 *AbM* . . . 9
2.2.2 GRO . . . 10
2.2.3 Other simulators using FBA . . . 11
2.3 Neural Networks . . . 11
2.3.1 Introduction to Neural Networks . . . 11
2.3.2 Neural Networks nowadays . . . 12
2.3.3 Deep Learning . . . 13
2.3.4 Optimization algorithms to find the best neural networks hyperparameters . . . 13

3 Problem definition **15**
3.1 Motivation and Novelty . . . 15
3.2 Research questions and hypothesis . . . 17

4 Proposed solution **19**
4.1 Problem specification . . . 19
4.2 Methodology . . . 22
4.2.1 Tools . . . 22
4.2.2 Prior decisions . . . 22
4.3 FBA hypothesis validation . . . 25
4.3.1 Time . . . 25
4.3.1.1 FBA Execution Time . . . 25
4.3.1.2 Neural Network Execution Time . . . 27
4.3.1.3 Conclusions . . . 30
4.3.2 Accuracy . . . 32
4.3.3 Conclusions . . . 36
4.4 Optimization hypothesis validation . . . 37
4.4.1 Training requirements . . . 37
4.4.2 Metaheuristics . . . 40
4.4.2.1 Ant Colony . . . 41
4.4.2.2 Simulated Annealing . . . 47
4.4.2.3 Evolutive . . . 50
4.4.2.4 Search methods . . . 55
4.4.2.5 Conclusions . . . 59
4.4.3 Exhaustive methods . . . 60
4.4.3.1 Direct methods . . . 60
4.4.4 Final Implementation . . . 68
4.4.5 Conclusions . . . 72

5 Conclusions **73**
5.1 Conclusions . . . 73
5.2 Future work . . . 74

A FBA specification **77**
A.1 *E. coli* Core . . . 77
A.1.1 Model Summary . . . 77
A.1.2 Reactions . . . 77
A.1.3 Metabolites . . . 80
A.1.4 External metabolites . . . 80
A.2 *E. coli* . . . 80
A.2.1 Model Summary . . . 81
A.2.2 External metabolites . . . 81
A.3 *Salmonella* . . . 81
A.3.1 Model Summary . . . 81
A.3.2 External metabolites . . . 81

B Results **83**
B.1 Number of connections 83
B.2 Time 84
B.2.1 *Core* 84
B.2.2 *Salmonella* 84
B.3 Accuracy 85
B.3.1 *Core* 85
B.3.2 *Salmonella* 87
B.4 Training 89
B.4.1 *Core* 89
B.4.2 *Salmonella* 90

C Neural Network implementation using Keras **91**

D FBA implementation using Cobra **93**

Bibliography **95**

List of Figures

2.1 Quick FBA explanation [1] 8
2.2 Biological simulator GRO execution example [2] 10
2.3 Example of a neural network structure [3] 12

4.1 Metabolism simple example 19
4.2 Examples for different FBA executions in *E. coli* metabolism 20
4.3 Execution of the FBA for a random input with 25 repetitions 26
4.4 Execution of the FBA for a null input with 25 repetitions 26
4.5 Execution of the FBA with null input metabolites 27
4.6 Multiple execution of 50 times of the FBA for a random input with 25 repetitions each input 28
4.7 Time execution for a simple Neural Network 28
4.8 Time execution for a complex Neural Network 29
4.9 Time execution depending on the number of connections 30
4.10 Time execution depending on the number of connections 31
4.11 Part of the dataset to train and test the neural networks 33
4.12 Error and time estimated for random networks 34
4.13 Results modifying parameters 35
4.14 Error curve for training in standard neural network 38
4.15 Different error curves for random networks 38
4.16 Execution of 100 random neural networks to measure the error decreasing ratio. 39
4.17 Results for Generic Ant Colony algorithm 43
4.18 Results for Generic Ant Colony algorithm 44
4.19 Results for Layer Ant Colony algorithm 45
4.20 Result for Layer Ant Colony algorithm with normal distribution . . . 46
4.21 Generic Simulated annealing error evolution 49
4.22 Decreasing Simulated annealing error evolution 50
4.23 Execution of Evolutive algorithm during a half day with a population of 25 individuals 52
4.24 Multiobjective Evolutive algorithm execution 54
4.25 Forward Search error evolution 56
4.26 Inverse Search error evolution 58
4.27 General structure of neural networks for final solution 61

4.28 Execution for all the metabolism. Red dots indicates number of connection per network. Green line represents the error measured in *Biomass* metabolite 66
4.29 Execution for all the metabolism. The red dots indicates the number of connections. Green line represents the evolution of the best solutions visited for each l 67
4.30 Repetitive executions for every model along different sizes. These values has been calculated repeating 10 times each hidden layer size with **random** datasets each time 71

A.1 *E. coli* Core metabolism summary 77
A.2 *E. coli* metabolism summary 81
A.3 *Salmonella* metabolism summary 81

B.1 Mathematical number of connections 83
B.2 Core mean execution 84
B.3 Salmonella mean execution 84
B.4 Error and time estimated for random networks 85
B.5 Results modifying parameters 86
B.6 Error and time estimated for random networks 87
B.7 Results modifying parameters 88
B.8 Execution of 100 random neural networks to measure the error decreasing ratio. 89
B.9 Execution of 100 random neural networks to measure the error decreasing ratio depending on the number of connections. 90
B.10 Execution of 100 random neural networks to measure the error decreasing ratio. 90

List of Tables

4.1 Final results for *Direct Methods* execution 65
4.2 Best individuals for *Direct Methods* execution 65
4.3 Results for final implementation along 100 random networks for every case 70
4.4 Worst solution expected for final implementation 70

List of Algorithms

1 Ant Colony standard algorithm 41
2 Algorithm to create the new individual 42
3 Simulated Annealing algorithm 48
4 Population Based Evolutive algorithm 51
5 Individuals Crossing 52
6 Multiobjective Evolutive algorithm 53
7 Forward Search 56
8 Inverse Search 57
9 Two Layer Direct Algorithm 62
10 Three Layer Direct Algorithm 64
11 Final implementation 69

Chapter 1

Introduction

1.1 Project introduction

1.1.1 Motivation

Synthetic Biology is a term that has been appearing in the last decades in the scientific world. This field is an area where the technological improvements in the computer science field has been used in the biological world. This work is a perfect example of how the Computer Science, more precisely the Artificial Intelligence, helps to overcome some obstacles in the simulation of certain biological characteristics.

This work arose from the need to improve a program called GRO, software that is currently in phase of implementation and improvement in the *Laboratorio de Inteligencia Artificial* in *Universidad Politécnica de Madrid*.

GRO software is a simulator based on individuals, or agent-based model *AbM*. This means that, in this simulator, we can represent a colony of bacteria cell by cell.

GRO works using its own specification language. In this language, the user can specify metabolism of each kind of cell, understanding as metabolism the inner reactions that consume and expel metabolites from every cell in order to get biomass.

But the GRO metabolic model is very basic, and it does not allow complex connections between different metabolites,. Also it requires a hard work from the user to write the entire metabolism for each experiment.

The idea was to find a model that could approximate the behaviour of the metabolic reactions inside a cell.

For this proposed, we chose FBA or *Flux Balance Analysis* as a theoretical model to approximate the behaviour of a cell. This theoretical model is explained more in detail in the next section.

Summarizing, we have a simulator where we can represent cell by cell a whole

colony, and also we have a metabolic model that can be implemented on each of these cells to simulate more accurately the real metabolic behaviour.

The problem now, and this is where the whole project was born, is that this FBA model requires so much time to compute that is not efficient to use it inside each cell in every *time step* of the simulation.

1.1.2 Idea

The main idea of this project was to create a reliable method that we could use as a *black box* to solve the FBA problem in a small amount of time.

Summarizing the bases, we wanted a method that could **learn from data** taken from FBA executions. This method could be used as a ***black box*** where we give the inputs and we can obtain accurate outputs in **few time**.

The idea which we arrived with (not the first one, but the most promising one) was to implement a **neural network** that would fit with the data that we executed from FBA.

This way, we could implement this neural network into GRO once it is trained, and we could execute the metabolism cell by cell in an individual, fast way.

We want to create a neural network that fits for different FBA metabolisms, as the *Escherichia coli* metabolism and the *Salmonella* metabolism among other possibilities. And we know that these different metabolisms, even when their main part could be similar, are very different one from each other. They could even change on the number of metabolites that they use as input. So this makes very difficult to create a useful neural network that works for every one of the metabolisms.

To this different metabolisms that we will have to deal with, we have to add also the possibility that the user could want to modify one already existed metabolism, as a result of a possible mutation.

This means that we can not create a database of *already trained neural networks* for each metabolism, because it will not be complete as there will be always new species that could be introduced, and because the user can create new metabolisms in every moment.

1.1.3 Work done

Our target for this whole project is to **implement** an algorithm that, using **optimization and heuristic methods**, we could arrive to **fast and accurate** network structures in **few time**. This structures for **neural networks** will be used as a *black box* of metabolic systems that fits with **specific** metabolic data.

1.2 Document structure

This work is framed as follows:

- The current chapter **Introduction** summarizes the whole project and introduces the structure and some initial concepts of it.
- Chapter 2 reviews the **State of the Art** of the field of study around this work, both the computing and the biological part.
- In Chapter 3 we explain the **Problem definition**, where we summarize the hypothesis and assumptions that we will use as the bases for our project.
- Chapter 4 contains the whole **Implementation** explanation of this project. In this chapter all the experiments are explained and their results are discussed. This chapter is divided in:
  - **Problem specification:** In this section we found the problem explanation, as it desired scope.
  - **Methodology:** In this section we found the main assumptions and prior rules chosen to be the guide for our project, in order to simplify it. And we found the different tools used along the project.
  - ***FBA* hypothesis validation:** In this section we found the demonstration of the first hypothesis of the work, that is defined in the problem definition section. This hypothesis is based on the motivation of the use of neural networks to approximate *FBA*.
  - **Optimization hypothesis validation:** In this section we present all the results of the different heuristic and non heuristic methods that we have implemented in order to optimize the neural networks structures. We also present the final solution proposed.
- In the **Conclusions**, we discuss the final solutions and results. We also suggest some future work in the field.

1.3 Concepts

In this section are introduced some concepts that will be needed for the whole understanding of this project. The detailed explanations for these concepts are in the next chapter 2.

1.3.1 Biology concepts

The main idea and focus of this project is from the computer science point of view, but we will need some biological concepts to understand properly the motivation of this work.

Synthetic Biology

Synthetic biology is an early branch of biology and engineering, that has joined the effort of both biologist and computer scientist.

Inside this field we can find a huge amount of researches based on the use of computers to simulate some biology systems, in order to use the advantages of the computation.

These methods allow biology to simulate some very expensive or long processes in a fast and cheap way.

A common practice in this field is the us of simulators to represent various biological experiments or processes.

Flux Balance Analysis

The *Flux Balance Analysis* (*FBA*), is a theoretical model that simulates the metabolism of a single cell of a determined species [4].

It is calculated in a deterministic way solving an equation system and an optimization problem. These equations vary from one species to another.

This allows to use the same solving system to different metabolisms.

1.3.2 Computer science concepts

These are the main technical concepts related with computer science theory that we are going to talk about throughout the entire project.

Artificial Neural Networks

Artificial Neural Networks are a computational model that is based on a group of connected *nodes* or *neurons*.

These nodes are connected with each other. The input information travels and is modify by the different weights of these connections.

This is used, in the first place, to train the network with known data, modifying the network's weights to fix with our dataset. And once a network is trained, the network will give an approximated value for a given input, even when it has not seen it before.

These models are used to solve many problems in pattern recognition. They have been prove to be very powerful and accurate. The kind of problems that they solve ranges from *expert systems* to *image recognition* within many others.

Optimization algorithms

These algorithms are a huge family that are based on obtaining the best solution possible to a problem.

We will have a *measure* for every *solution* possible. This measure will say if an individual is "better" than other. So, the algorithms will search between different individuals and they will arise with the best that they could find.

Inside these algorithms we can remark a huge amount of them as *Heuristic algorithms*, that follows a determined *Metaheuristic*.

A Metaheuristics is an stochastic method to solve a kind of computational problem for which there are not an specific algorithm that solves it [5].

Chapter 2

State-Of-The-Art

2.1 FBA as a metabolic model

A metabolic model is needed to represent the behaviour of a cell in a biological simulator, understanding *metabolism* as the group of reactions (physical or chemical) that take place inside a living cell. The input for these reactions are *metabolites*, which consist in different nutrients or molecules that the cell uses for its living functions.

The output of this metabolism will be the growth achieving for the cell after those reactions, and the metabolites that the cell expels after it. Those expelled metabolites could be because the cell has not used them totally or because they are leftovers from some internal reactions [6].

Each FBA is based on a stoichiometric matrix and some constraints. The matrix stores the data for the different reactions, linking the different reactions in the metabolism with the metabolites involved in them. This means that in the matrix is written the number of each kind of metabolite that will be used in a reaction, an its result metabolites. The matrices and constraints vary for every species, and are related with the maximum amount of every reaction that could occur in a cell.

The FBA is calculated by solving an equation system where each equation represents the flux of every metabolite, and the different variables are the reactions involved. Solving this system we can get the number of reactions that have taken place in the cell, an so, the number of metabolites that have been consumed or produced, and the biomass obtained [1].

Just for the reader to have an approximate notion, the most simplest model that we have worked with, that is the metabolic model for the carbon in *E. coli*, has 72 metabolites and 95 reactions related. That means that this equation system will have 72 equations and 95 variables. But only 7 of this 72 metabolites are *external* metabolites, so our data will have just 7 variables. These information could be seen in more detailed in the annex A.

This helps us to have an idea about how complex the resolution of this operation would be.

This system has rarely a single solution. The FBA assumes that the cell will "choose" the reactions that increase the growth of it. So, FBA executes an optimization algorithm to calculate which reaction path is more beneficial for the cell.

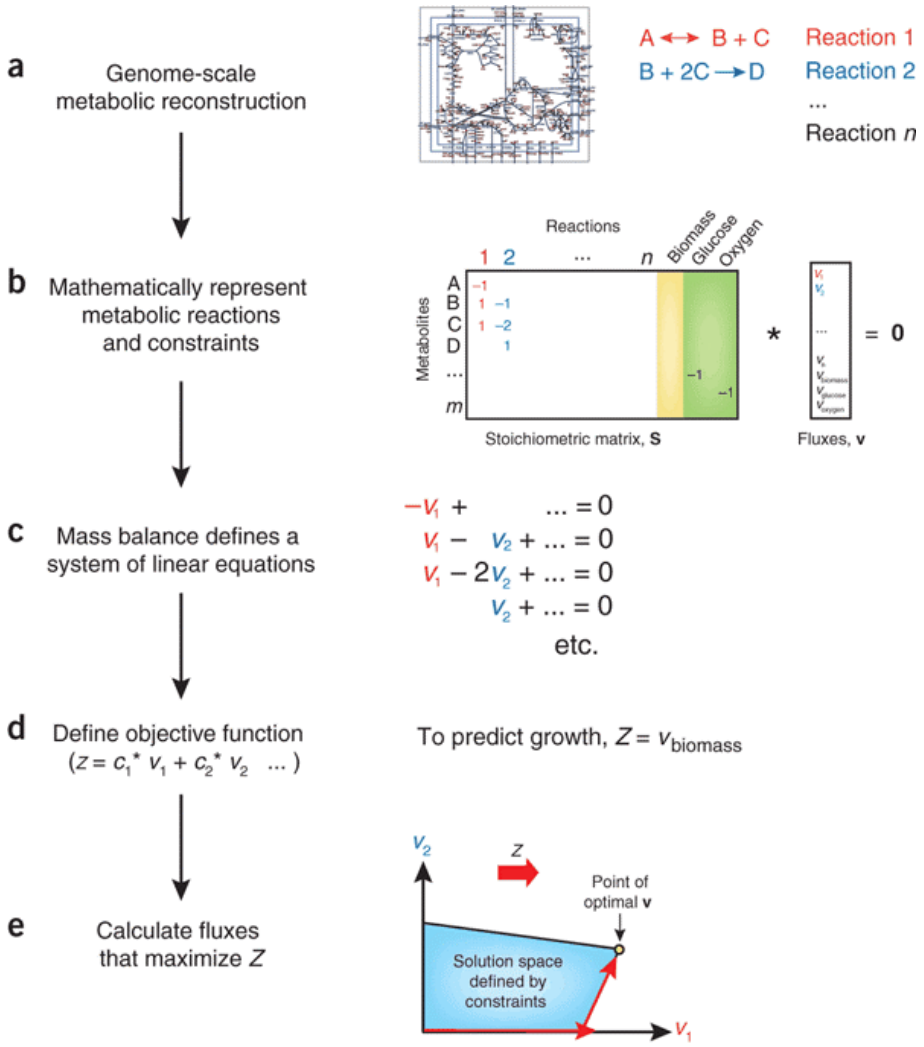


Figure 2.1: Quick FBA explanation [1]

So, this FBA method allows to approximate the metabolic reactions following some assumptions as:

- the cell is in an stable state, where the internal metabolites stays constant because the production and consumption rates match.
- the cell will optimize the metabolites use by choosing the reactions that will maximize its own growth.

These assumptions could not be seen as very realistic. But the metabolism itself is a so complex process that we need some faster models to approximate it.

We need to know that FBA is a theoretical model that allows us to see the metabolism as a *black box*. We set the concentration of the input metabolites around a cell, and FBA gives us the growth ratio and the metabolites that will abound the reactions.

2.2 Biology Simulators

The use of simulators in synthetic biology has played a very important role in the reduction of costs, both in time and money, for biological experiments [7] [8].

The simulators base their performance on the implementation and verification of theoretic models that represent real behaviors. The more adjusted and the more accurate a model is, the more realistic and reliable should be [9].

Some simulators employed in synthetic biology focus on the detailed study of the processes that are carried out intracellularly.

However, the models created according to this approach present difficulties while studying collective behaviors that take into account the interactions between cells.

The systems are of great complexity, especially in the multicellular case in which there is a large number of individuals with a behavior exposed to continuous variations, as it happens in the natural environment. Regarding the design of the model that best fits the system one wants to simulate, we must highlight two types: Models Based on Differential Equations (*DEs*), and Models based on Individuals (*IbM*, also known as Agents based Models or *AbM*).

2.2.1 *AbM*

Within the biology simulators, we find the huge family of *AbM*. Those simulators recreate cell by cell a whole experiment, storing individual information of each agent.

Agent-based models allow to take into account the behavior of each individual present in the system, which is an independent entity or *agent*. In addition, they allow to study the collective behaviors that emerge from all individual behaviors. In this way, they give a simulation of the behavior of the bacteria more precisely. They also allow to study the *spatial distribution* and local interactions of the bacterial colonies, which has been proofed to be a critical aspect for the behaviour of the colony.

There are many simulators that implement agent-based models, as: BactoSim, BacSim, GRO, CellModeler, iDynomics, CeCe. Any of them has been developed with the same purpose, to describe and represent the behavior of an independent

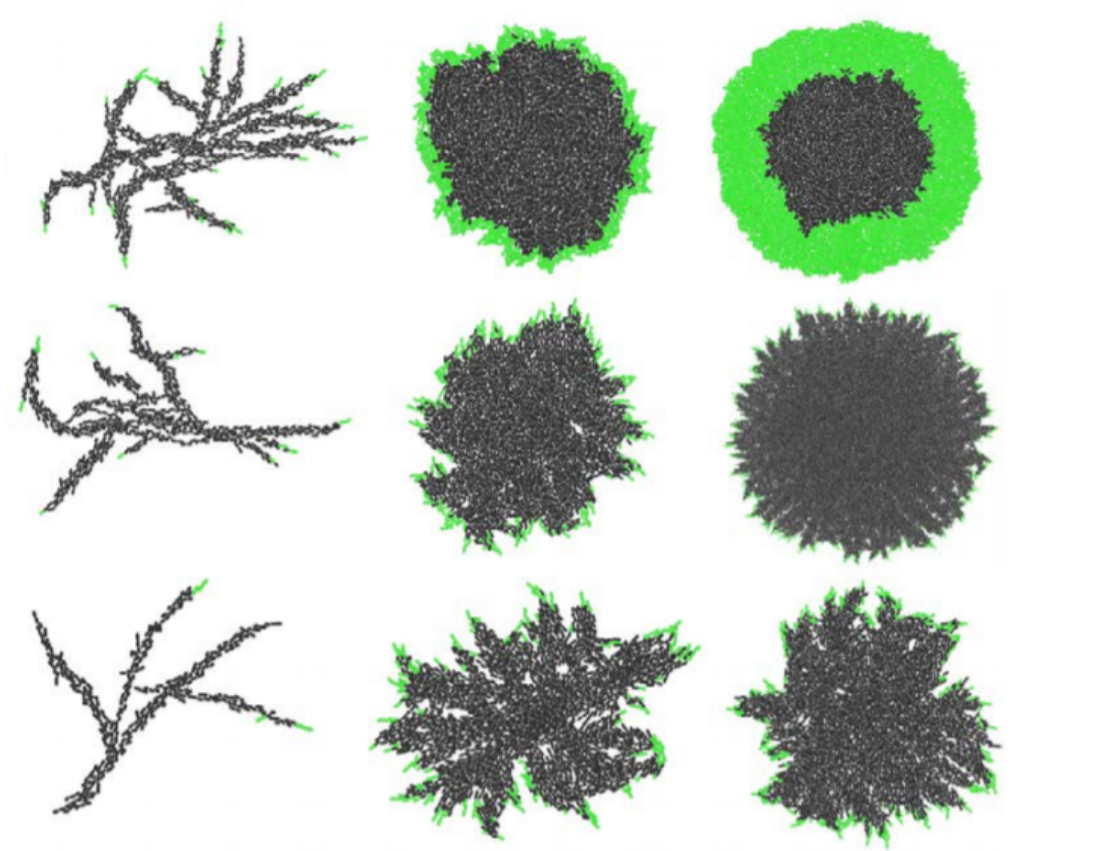


Figure 2.2: Biological simulator GRO execution example [2]

agent. They present differences in characteristics such as the scope of the simulations, the levels of abstraction allowed or their scalability with the number of individuals.

2.2.2 GRO

GRO is the chosen *AbM* simulator for this work. This framework is developed in C and C++ for simulating multicellular biocircuits in 2D with a visual GUI that displays the evolution of the colony.

GRO has been extended from its first version [10] at LIA-UPM, and at the time of writing this work is under developed for improvement by the researchers in this laboratory. The code could be found in: [11].

GRO metabolism

In the current version of *GRO*, the metabolism is too simple to simulate reliably the huge complexity of a multicellular metabolism.

At the moment, the cells in the simulator executes their metabolism independently of the concentration of metabolites in their medium. If the metabolite is

available in the medium it is absorbed with a specific rate (and without leftovers) independently of the concentration. Also it is not possible to relate the absorption or emission of different metabolites between them, or with the growth rate.

Those functionalities are necessary to simulate correctly complex systems as *quorum sensing* or *microbiota*.

2.2.3 Other simulators using FBA

There are different biological simulators for representing bacterial or cell colonies, and their metabolism and interactions.

We will mention in this section two of them, that are *COMETS* [12] and *BacArena* [13].

Both simulators are already using *FBA* as their main model of metabolism for their simulations, but each of them with different approaches.

In the first case, *COMETS* uses the hypothesis that the metabolism of a community could be simulated by optimizing the individual metabolisms. So the metabolism is calculated using FBA for the whole colony.

For the second simulator, *BacArena*, they calculate FBA in all the cells and step by step, as we are planing to do. But they calculate it explicitly, spending the time that FBA requires in each execution.

This last point of view is more similar to our approach, but we will use alternative methods that allow different metabolites. Our method will be faster to compute that FBA is, and it will also allow to use other metabolic models or more complex FBA models.

2.3 Neural Networks

2.3.1 Introduction to Neural Networks

Artificial Neural Networks are a computational model that are applied to approximate functions to data. These neural networks are vaguely inspired by the biological model of neurons in animals.

Their study started some decades ago, within the first techniques of the Artificial Intelligence. But their use has experienced a giant explosion in the last years thanks to the advances in hardware. This has allowed to work with larger datasets and bigger networks.

Along this work we will talk about *Feedforward Fully Connected Neural Networks*. These are the most simple neural networks in the literature, and we are going to work with them because of their simplicity and powerful [14].

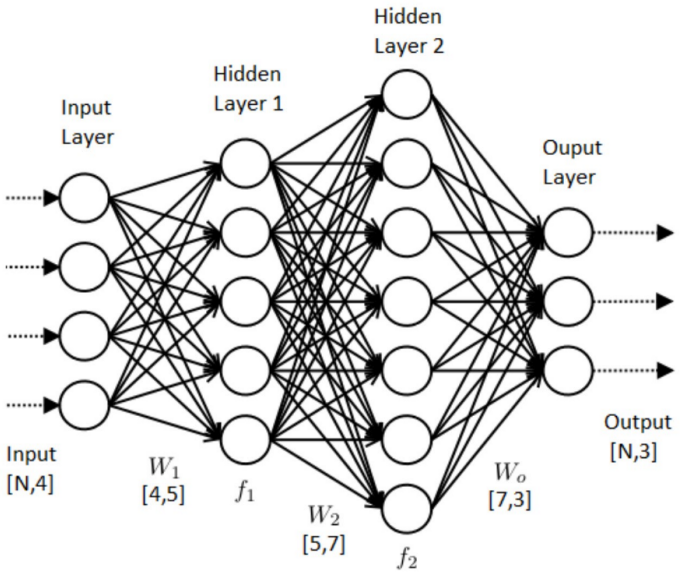


Figure 2.3: Example of a neural network structure [3]

These neural networks are based on several *neurons* or *cells* connected to each other by edges with weights. The learning is based on the adjustment of this weight net, to fit the final output of the net to the desired output.

The cells are divided in layers. The cells that belongs to the same layer share the same neighbours and some characteristics, as the *activation function*. Those layers are divided in *input layer*, *output layer* and a series of *hidden layers* that may vary in number and size.

There are plenty of theory behind the use of these networks that could be reviewed in the bibliography [15] [16]. What we need to know about them to understand this work is that they could be seen as a *black box* where you set an input of N data and you receive an output of M values. This final result is calculated in very few time by executing different operations between the layers.

This result will be close to the desired output for this input if the network is well trained (even when the network has never seen this exact input before). This training is the process that most time requires, and it usually needs a big amount of data to approximate it accurately.

2.3.2 Neural Networks nowadays

We could say with certainty that this is the field that has evolved more within the classification and regression methods in the last years.

Neural networks has been studied since the beginning of the Artificial Intelligence [14], because of their potential power and their simplicity. But it has not been until

few years ago when those optimization techniques have become the spearhead of every Artificial Intelligence project. The advances in hardware and software (as the parallelism or the use of cloud computation) have been overriding to make the neural networks a powerful tool to approximate any kind of data.

Using from classification problems and pattern recognition to image and video analysis, the neural networks predominate in the actual computer science research.

2.3.3 Deep Learning

Deep Learning is related with the learning of specific connectionism systems from data [17].

The use of neural networks in deep learning is highly extended, being deep neural networks the most common system for this goal. This deep neural networks are based in the same ideas of the neural networks mentioned, but with the characteristic that they have several hidden layers (generally more than two). This gives the network more power to abstract the information and be allowed to better approximate the results.

But deep neural networks have some several problems that traditional neural networks does not have. Between them there is the *vanishing gradient* problem [18], and other problems as the exponential increasing of possible structures.

2.3.4 Optimization algorithms to find the best neural networks hyperparameters

The complex structure that a neural network needs, together with the amount of different hyperparameters that they have, create a difficulty when we want to work with them. That is because, commonly, finding this data requires a lot of time and testing.

The networks structure is usually related to the problem that needs to solved. There are a lot of already existing network structures that are known to fix with one specific kind of problem, as convolutional networks for image recognition.

When we work with new kinds of datasets that are not related to anything along the literature, we have to search for the most appropriate structure for them.

This is where the optimization algorithms appear to help in neural network designs [19]. This combination allows to use the powerful tools of optimization processes to improve the neural network structure and hyperparameters in order to satisfy the needs of the users.

This means that neural networks are optimized to approximate better the data that we have.

Chapter 3

Problem definition

The **main goal** of this project is to build a reliable and fast method that simulates accurately the metabolism of a single cell. We want to do it using as few data as possible and in a small amount of time.

3.1 Motivation and Novelty

In biology simulation field, the aim is to achieve a realistic model that approximate biological logic rules.

This triggers a problem between the biological and the computation point of view. The faster a simulator is, the less reliable it will be.

This leads us to simplify the simulation systems by means of assumptions or omitting some variables or parameters in order to get a more faster, useful simulator. This could be seen as paying in reliability to obtain usability.

Motivation in GRO

Our simulator GRO does include some metabolic functionalities in the simulations using a separate growth rate individual for each cell. This growth rate has the capability of changing during the process depending on the environment of the cell.

However, these methods are still very simple to be realistic.

The solution would be to use a fast and highly reliable metabolic system that allows to calculate the cellular metabolic response to environmental conditions in a small amount of time. This will allow the user to get more realistic solutions.

It is also interesting to simulate different species with different metabolisms, or even same species with small variations (results of mutations or regulation) and get different values for their growth and behaviour.

This idea is useful for GRO as it is for any other simulator that needs an individual metabolic model.

Motivation to use FBA as data

FBA is a mathematical approximation for the metabolism of a cell that is based in several assumptions, as it is explained in section 2.1. Even though FBA is just an approximation for a real behaviour, it is fast and easy to get a huge amount of data from it. This data could be used to recreate the whole metabolism of a cell. FBA has several benefits that support using it rather than other kind of data, as:

- It is fast to calculate.
- It is a mathematical deterministic process, so it is always possible to get more data as realistic and reliable as the first one.
- It is a chemical approximation, so even when the assumptions are not satisfied, it would always be a correct chemical solution.

Motivation to use neural networks

The accuracy of the methods that we use in the simulators are capital, as it is the amount of time required to achieve it.

There has not been founded works related with the approximation of FBA using machine learning approaches, in particular neural networks.

As we will demonstrate in upcoming sections, there are some advantages that we can achieve using these methods. Those are getting calculations in a short time and with an small error.

Also, the use of neural networks allows to extrapolate these whole project to other kind of data apart from FBA, as other metabolic model or even experimental data.

Motivation to optimize the network structure

The amount of time spent in the calculations of this data is very important, because we will use this methods in every cell at every single step. So the time required will be highly multiplied.

With the optimization of the neural network, we try to get the fastest network that gives solutions in small space of time, but without losing accuracy.

The practices of optimizing the structure of neural networks is a highly extended praxis along the deep learning community. In this work we use some of this already existing methods, especially optimization algorithms.

Novelty

There are other simulators that include already with a FBA based metabolism to simulate a cell colony.

The advantages of this approach are that we will get a good spatial and temporal realistic model of it. And we will do it saving a huge amount of time for each execution, allowing our simulations to run faster.

We have not seen over the literature (to our knowledge) other simulators that use FBA (or other approximation of metabolism) to recreate a metabolism in execution time and, at the same time, using faster an time limited methods to calculate those values.

3.2 Research questions and hypothesis

Research questions

The research questions that motivate this work are:

- **RQ1:** Is it possible to approximate the metabolism of a cell using fast and reliable algorithms?
- **RQ2:** Can we find a fast and accurate neural network that fits with a proper problem, in a limited amount of time?

Each of these questions motivates the two different parts of this work. The first one is focused on proving that neural networks solve our problem, while the second one is more focused on working with neural networks and optimizing them.

Assumptions

Some of the hypothesis that we work with are not involved in this work domain, so we will assume them as true. However, they must be demonstrated, or at least checked, before assuming the truth for this whole work. These assumptions are:

- **A1:** Individual FBA is an accurate model to approximate the metabolism of a cell.
- **A2:** The computer used to run and execute all these experiments spends a proportional time in all these different task as any other machine where this project is expected to run into.

Hypothesis

Assuming as true the points mentioned above, we will try to demonstrate along this work the following hypothesis:

- **H1:** FBA is accurately approximated by a neural network.

- **H2:** The neural network is capable of being executed in fewer time than FBA.

- **H3:** We are able to get an optimized neural network that fulfilled **H1** and **H2** in a limited time.

We have proposed some characteristic values for the validation of those hypothesis, in order to have a general idea of how much improvement we should expect from the final solution.

The accuracy and executing time of the neural networks depends on the requirements of the user. Even so, we have set values to those hypothesis in order to measure the worth of the final solution.

We will assume as a valid solution a neural networks that achieves a maximum error of **10%**. This network would be fast enough if it is able to run **100 times** faster than the execution of FBA.

Those are the requirements that we have set for the **hypothesis 1 and 2**. They are validated in section 4.3.

Finally, the time needed to get a solution was a user limit. This method should execute before an experiment with a simulator, so it could not take too much time since the user run it. We need a solution in short time.

We assume that a maximum time to get this solution will be **10 minutes**.

With these we will probe the validation of the **hypothesis 3**, that is validated in section 4.4.

Chapter 4

Proposed solution

In this chapter we find all the detailed information about the implementations that we have made along this project. We will explain every step, explain the motivation of each one and show their results.

4.1 Problem specification

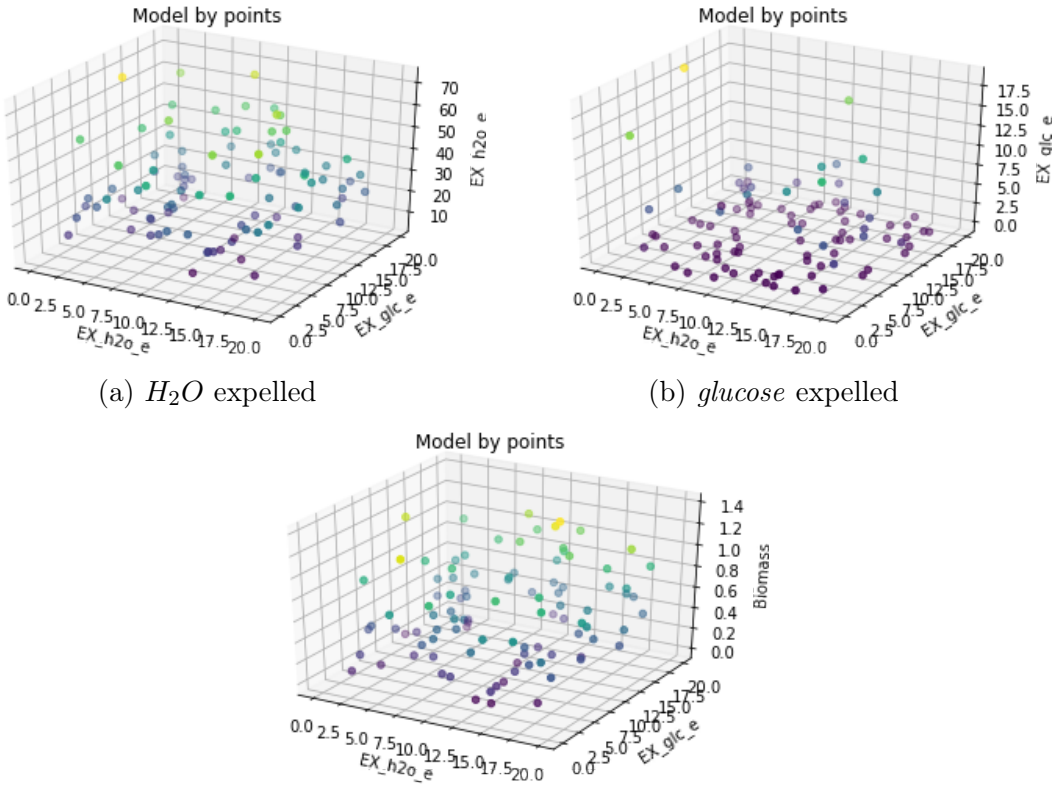
First of all, we are going to explain what is the main idea that we are going to work with.

As we can see in the simple schema shown in the figure 4.1, we have a metabolic model that will work as a *black box*. This means that we will not care about what is computed inside *bacteria metabolism* but about which inputs we give it and which outputs we receive.

Figure 4.1: Metabolism simple example

We have the FBA model, that calculate this function. This model takes some metabolite concentrations as input, and it gives the spare metabolites that the cell will expel. These calculations vary depending on the specie that we are working with, and depending on the specific restrictions that we have chosen for it.

In the examples of the figure 4.2 we see some FBA executions for the *E. coli* metabolism. We see how the glucose is consumed almost completely in all the situations, while the water is expelled in high quantity.



(a) H_2O expelled

(b) *glucose* expelled

(c) *Biomass* produced

Figure 4.2: Examples for different FBA executions in *E. coli* metabolism

The main idea is to reproduce this calculations but using neural networks instead of an FBA model. These networks will be trained with the data calculated by *COBRApy*. This way, we can have a system that follows the schema in 4.1 but much faster.

Final solution expected

The final solution that we want to achieve is a system that is capable of finding a neural network that fits a specific dataset. Those datasets could be chosen before a simulation execution, so this neural network must be generated and trained in few time (we can place the limit in 10 minutes).

A possibility would be to have every metabolism with an already optimized trained network. But the solution expected also has to find an optimized network for a dataset that has not been seen yet. Those new datasets could be the result of some little modifications to already existed metabolisms.

So, we want to implement a system that is able to give an optimized trained network that is accurate and fast to execute for some given metabolic dataset. And this system needs to give these solutions in a limited and short amount of time (obviously, the time needed to create the dataset will not be included in the execution time of this system).

Solution scope

We will focus this work on the theoretical part of the project, more than on the implementation part.

This means that we will try to probe the veracity of the hypothesis and we will demonstrate that it is possible to build a system with these specifications.

But we will not focus on the specific improvements that this system will need once it is built. We leave this part as future work.

4.2 Methodology

4.2.1 Tools

In this section we are going to expose the different tools that we have used for all these experiments:

- **Anaconda-Python** We have used *Python* as the main (and only) programming language for this whole project. The version used has been **Python 3.5**. We have used the following libraries of *Python* along the project:
  - **tensor flow 1.10.0** neural network framework [20].
  - **keras 2.2.2** sublibrary from *tensor flow* that implements a *wrapper* over the neural networks to make easier to work with them [21].
  - **COBRApy 0.15.1** *Python* library from *cobra*. We get all the *FBA* data from the functions of this library [22].
  - **numpy 1.14.3** mathematical framework [23].
  - **pandas 0.23.0** data framework [24].

  Apart from the *Python* libraries, we have also used the *Anaconda* environment to work with.
  - **spyder 3.3.1** programming environment.
  - **jupyter 5.5.0** experiment environment.
- **Computer structure** All this work has been run in the same computer. As most of the experiments requires a time specification, it is important to show the computer architecture and operative system:
  - **Windows 10 v.1803** as Operative System.
  - **Intel i7-4820K CPU 3.70GHz** as Processor with 32.0 GB of Installed RAM.

4.2.2 Prior decisions

Here we show the main beforehand decisions used to start the implementation and the reasons why we have chosen them.

- **R1. Neural Network parameters** All the neural network architectures used have been implemented with *keras* library. For more information, see annex C.

- **R1.1. Single network:** The first decision that we took was to use a single neural network instead of one network for each output required. This was because one of our most important requirements was the time needed to execute the network to get all the solutions, so the separation of the network in 7 or 25 new networks would not be efficient. Also, we have assumed that, as the outputs are related with each other, to have a single network will not be a problem for the accuracy.
- **R1.2. Forward fully connected neural networks:** We have decided that we will use forward fully connected neural networks for two main reasons. The first one is because of the massive use of this kind of networks in the examples seen. The second reason is that we will work with optimization algorithms to find an optimal solution for the structure of the network. The use of non fully connected networks will produce an exponential growth of different structures that we should use, and it will make the problem much slower.
- **R1.3. ReLU as activation function** We will use *Rectified Linear Unit*, for so on, *ReLU*, as activation function for two reasons. The first one is because one of our main goals is to get a fast neural network (fast in execution time). So, using other activation functions with exponential operations would be much slower than using a singular *if* function. The second reason is because we will work with *deep* networks, and the use of some functions lead to some problems as the *vanishing gradient*, that is solved with the use of this function [18].
- **R1.4. Neural Networks hyperparameters** Apart from the network structure and the activation function, the rest of the hyperparameters (as the initialization function or the learning rate), has been fixed to avoid the exponential growth of the problem. We have used *normal* as the kernel initialization, *adam* as optimizer, *mse* as loss function and *mae* as metric. Also, the networks will be trained using a batch size of 25 times smaller than the data size. These values has been chosen following many examples seen and following our own experience in the earliest experiments tried.

- **R2. FBA data** We will create all the data for this project with the framework *COBRApy* [25] as it is mentioned in the section 4.2.1. For more information, see annex D.

  - **R2.2. FBA metabolism model used** We will use as metabolic models the three main metabolisms implemented in *COBRApy*. Those are *E. coli* model, *E. coli core* model or the model based on carbon for *E. coli*, and *Salmonella* model. We will call along this work *Ecoli* and *Ecoli core* to the metabolic FBA models related with *E. coli* and *carbon based E. coli* metabolisms. Those FBA metabolisms are explained in the annex A.

The *Ecoli* model is the most complex one. This is why the first hypothesis will be demonstrated with it. The *Ecoli core* model is a very simple and reduced model, so we will try the different optimization algorithms with it to minimize the time required. *Salmonella* model will be used as validation model, using it as contrast for the other two.

  - **R2.3. Maximum concentration used** There is not an exact value to fix the maximum concentration that would be in a cell environment, so we will use the value of **25** as maximum concentration for the metabolites in the executions (measured in nM). Above this value, our models will limit the metabolites because of their restrictions, so we will have redundant data.
  - **R2.4. Data normalization** Data normalization or data standardization is pretty advisable, but we have chosen to do our experiments without any kind of normalization for some reasons. The first reason is that we will want to have a fast neural network to run into a simulator. That means that will be faster if we jump normalization calculations in the input and output. Another reason is that out variables are very similar between them (we talk about metabolites concentration), and they will balance between 0 and 25 in the input and 0 and 80 in the output. So, we can not take a final value to normalize because the output value can always be higher than the one established, but it is an approximates bounded problem, so normalization is not completely needed.

- **R3. Optimization algorithm implementation** All the algorithms that have been explained in the *optimization* section 4.4, have been implemented manually. This decision was taken instead of using already implemented algorithms as in *scikit-learn* library, because three reasons. The first one was because our problem was not so easily compatible with some methods, as the parameters that we had were variable in number. The second reason is because we wanted to have total control over the execution of the algorithm, to change every parameter or function as we wanted. And finally, we did it because it did not take too much time to implement it by ourselves.

4.3 FBA hypothesis validation

This section is focused on the programs that have been implemented in order to demonstrate the firsts hypothesis of this project (**H1** and **H2** explained in 3.2). This is related with the use of neural networks to solve our problem.

The main idea of this section is to probe that the use of FBA in an *individual step by step* simulator is not efficient because of the time required to calculate it.

We will probe that the use of small, and therefore, fast neural networks, will achieve an execution time small enough to be executed in the simulator, and with an accurate enough result.

Also we will show the most relevant experiments that have been developed in order to explain step by step the design ideas taken.

We have used the *Ecoli* model in all this section to demonstrate our hypothesis. We have chosen this model because it is the most complex one implemented in *COBRApy*, and the most common one. So we can have a better idea of how much improvements we will achieve.

The experiments have been executed for the three metabolisms, the rest of results that are not shown in this section are collected in the annex B.

4.3.1 Time

In this section it will be discussed the time required to execute FBA and the time required to execute fully connected neural networks. This way we can compare the results and see how much time we will save using one or the other.

4.3.1.1 FBA Execution Time

As we have mentioned before, we will use *Ecoli* model. The specific parameters and information about this model could be seen in the annex A.

This model in *COBRApy* counts with 1805 metabolites and 2583 reactions. Within them there are 25 external metabolites that we will work with.

Executions for random input

First, we have tried to execute the FBA with a random input given. This input is related with each of the 25 input metabolites. This experiment has been repeated 25 times to get a distribution of how much time it would cost. These 25 times have been executed with the same input, so we can see if the time expired is deterministic.

In the figure 4.3 it could be seen that the time required to execute the same FBA (same input) is quite different. That could be because of the Operative System processes, but we can assume that so huge differences are due to the fact that it is an optimization problem, so the exact result is not always achieved in the same amount of time.

Figure 4.3: Execution of the FBA for a random input with 25 repetitions

Executions for null input

Another interesting execution that we could try is by using *null* or *zero value* for each of the metabolites. That means, the execution when there are no metabolites in the medium.

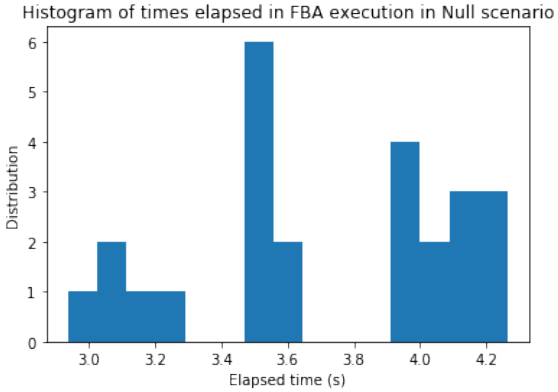


Figure 4.4: Execution of the FBA for a null input with 25 repetitions

In the figure 4.4 we can appreciate that the time needed for the system to achieve a solution with null input is much bigger than if we use random input. That could be related with the fact that it is more difficult to find an optimal solution when there are no biomass to optimize (all the possible solutions get a 0 value for biomass).

Varying null metabolites

In order to know how important is the number of null metabolites in the execution, we have tried some random inputs where we change some of the metabolites to have 0 value.

In the figure 4.5 we see graphically the time required to execute the FBA when none of the metabolites are null, and when we change some of this metabolites to

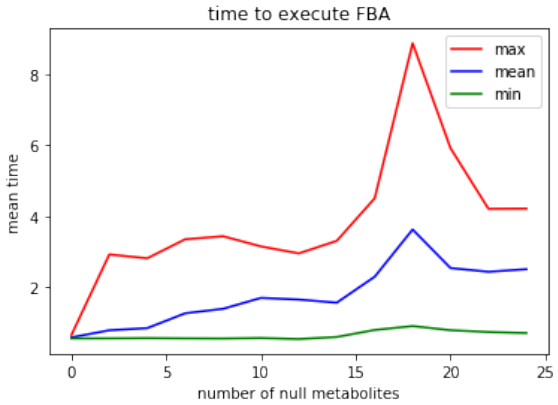


Figure 4.5: Execution of the FBA with null input metabolites

null input. Notice that the x-axis represents the number of metabolites that are not null. Each case has been proved with 25 different inputs, repeating 25 times each execution to erase the noise.

We can see that the executions with all null metabolites are quite slow, but they are not the worst case, as in 18 null metabolites we get the worst solution. Also we can see how the solution with no null metabolites is the only one with an small execution time.

We do not have an exact explanation about this phenomena, but we supposed that it is related with the optimization problem of FBA when it tries to optimize something that has not a "good" solution. So, if many of the solutions are 0 for biomass, it has to try more solutions to arrive to the final conclusion.

Multiple executions for random inputs

This time we will execute the FBA 50 different times for different random inputs, and measure the time values obtained. Each input will be executed 25 times and we will take the mean value of each to have the approximate value.

In the figure 4.6 we see, as we have mentioned before, that there are some specific inputs that take a much longer time to execute.

Finally we can conclude with this experiments the next results:
Mean time: 1.367 seconds.
Maximum time: 5.742 seconds.
Standard deviation time: 0.633 seconds.

We can conclude that the FBA execution has a very variant execution time.

4.3.1.2 Neural Network Execution Time

In this section it will be executed some experiments in order to know how much time cost to run a neural network. In all of the experiments shown below, we will

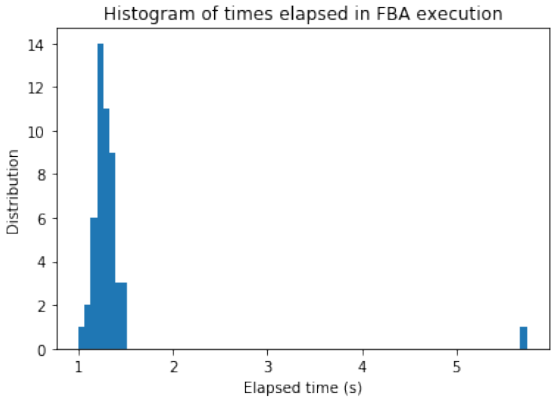


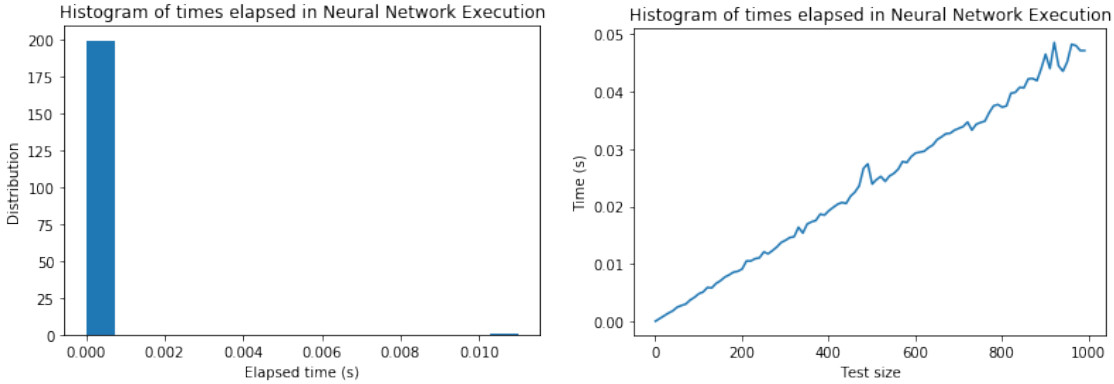
Figure 4.6: Multiple execution of 50 times of the FBA for a random input with 25 repetitions each input

use standard neural networks **without training** to just measure how much time costs to run it. We will not care about accuracy in this section.

Simple Neural Network

First, we will start with a simple neural network to see if the time of different executions are related between them. We will use a neural network with one input and one output, and two layers of 8 cells each. We will use this size specific network because it follows the first networks that we worked with that gave us good solutions.

For now on, we will named this networks as an array where we write the different cells for each layer, in this case the network will be $[1, 8, 8, 1]$.



(a) One execution (b) Multiple executions

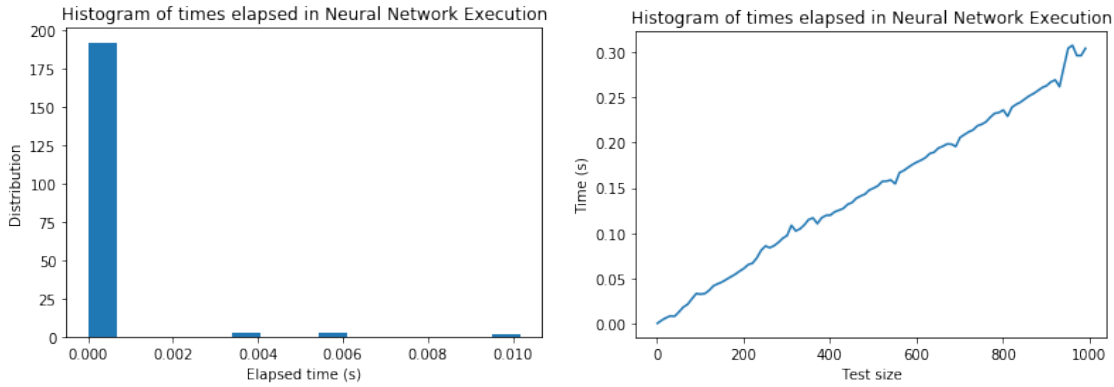
Figure 4.7: Time execution for a simple Neural Network

In the figure 4.7a we can see that almost every execution for a neural network has a time execution close to $5\dot{1}0^5$ s. There are some exceptions, that we assume that are because of the Operation System.

In the figure 4.7b we can see how, increasing the number of times we run this process, the time increases linearly with the size of data. This way we can calculate easier the exact time for a singular execution. This graph has been taken calculating the mean value between 200 executions for every step.

Complex Neural Network

In this case we will use the network: $[7, 10, 10, 10, 10, 8]$ to see the different results for larger networks. This is another network example that we tried when we started to work with deep neural networks, and it gave us also a good accuracy.



(a) One execution (b) Multiple executions

Figure 4.8: Time execution for a complex Neural Network

With the results shown in 4.8a we can see that the times for the execution of a neural network is **equal in every execution**. The 0 value is because of the *time Python* library round when we work with very short times.

In the figure 4.8b we can see the same result as before. The execution time is linearly proportional to the number of executions.

This is an important point to focus, because the differences between the network sizes are very important in the time that they spend on an execution. In this example we can see how it multiplies the amount of time needed by 6 from figure 4.7b to 4.8b.

Time depending on the connections number

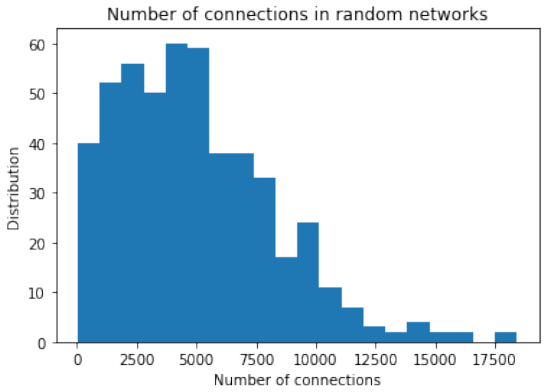
In this section we will study the different networks that we can use, and how their structures modify their execution times.

This connections are calculated assuming fully connected network, and without taking into account the *bias* values, just cells connections. This is because we have assumed that the bias value would take despicable time compared to connections.

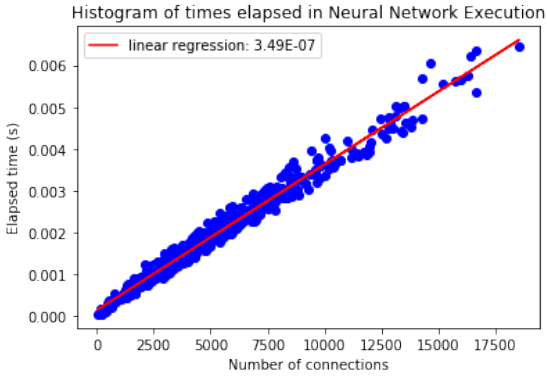
First of all, we should know how many connections we will work with. We have created 500 random networks with 10 maximum layers and 30 maximum cells per

layer. Those values have been varying along the project. The values have been chosen after many experiments with time and with accuracy of the networks, and finally these have been probed to be the bigger ones that we will work with.

In figure 4.9a we see this distribution, and we can see how our networks will be surrounding 17500 connections (if we use this limits).



(a) Number of random networks with specific number of connections



(b) Execution time of a neural network depending on their connections number. This graph has been calculated using random neural networks and measuring their mean execution time within 100 repetitions

Figure 4.9: Time execution depending on the number of connections

In figure 4.9b we see that the time elapsed to execute one of this neural networks is linearly proportional to its connection number, with an additional (probably Operative System) deviation.

Finally, in figure 4.9b we can see a summary of the different neural networks that we can have, assuming that every layer has the same amount of cells.

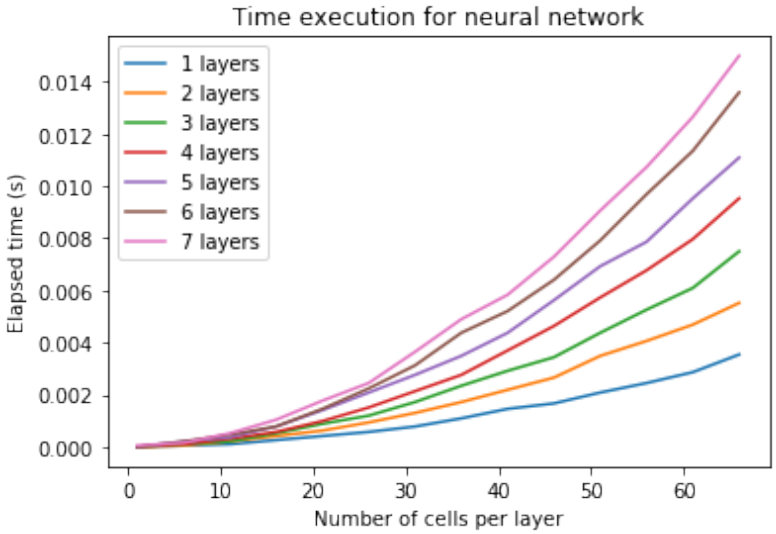
This final figure contains the whole conclusion of this last section about neural networks, that could be seen as:

The time required to execute a neural network is directly proportional to its number of connections.

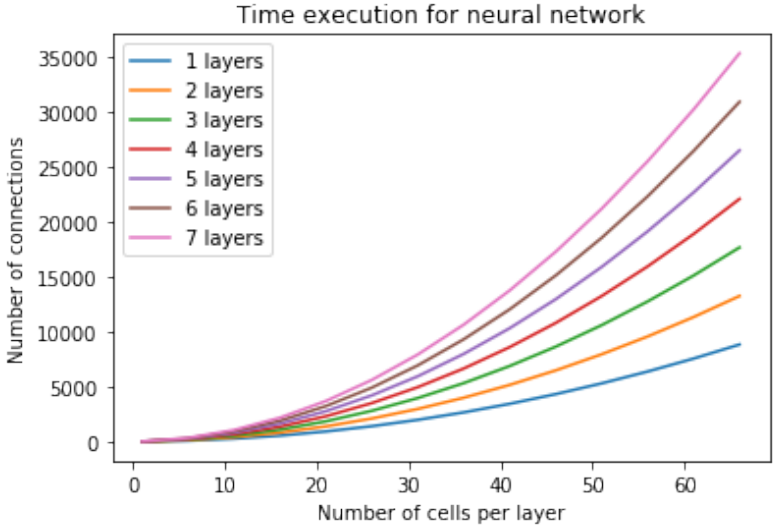
4.3.1.3 Conclusions

The time of a FBA execution varies in each execution and depending on the input, while the time spent by a neural network is always proportional to its number of connections.

We can conclude that not too big neural network (assuming as big neural network one with more than 35000 connections) will be at least 100 times faster than an FBA execution. And it will be always the same time regardless of the input.



(a) Execution time of a neural network depending on their number of layers and cells per layer



(b) Number of connections of a neural network depending on their number of layers and cells per layer

Figure 4.10: Time execution depending on the number of connections

4.3.2 Accuracy

Once we have discussed about the time elapsed in executing FBA and Neural Networks, now we will study the main accuracy that we can achieve (or hope to achieve) by using neural networks to approximate FBA solutions.

Data

For this section we will work with different neural networks, but we will use the same dataset in all of them, to have a comparable estimation between the different network structures.

As we are working with the *Ecoli* model, we have 25 metabolites, so our training dataset will have 25 different variables, and our output dataset will have 26 variables (25 exit metabolites plus one biomass output).

We have created this random data for training and test. In the figure 4.11 we can see in the horizontal plane the input data for two random metabolites, and in the vertical axis we can see the output data that FBA gives.

Just as a quick comment, we can see in 4.11a how the *glucose* is almost finished in every execution, as it is the main metabolite necessary for the metabolic reactions, as the metabolite *potassium* is almost unused 4.11c. And we can see in 4.11e that the biomass increases almost proportionally with the *glucose*.

Accuracy in random networks

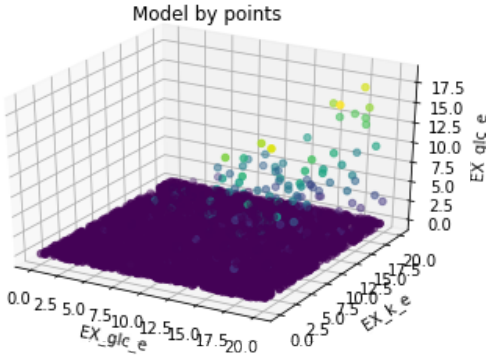
Now we will measure the mean error that we achieve using different structures of networks. For this, we are going to calculate this error as the mean error achieved for every metabolite in every test data.

This measure is not generalizable because the maximum value depends on each metabolite and on each dataset, but we will work with absolute error instead of with percentage, because it gives us a more reliable information, even when it is not so comparable (also the datasets are variable and some of them could be much "easier" than others).

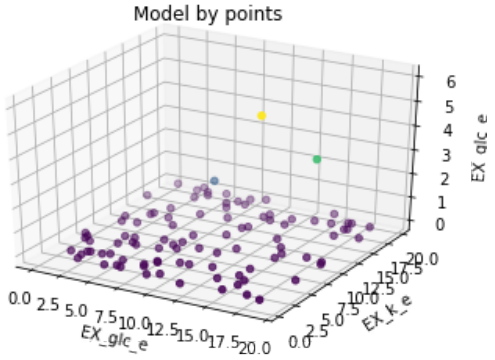
For this training we are using a top of 1000 epochs and 5000 data for training.

As we can see in the figures in 4.12 the error that we are moving around is about 3 and 5 for **random** networks, so we can assume from here that the error for random networks would be **less than 10%** in almost every case (in this case assuming the maximum values in our output).

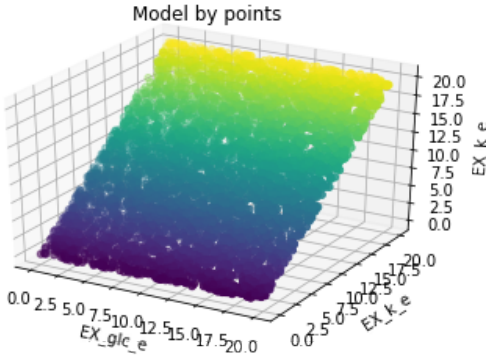
Also, we see that the distribution of connections is similar to the one shown in the last section, so we can not assure for now if there is a connection between the size of the network ant it's accuracy.



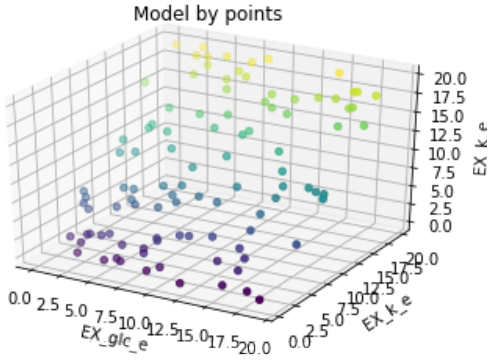
(a) Train points for *glc* metabolite



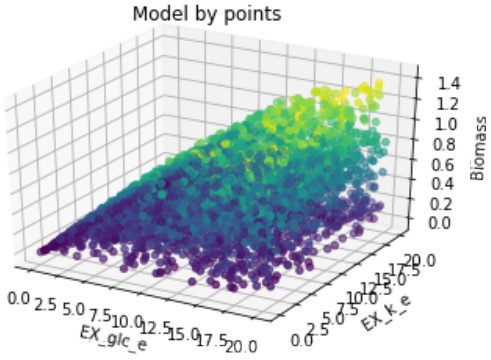
(b) Test points for *glc* metabolite



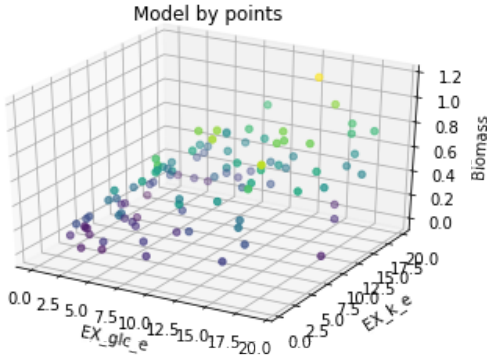
(c) Train points for k metabolite



(d) Test points for k metabolite



(e) Train points for *Biomass* metabolite



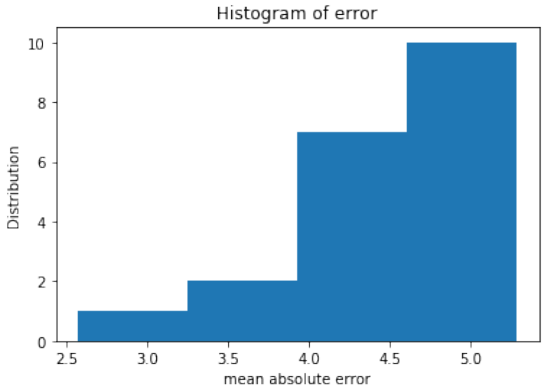
(f) Test points for *Biomass* metabolite

Figure 4.11: Part of the dataset to train and test the neural networks

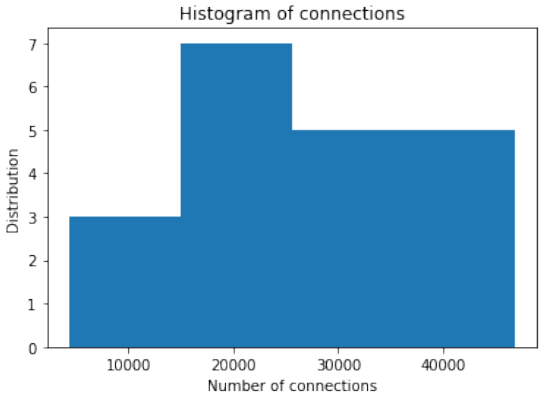
Accuracy modifying parameters

We have calculated different networks modifying their parameters in different ways to try to find the real important parameters that we should take into account.

We are going to discuss different aspects about the results in each of the parameters variability. Notice that these executions take a lot of time to be executed, so



(a) Error for random networks



(b) Connections distribution for random networks

Figure 4.12: Error and time estimated for random networks

they have not been executed repeatedly, so it is possible that they are affected by some noise.

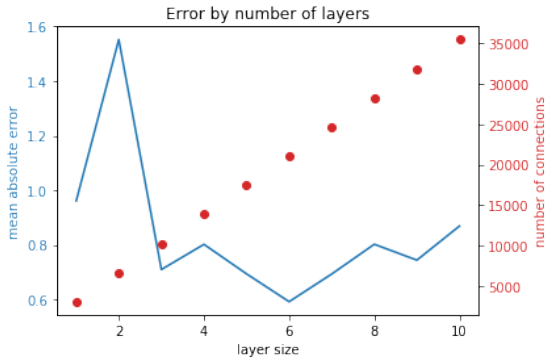
First, in 4.13a we can see how the network is not accurate for a single or double layer, but it decreases it's error with three layers. But with more layers for the same amount of cells per layer it has worse results. This may be because with more layers are worse networks, or just because it needs more epochs to train so we get worse solutions.

About the number of cells per layer, we see in 4.13b that the error decreases as the cells per layer increases. This is interesting but it is also a reasonable information that we could assumed from the beginning.

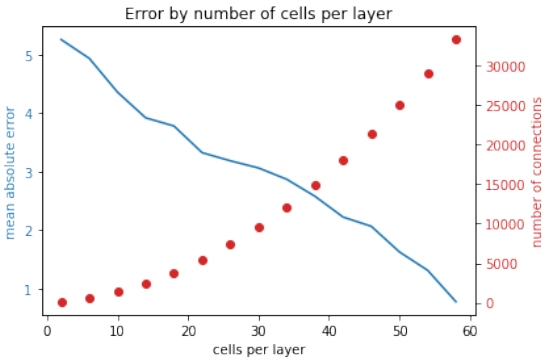
These two experiments are summarized in the figure 4.13c, where we can see how the error changes depending on both of the parameters, but mostly in the number of cells per layer. Here we can see how increasing the number of layers over 2 will not strictly improve the result (and it will increase highly the number of connections) so our optimal point will be between 2 and 4 layers and with as much cells as possible. (Notice that this execution has cost more than 3 days to run completely.)

But, we have an important result in 4.13d, that demonstrates that the number of connections in random networks is not related at all with the error achieved in it. This could be because of the few number of networks used for the experiment (it takes a long time to evaluate any of them) or because the structure of the networks is much more important than it's size.

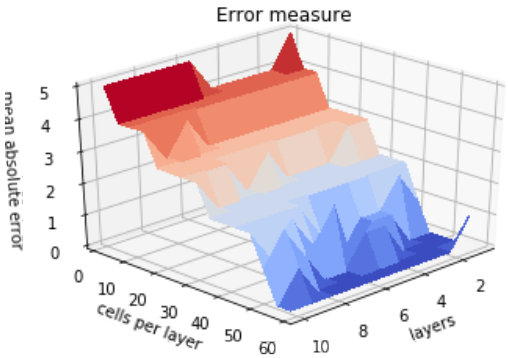
About graphs 4.13e and 4.13f we see a logical behaviour where the most epochs or data we use to train the net, the better results we get. Those results are for the maximum network, that would be 10 layers of 60 cells each. The peaks that we see in the figures are probably related with random training noise.



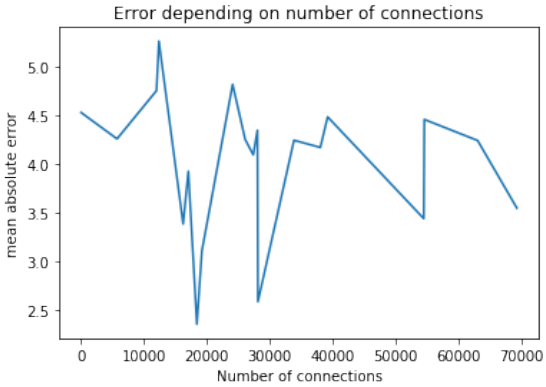
(a) Error depending on the number of layers for 60 cells per layer



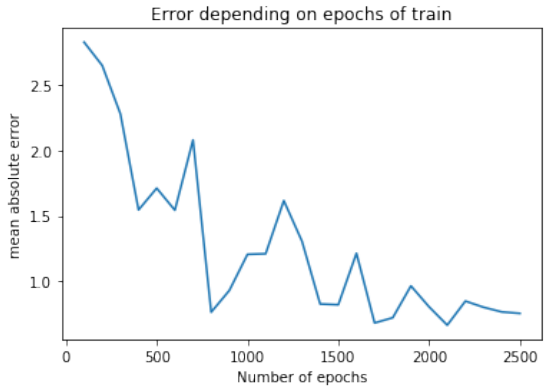
(b) Error depending on the number of cells per layer for 10 layers



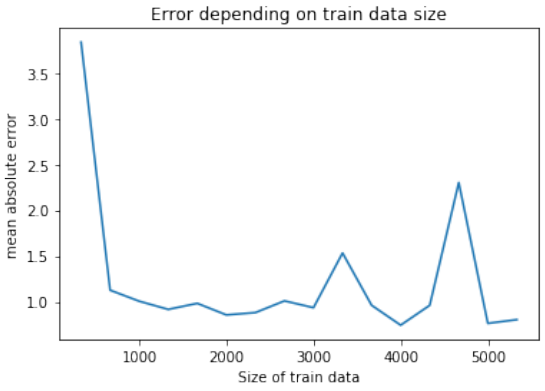
(c) Error depending on the number of layers and the cell per layer



(d) Error depending on the number of connections



(e) Error depending on the number of epochs of training



(f) Error depending on the size of the dataset

Figure 4.13: Results modifying parameters

4.3.3 Conclusions

Summarizing all the results that we have calculated along this section we can assure that:

1. Even when the time required to execute a neural network depends highly on its structure, it will always be lower than the execution of a FBA.

2. The error that we will work with is a relative low error (below 10%), noticing always that FBA is itself an approximation over the real world.

4.4 Optimization hypothesis validation

In this section we will talk about the experiments that we have implemented in order to demonstrate our third hypothesis (**H3** explained in 3.2).

We will see different experiments related with the training of the neural networks and which information we can take from them to apply in our implementations.

After this we will enumerate the different algorithms with their respective variants that we have implemented to optimize the network structure following the given FBA.

The main objective that we will try to target is to find a number of layers and the number of cells in each layer that gives us a fast and accurate neural network.

Every other parameter in the neural network has been skipped with default values to avoid extra complexity.

4.4.1 Training requirements

In this section we will show some experiments that we have executed in order to achieve some information that we could use to train our neural networks faster and more efficiently.

These experiments are related with the size of data that we will use for the training of the neural networks, and the number of epochs that are needed to do so.

Training curve

In this paragraph we talk about the common error curve that we can expect in our neural networks training.

In the figure 4.14 we can see a common error curve for a neural network with structure $[25, 50, 50, 50, 26]$ where the error in training and in test drop with the number of epochs. We can see that, even when the train error decreases more than the test one, there is not overfitting (there are not enough data to generate overfitting) and that the error decays quickly in the early epochs.

As we can see in the different figures in 4.14 the error curve for random networks are similar to the one shown (and to other examples not attached).

We can see how the curves have little jumps where stop to decrease for a while and then they start to improve again after some stagnant epochs. We could be for example in 4.15b that this jump occurs slightly in the early steps, while in 4.15c there are many jumps all along the epochs.

But even with the different examples shown, we can conclude from here that the curve will be decreasing with more epochs, and that the improvement ratio is much faster in the early steps than in the final ones.

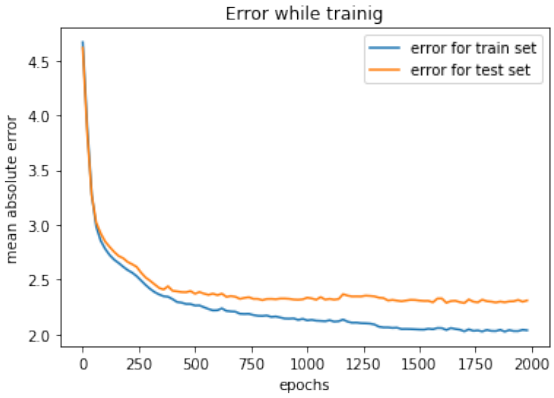
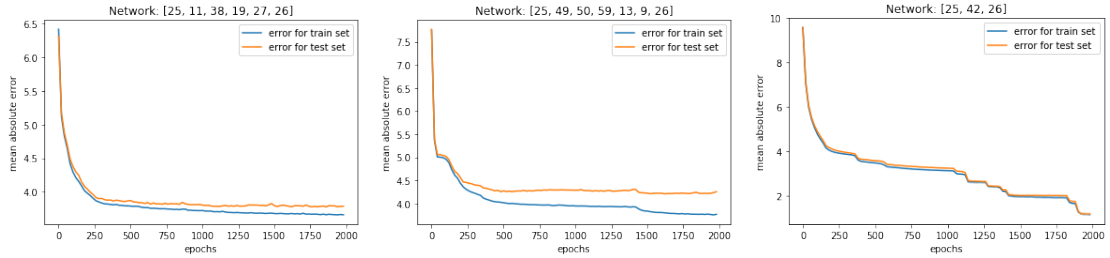


Figure 4.14: Error curve for training in standard neural network



(a) Error curve for random network (b) Error curve for random network (c) Error curve for random network

Figure 4.15: Different error curves for random networks

Number of epochs for full training

In this section we have executed many random neural networks to try to get a reliable idea of how the error is decreasing along the epochs.

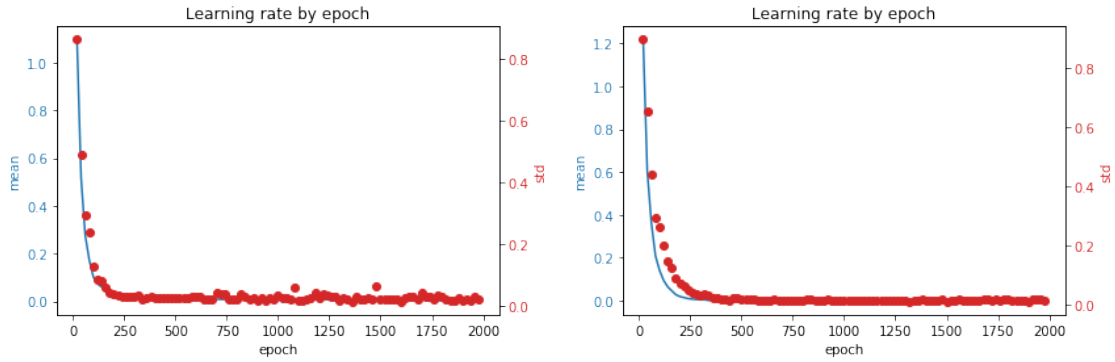
In the figures 4.16 there is summarized an experiment where we have executed several random neural networks, and we have measured the difference between the error in epoch n and the error in epoch $n+1$. We have measured the mean decreasing ratio and the standard deviation to measure if there are too much networks that moves from the standard behaviour.

This way, we can see that most of the networks that we will work with will achieve their *approximate best* solution in the early epochs (in spite of some examples as the one seen in 4.15c.

This experiment has been executed with 5000 data in 4.16a and with the half of it in 4.16b. We can see how the behaviour is practically the same, despite of slightly bigger deviation in the early epochs.

We have also probed that the different number of connections or the amount of

data neither affect in this values. To see this result review the annexed B the figure B.8.



(a) Error decreased over epochs for several networks with 5000 data

(b) Error decreased over epochs for several networks with 2500 data

Figure 4.16: Execution of 100 random neural networks to measure the error decreasing ratio.

Conclusions

After this experiments we can conclude that, in order to reduce the amount of time spent in the training of a neural network, we can **train the networks to 250 epochs** to know in few time how accurate this network will be once it is fully trained.

4.4.2 Metaheuristics

In this section we are going to summarize all the different attempts that we have made to find a proper algorithm that allows us to find a "good" network structure.

For this section we have used *Ecoli Core* metabolism, that is the carbon metabolism for the *E. coli* cell, instead of the whole metabolism. This choice has been made because some of this algorithms have very long execution times, so with a faster model we can get more results in few time.

Also, this way we can delimit the domain of networks structures that we are working with, because networks for *Ecoli* has been proved that need bigger networks, so more time to execute search algorithms in it.

Now we are going to present every algorithm that has been used to optimize this problem, and the results and comments about them.

Every algorithm explained here has been implemented personally, even when most of them have been designed following an already existed algorithms or methods [26].

In this work we will use in most cases their original names even when most of them are personal variants from the original algorithms.

We will train the neural networks only with 250 epochs. This design decision has been made because of the result of the last section, where we demonstrate that with this epochs we can already decide how accurate a network would be.

Another design decision that we have made is to run every algorithm with different datasets (all of them with same size of data). These datasets are produced randomly following a uniform distribution over all the metabolites. The code used could be seen in annex D.

This decision could be seem as illogical because of the time elapsed in obtaining these datasets, and because the comparison between different algorithms could be not so relivable because of the difference in the measures (we will measure the accuracy by the *mean absolute error*).

But this decision has been made because we are not going to focus in which algorithm get the best solution for a determined dataset, but the error curve and the different solutions that all this algorithms present. So, we do not want to get a real solution, we are evaluating the algorithms themselves.

4.4.2.1 Ant Colony

When we think about our problem, that is to find the correct number of layers and the correct number of cells per layer, we intuitively could see this problem as a *path finder*.

That is the reason why we has implemented a *Ant Colony based* optimization algorithm [27] [28]. Because we can exploit the power of this algorithm by using an already executed neural network to anticipate where the good solution should be.

These is a tricky problem because, in our case, the number of layers in the networks is variant, and depends on this value, the number of parameters that our network has is different.

For example, in a neural network with only 2 layers, the number of parameters of this network will be 3, the number of layers and the number of cells in each layer. But in a network with 4 layers, we will have 5 parameters.

This complicates the problem, because it means that we do not have a bounded domain to find a path, but this domain could increase with every execution.

So, we could not use the standard algorithm but a personal one that allows the individuals to have a variant amount of parameters.

Algorithm explanation

We have implemented two different variants for this algorithm. Both will follow the simple logic shown in the algorithm 1.

Algorithm 1 Ant Colony standard algorithm

Input: actual fitness data
Output: best network
  while stop flag **do**
    build new network using already fitness data
    calculate fitness of the new network
    update fitness data
  end while

The difficulty in this algorithm was not calculating the fitness because we do it just by calculating the *mean absolute error* (that will be comparable inside the algorithm as every network tried would use same dataset) and pondering it with the time (calculated by number of connections) that takes the network to execute.

The difficulty is neither in the network built part. This is done following the schema 2. Here we can see that we use different random distributions (as geometric, normal, etc.) to get a random number that will be a parameter in the new neural

network. So, from here, we just need to know which value is the one that we will use as **mean value** for any of the parameters, and we know already how this algorithm works.

Algorithm 2 Algorithm to create the new individual

Input: $data$ = actual fitness data
Input: nd = numerical random distribution
Output: new network
  $layerMean \leftarrow$ from $data$ get mean layer value
  $layerSize \leftarrow$ from nd get random number with mean = $layerMean$
  $newNetwork$ = new network with $layerSize$ layers
  for $layerSize$ **do**
    $cellMean \leftarrow$ from $data$ get mean cell value *// this is where the variants differs when they get the value from data*
    $cellSize \leftarrow$ from nd get random number with mean = $cellMean$
    $newNetwork$ add new layer with size $cellSize$
  end for
  return $newNetwork$

The difficulty then, resides in the fitness data, and how to storage and update it. This is the part of the algorithm where the *path* for the ants is written.

And here is also where both variants diverge one from each other.

Ant Colony generic

In this variant of the algorithm, we assume that the neural networks with different amount of layers have not improvements in common. So, in this variant, we will storage fitness data for each number of layers, and in each of them, we will store fitness for each number of cells in each layer.

To understand this variant we will see it as if we have a table where we store each of the fitness obtained depending on the number of layers that our network has. This way, we can calculate easily which *number of layers* would be more appropriate for the next individual to improve.

We have now the same table that we had in the step below. This time this table just remains to one of the *number of layers*, so we have N tables where N is the maximum number of layers we have obtained.

In this new tables, we are going to storage the values of the fitness for each $numberofcellsineachlayer$.

Schema

We will show this schema with an easy example:

First, we assume we start with a network that has just 1 layer, and in this layer it has 3 cells and with total fitness 5, we will have our tables shown in the figure 4.17a.

After this step we will create a network with obviously mean layer size 1, but let's assume we create the second network (randomly) with two layers, that are 3 and 2, and total fitness 1. We will have an updated table as the one seen in 4.17b.

Knowing this values, we can see that we have a much better fitness in networks with 2 layers than with 1, so out layer mean will be calculated in order to have proportionally to the fitness more probability to be 2. In this mean value for the new network will be 11/6, so it would be more likely to have 2 than 1.
Once we have chose randomly if the network will be of one or two layers (or more) we will choose the value for each of the layers according again with the table. Let's assume that the new network is a [_,3,3,_] network with fitness 2. We would update the table as shown in 4.17c.

From here we can see that the mean layer size will be also similar to the one before, more concretely it will be 23/13 (a bit lower than the one before as the fitness in two has decreased). And so on we will find one by one the next network and we will update the table.

Table of *number of layers*		Table for 1 layer		
Number of layers	Mean fitness	Layer	value	Mean fitness
1	5	1	3	5

(a) Example1, adding [3] with fitness 5

Table of *number of layers*		Table for 1 layer			Table for 2 layer		
Number of layers	Mean fitness	Layer	value	Mean fitness	Layer	value	Mean fitness
1	5	1	3	5	1	2	1
2	1				2	3	1

(b) Example1, adding [2,3] with fitness 1

Table of *number of layers*		Table for 1 layer			Table for 2 layer		
Number of layers	Mean fitness	Layer	value	Mean fitness	Layer	value	Mean fitness
1	5	1	3	5	1	2	1
2	1'5					3	2
					2	3	1'5

(c) Example1, adding [3,3] with fitness 3

Figure 4.17: Results for Generic Ant Colony algorithm

This method could be seen a bit tricky and anti-intuitive, but it was the only

reasonable way that we found to avoid the variant number of parameters for the individuals.

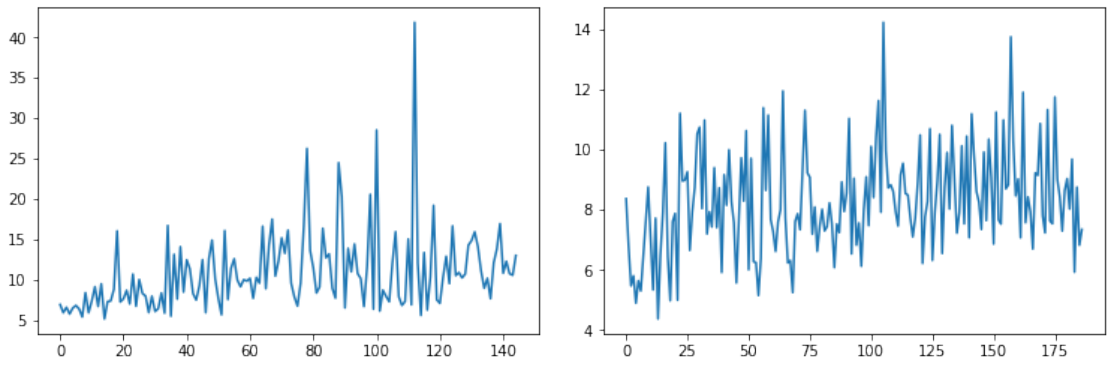
Results

So following this algorithm, we have executed it with two different random distributions, one with a **geometric** distribution and the other with a **binomial** one.

In the figure 4.18 we can see how this different methods does not converge to any valid solution. Instead, the values diverge extremely because once we have chosen randomly a huge neural network, the mean value for *number of layers* will increase, even when the fitness is really high against the others.

Also, as we do a proportional chosen between all the fitness, and with big networks the fitness will increase quickly (there is a pondering in the fitness calculation about the number of connections). The lower values, even when they are pretty different, will be seen similar at the time to calculate the new mean.

This is better seen in the geometric distribution. That even when is more probable to get lower values than high values, it is much probable to get much more bigger numbers than in a binomial. That is why the error increases extremely, as the mean *number of layers* does.



(a) Generic Ant Colony with geometric distribution

(b) Generic Ant Colony with binomial distribution

Figure 4.18: Results for Generic Ant Colony algorithm

Ant Colony by layer

In this variant, the methodology is the same as we have explained before, but we are assuming this time that, if we have x cells in layer n and it is a good value for a network with N, it will be also a good value for the layer n for a network with $M > n$ layers.

With this, what we achieve is to reduce the fitness table, so we reduce the complexity of the algorithm. Also we can reuse some early structures that could be useful for different layer numbers.

Schema

The same example shown in the variant before it is represented in the figure 4.19.

Table of *number of layers*		Table for layers		
Number of layers	Mean fitness	Layer	value	Mean fitness
1	5	1	3	5

(a) Example1, adding [3] with fitness 5

Table of *number of layers*		Table for layers		
Number of layers	Mean fitness	Layer	value	Mean fitness
1	5	1	2	1
2	1		3	5
		2	3	1

(b) Example1, adding [2,3] with fitness 1

Table of *number of layers*		Table for layers		
Number of layers	Mean fitness	Layer	value	Mean fitness
1	5	1	2	1
2	1'5		3	3
		2	3	1'5

(c) Example1, adding [3,3] with fitness 3

Figure 4.19: Results for Layer Ant Colony algorithm

Results

As we had much less data stored for this method, we have used a **normal** discretized distribution, because we could store the mean and standard deviation for every case. This way we can narrow the random distribution to be focused in more efficient values.

The execution of this algorithm could be seen in the figure 4.20.

We see here the same problem that we had in the other variant, even when our standard deviation compress the searching domain.

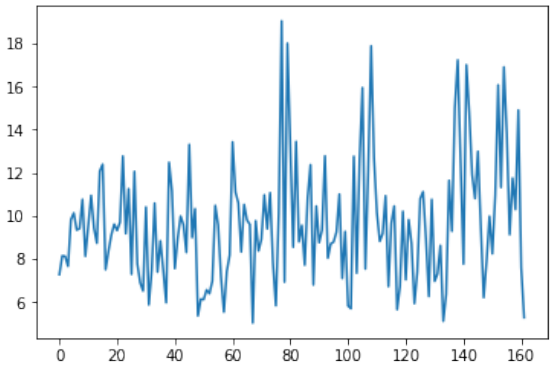


Figure 4.20: Result for Layer Ant Colony algorithm with normal distribution

Conclusion to Ant Colony

As we could see in the execution and the new individuals that these algorithms create, we can assure that **this is not an efficient method** to solve this problem.

4.4.2.2 Simulated Annealing

Following the intuition, we can try to assume that, if a neural network with an specific structure has an error x, another network "similar" to this one would have a similar error $x \pm \epsilon$. We understand similar networks as those ones that have many layers in common, or similar size of cells in each layer.

Guided for this intuition, we thought about the possibility of using a local search algorithm for trying to solve our problem.

We thought in simulated annealing because it has the powerful to change locality in the earlier steps and it can explode an specific neural structure neighbourhood [29] [30].

Algorithm explanation

The algorithm used for this implementation is the standard *Simulated Annealing* algorithm. We present it in the algorithm 3 in order to declare the exact version of it.

We have implemented two different variants of this algorithm, both following the schema presented in algorithm 3. Their difference remains in the way that the neighbourhood is design.

Generic Simulated Annealing

Schema

In this version of the algorithm we use the following changes as the steps to get a *neighbour* network, and each of them with the presented probability:

1. Add a cell to one layer $\rightarrow 1/3 * 0.95 = 0.32$
2. Remove a cell from one layer (if there are more than one cell) $\rightarrow 2/3 * 0.95 = 0.63$
3. Duplicate a layer $\rightarrow 1/3 * 0.05 = 0.02$
4. Remove a layer $\rightarrow 2/3 * 0.05 = 0.03$

We have set the probability of increasing the network to 1/3, and the probability of changing the number of layers to 0.05.

This way we see that there are more chance to create smaller networks (because one of our targets is to reduce the network). Also the chance of modifying the number of layers in the network is much smaller, because it supposed a much bigger step.

Algorithm 3 Simulated Annealing algorithm

Input: T_0 initial temperature, α temperature reduction
Input: nn_0 initial neural network
Output: best network

```
T ← T_0
nn ← nn_0
actualFit ← nn fitness
  // Stop flag will be true when the temperatures arrive to a limit point or when the maximum time is reached
while stop flag do
  newNN ← nn random neighbour          // Different variants diverge here
  fit ← newNN fitness
  if fit < actualFit then
    nn ← newNN
    actualFit ← newNN fitness
  else
    worsenProbability ← (fit − actualFit)/T
    if randomValue < worsenProbability then
      nn ← newNN
      actualFit ← newNN fitness
    end if
  end if
  T ← T ∗ α
end while
return best nn studied
```

Results

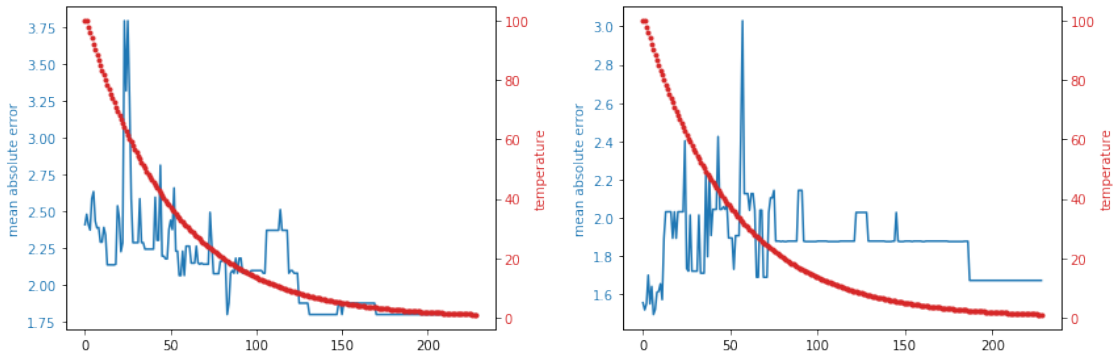
To try this algorithm we have run two different executions. Both starts from the same initial network $[15, 15, 15]$, and the different between them is the *time* weighing. This initial networks is as a good initial one, found from other executions.

In the figure 4.21 we see two different executions. In figure 4.21a we can see the execution where we have into account the size of the network, so the number of connections have repercussions in the fitness. While in the figure 4.21b we see the execution where the time value is 0, so in this execution the algorithm will just take into account the accuracy of the network, and not its size.

In both figures in 4.21 we can see how in the early epochs the algorithm changes more quickly and it allows higher fitness values, but in general we can see how, as the temperature goes lower the jumps start to disappear to achieve a *good* solution.

It is important to notice that both algorithms starts in the same individual, but as the fitness is calculated with or without time, they don't have the same results, so they are not directly comparable using fitness as measure.

We see also in the steps given for the algorithm, that both decreases the number



(a) Generic Simulated annealing with 0.002 time weighing

(b) Generic Simulated annealing with 0 time weighing

Figure 4.21: Generic Simulated annealing error evolution

of layers and the number of cells in each layer as they go lower.

In 4.21a we clearly see how the fitness decays with the epochs. This is because is much easier to find better fitness just by decreasing the network size than finding a better accuracy, so the fitness decreases as the size of the network does.

Decreasing Simulated Annealing

In this version of the algorithm, we will just assume that our network can only decrease its size.

This could be not logical intuitive to do in this kind of algorithm, because we lose the possibility of going back to an already seen good network. But knowing that the results in the last version does not work because of the big networks visited, we tried this new version, easily implemented from the other.

Schema

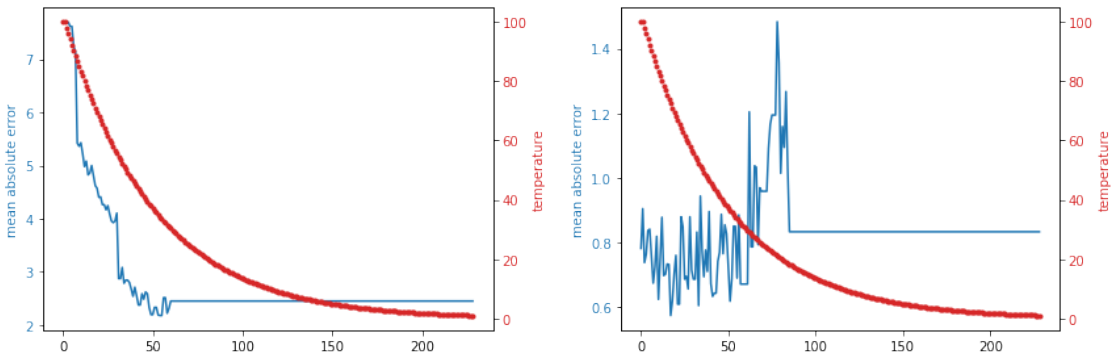
In this version we use the following neighbourhood:

1. Remove a cell from one layer (if there are more than one cell) $\rightarrow$ 0.99
2. Remove a layer $\rightarrow$ 0.01

This way we see that every new step we will have a smaller network. Also the chance of modifying the number of layers in the network is much smaller, because it supposed a much bigger step.

Results

As in the other version, we have tried to execute this algorithm taking into account the number of connections, and without them.



(a) Decreasing Simulated annealing with 0.002 time weighing

(b) Decreasing Simulated annealing with 0 time weighing

Figure 4.22: Decreasing Simulated annealing error evolution

The results that we see in figure 4.22 have similar behaviour to the ones in the variant before. Both change faster at first and starts to stabilize. In this case, both falls in a local minimum, fact that is easy to anticipate because they both decreases the size with the time, so they will arrive to a small structure that does not allow better accuracy.

Conclusions to Simulated Annealing

We can conclude seeing the results given that *Simulated annealing* is not a practical method to find the structure of a neural network. We assume that this is because we need so much time to train a network, and this algorithm needs a great amount of epochs and a slower temperature shift to achieve good solutions.

Even so, we could get some good solutions from this executions, and also we could see how the networks changed with the epochs finding the best structures.

4.4.2.3 Evolutive

Following some advises from along the Literature [31], we have tried to approach this problem with evolutive algorithms [32] [33]. More specifically we will use genetic algorithms.

This kind of algorithms has the disadvantage that are slower, because they need to work with a *population* of individuals. So we will need to execute more networks to achieve a good solution than in the other algorithms.

Even with this disadvantage, the literature shares the conclusion that it is a good choice to find better solutions in this kind of problems.

We have implemented two very different algorithms that will explode different aspects of the evolutive systems.

Population based algorithm

This is a standard simple evolutive algorithm. It is based in having a random initial population that will be crossed and mutated.

Algorithm explanation

The evolutive implementation that we have made follows the algorithm 4, following an already existing algorithm published [31].

Algorithm 4 Population Based Evolutive algorithm

Input: Population size
Output: best network
  population ← Random initial population
  while stop flag **do**
    $twoIndividuals$ ← take two random from *population*
    $twoNew$ ← cross $twoIndividuals$
    $twoNew$ ← mutate $twoNew$
    population add $twoNew$
    population erase two worse individuals
  end while
  return best *nn* in *population*

The difficulty in this system is to declare which is the best way to cross individuals.

This method follows the next algorithm 5 based on the methods seen in the State of the Art base on *grammars* [34].

In this method we cut randomly the two networks and we concatenate one of the first half of the first individual with the second half of the second, and vice versa.

The mutation of the individuals is made over the new individuals and it follows the same rules as in the *Simulated Annealing* case, but with different probability values:

1. Add a cell to one layer → 0.36
2. Remove a cell to one layer (if there are more than one cell) → 0.54
3. Duplicate a layer → 0.04
4. Remove a layer → 0.06

Algorithm 5 Individuals Crossing

Input: I_1 first individual
Input: I_2 first individual
Output: two new individuals
$index_1 \leftarrow$ random index lower than $len(I_1)$
$index_2 \leftarrow$ random index lower than $len(I_2)$
$newI_1 \leftarrow I_1$ to $index_1$ **appended** I_2 from $index_2$
$newI_2 \leftarrow I_2$ to $index_2$ **appended** I_1 from $index_1$
return $newI_1, newI_2$

Results

In the figure 4.23 we can see the error decreasing for this algorithm.

We can see clearly how the mean value of the fitness over the population is decreasing over the epochs, as we do not allow worse individuals to survive.

Also the best individual is improving in the early steps, achieving pretty good solutions against the last algorithms.

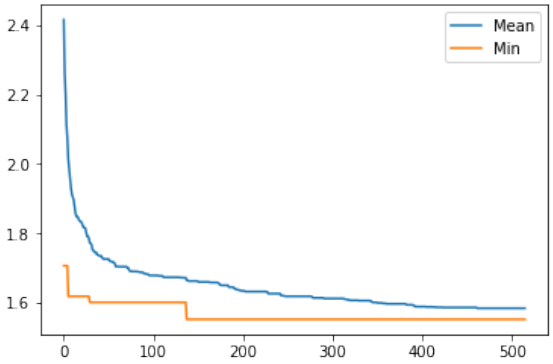


Figure 4.23: Execution of Evolutive algorithm during a half day with a population of 25 individuals

Multiobjective algorithm

For this algorithm, we have exploited the power of using multiobjective heuristics. This means that we will improve both objectives separately, and we will get best solutions for both objectives hoping that there would be also solutions in the middle way between them [35].

Algorithm explanation

For this algorithm we have based on a Literature algorithm called *Pareto Archived Evolution Strategy* or *PAES* [36].

In our variation of this algorithm we have a variable size population that archived every solution that is not dominated. This means that there are not any other solution that has both objectives better at the same time.

This process is represented in algorithm 6, following the crossing and mutation process as the algorithm before.

This multiobjective division will lead to have very good solutions in both objectives but maybe non good one in both. But as we are working with evolutive algorithms with crossing, we can assume that medium solutions will appear.

Algorithm 6 Multiobjective Evolutive algorithm

Input: initial population size
Output: population of non dominated individuals

```
population ← Random initial population
while stop flag do
  twoIndividuals ← take two random from population
  twoNew ← cross twoIndividuals
  twoNew ← mutate twoNew
  for new in twoNew do
    if new is not dominated then
      population add twoNew
      for all i in population dominated by new do
        population remove i
      end for
    end if
  end for
end while
return population
```

Results

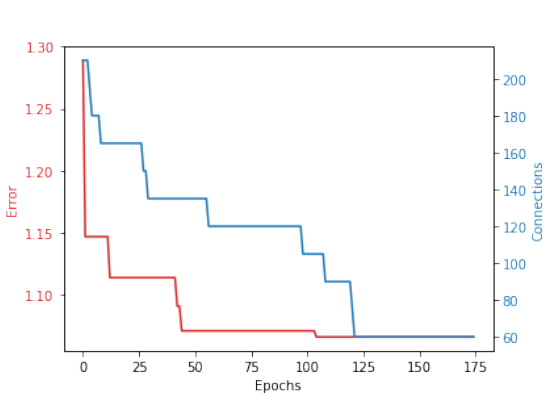
In this section we present the results for this multiobjective algorithm. The reader must take care at the time of getting conclusions over the solutions, because this graphs are different from all the others in the way that they represent differently both objectives.

In the figure 4.24a we can see how both objectives have been improving with the epochs. There is not shown the mean value for each objective because we have to remember than in the population are both individuals with very good fitness in one objective and very bad in the other.

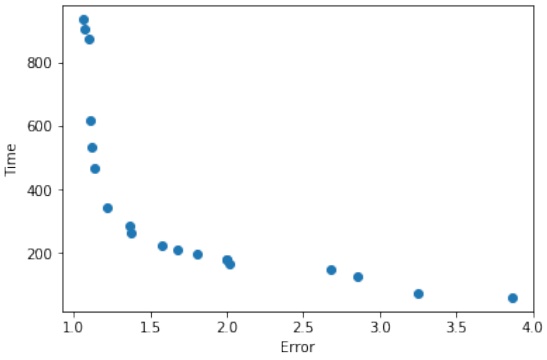
The error has decreased quickly but it got stuck early, while the size of the network was easy to improve during long times (it just needs to reduce the network size).

In the figure 4.24b we see the final population that we got from the execution of this algorithm till it got stuck in 50 epochs without improvement.

We can see clearly that the best solutions for both objectives are at the same time very bad solutions in the other objective. Something typical in multiobjective problems. But we can solve this problem by using as an optimal solution one of the middle solutions.



(a) Error evolution over the epochs



(b) Solution distribution over the two objectives

Figure 4.24: Multiobjective Evolutive algorithm execution

Conclusions to Evolutive algorithms

This kind of algorithms have led us to the best solutions and networks achieved till the moment. This means that, even when the other algorithms tried has been not determined to use as possible results, there are some searching methods that really lead us to accurate solutions.

With this heuristic searches we can assure that we will find an *accurate enough* solution in all objectives thanks to the crossing and slightly mutations over already good networks.

But this method is not yet good enough to use as a final implementation because it requires more individual studies (neural networks training) to create the initial population, what makes it much slower than others.

Also, as this methods just allow the best solutions to be part of the population, there is not much probability of avoiding a local minimum if the population falls in one.

4.4.2.4 Search methods

After trying with heuristic methods to solve our problem, we arrived to the conclusion that non of the heuristic method used would be fast and efficient to solve the problem.

This motivates us to create *non heuristic* methods, or what we have called in this work: *search* methods. These methods follow an *exhaustive search* idea, in the way that we force which would be the next individual in each step.

This way, we reduce highly the time needed to find a good solution. And, at the same time, we search for an answer in certain local points where we know that there are good results.

We do not want to use this methods as the final implementation for our program, but to learn more about our problem and their local values.

We have implemented two different variances, that follows similar principles, but have opposite points of view.

Forward Search

In this algorithm we have focused the search in a single assumption: *if in a neural network with n layers, the best amount of cells for layer i is x, then for a network with* $n + 1$ *layers, the best amount of cells for layer i would be x.*

As we can notice, this is a non logical assumption, and we had the possibility of failing with it. But as we have said before, this attempt is an approximation for further algorithms, and not a solution itself.

Algorithm explanation

This method is represented in the algorithm 7. In this algorithm we increase the size of the last layer in every step. When we have not improved the result in x steps, we return to the best solution with this amount of cells and we add a new layer.

Results

The evolution of the error for this algorithm can be seen in the figure 4.25.

In the figure 4.25a we can see how the fitness grows with the epochs as the network will grow in size with time.

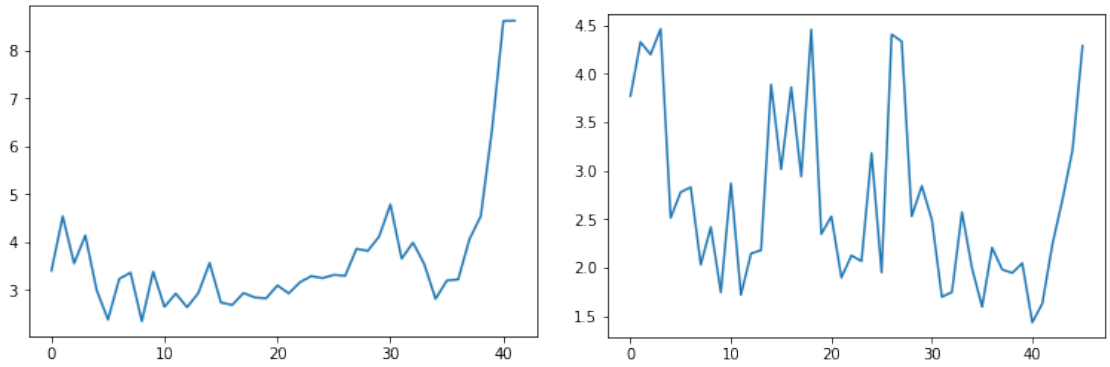
In the figure 4.25b there is another behaviour because it just takes into account the accuracy taken. We can see clearly how there is not a logical evolution of the error.

Both approaches falls in the same error, because they create a too big first layer, and from there they lost the possibility of improving the accuracy, so they just visit

Algorithm 7 Forward Search

Input: I_Max iterations without improving
Input: L_Max maximum layer size
Output: best network

```
actualNetwork ← [input, 1, output]
while actualNetwork layer size < L_Max do
    bestFitness ← ∞
    i ← 0
    while i < I_Max do
        fit ← actualNetwork fitness
        if fit < bestFitness then
            bestFitness ← fit
            i ← 0
        end if
        add cell to last layer in actualNetwork
    end while
    actualNetwork ← best network of this size
    add layer to actualNetwork
end while
return best network visited
```



(a) Forward Search execution with 0.005 time weighing

(b) Forward Search execution with 0 time weighing

Figure 4.25: Forward Search error evolution

random networks.

So this method do not give us good solutions or interesting information.

Inverse Search

In this approach we use the opposite point of view as in the algorithm before. We start from a big network, and reduce their cells or layers in order to achieve a smaller and better network.

Algorithm explanation

This method is represented in the algorithm 8.

In this variant we erase cells or layers from the networks in a **sorted** way. First we check layer by layer if erasing a cell improves the fitness, and if this fails, we try by erasing layer by layer.

In this algorithm we only change of network when we get a better solution, so it could be seen as a *gradient search*.

Algorithm 8 Inverse Search

Input: $First\_Network$ initial neural network
Output: best network
  $actualNetwork \leftarrow First\_Network$
  $actualFitness \leftarrow actualNetwork$ fitness
  1.
  for all i **in** layer in $actualNetwork$ **do**
    $newNetwork \leftarrow actualFitness$ with 1 cell less in i layer
    $fit \leftarrow newNetwork$
    if $fit < actualFitness$ **then**
      $actualNetwork \leftarrow newNetwork$
      $actualFitness \leftarrow fit$
      Go to 1.
    end if
  end for
  for all i **in** layer in $actualNetwork$ **do**
    $newNetwork \leftarrow actualFitness$ removing layer i
    $fit \leftarrow newNetwork$
    if $fit < actualFitness$ **then**
      $actualNetwork \leftarrow newNetwork$
      $actualFitness \leftarrow fit$
      Go to 1.
    end if
  end for
  return $actualNetwork$

Results

The results for different executions of this algorithm are shown in the figure 4.26. This executions has been run with an initial network of $[\_, 30, 30, 30, 30, 30, \_]$

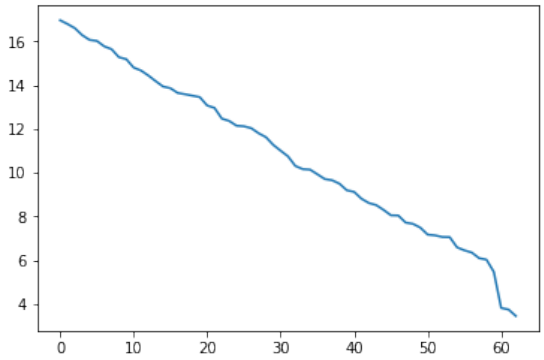
In the figure 4.26a we can see how the fitness decays while the networks goes smaller. The high jump in the final steps is related to an erased of two different layers in two different consecutive steps, so the size of the network is reduced highly, and so is the fitness.

In the other two figures 4.26b and 4.26c we have very poor results because erasing just one cell or layer does not achieve a better solution.

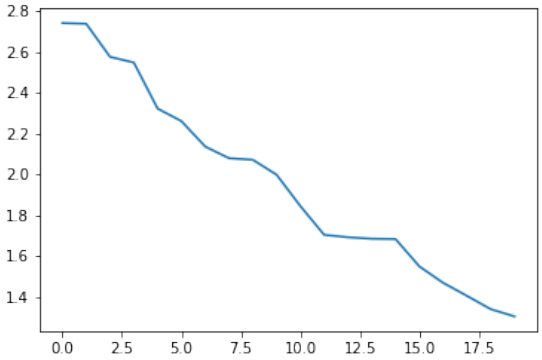
In the case of 4.26b we keep reducing the fitness as we erase layers. This reduce drastically the total fitness even when the weighing is so low, so we get more solutions.

But in 4.26c we see how the execution fails trying to find a network with better solution in just 4 steps. Also we have to see that the first solution taken is pretty accurate from the beginning.

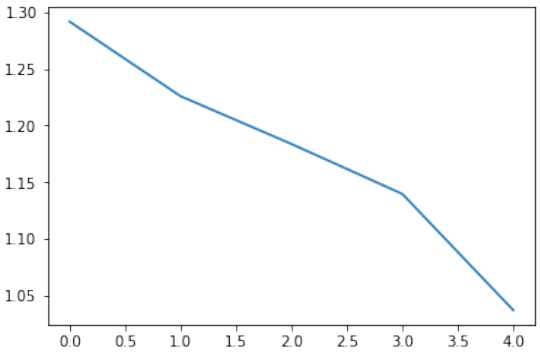
This could be because of random facts that makes the training not improve, but the result is that we do not have an accurate solution.



(a) Inverse Search execution with 0.005 time weighing



(b) Inverse Search execution with 0.0001 time weighing



(c) Inverse Search execution with 0 time weighing

Figure 4.26: Inverse Search error evolution

4.4.2.5 Conclusions

So, to conclude with this heuristic method, we have to say that **we have not found a fast reliable method** to get a neural network in few enough time as we expected.

But, all this process has been useful because we have learned important information about which structures are more likely to have better solutions and which ones are so slow or so big that we should not even search between them.

First of all, what we have found is that, **deeper structures does not necessary means better solutions**, as we already knew from the section 4.3.2. They could have in some cases better accuracy but deeper solutions have more connections, so they are slower. We have also seen that **one layer structures does not approximate accurately** our problem, or at least not with a reasonable amount of cells. This reduce our searching space widely to 2 or 3 deep layer networks.

Another interesting (and unaccountable) fact that we have got from the results is the "specific" structures that get better solutions for our problem.

This is what motivates the next section.

4.4.3 Exhaustive methods

4.4.3.1 Direct methods

So we finally came to the conclusion that our network searching method should be completely focus in an already known structure, to reduce the number of *changes* or *parameters* that it has.

With this proposed we have implemented two new direct and focused methods that will achieve a proper solution in few time, with the accurate results that we will show.

This methods are not heuristics anymore, because they focus their searching in a *brute force* algorithm that exploits every possible solution.

This two structures follow the schema shown in figure 4.27. Those schema have been probed to be the most accurate and small ones given by the different algorithms presented before.

Both approaches has their advantages and also their different disadvantages, but in general they achieve the best solutions for our problems.

This approaches has been proved with the 3 different FBA metabolism that *cobraPy* published, that are:

- **Ecoli Core** or the carbon metabolism for *E. coli*. This is the metabolism used in the heuristic part of this project 4.4. There is also the simplest one, with just **7** input variables.

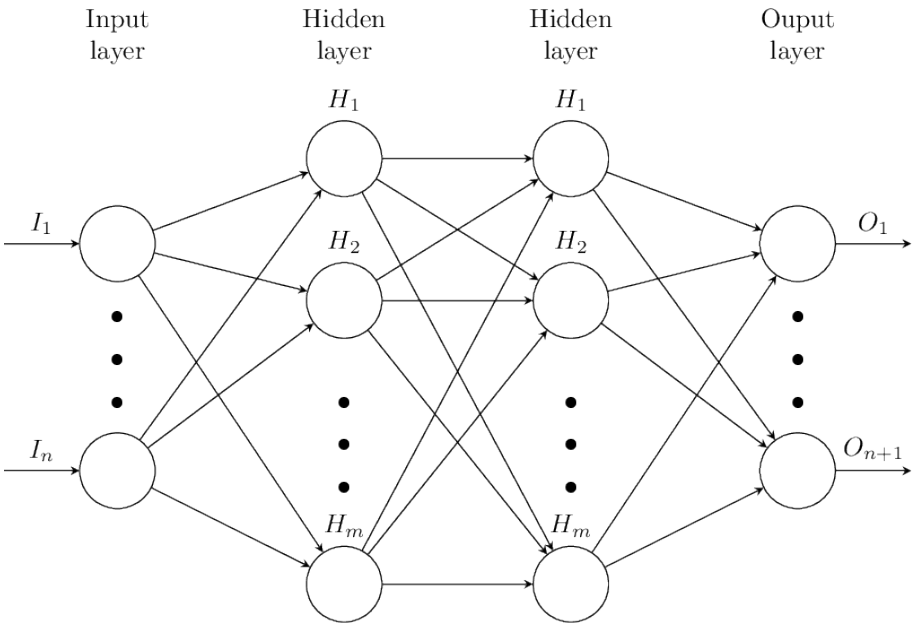
- **Ecoli** metabolism. This is the metabolism used in the FBA hypothesis section 4.3. It is the most complex one with **25** metabolites.

- **Salmonella** metabolism. It has **20** metabolites.

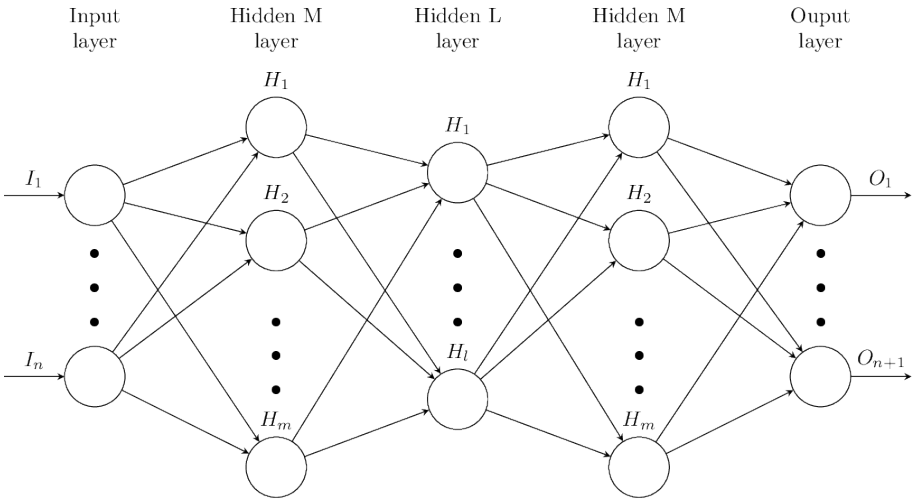
In each of them we have resized the *Biomass* output value with a weighted (in our case has been 100) to make the system focus more in the improvement of this specific metabolite. This was because this output will be the most relevant one within the others.

In this methods we also use a different way to measure the fitness, that is with the percentage of the error, instead with the *mean absolute error*. This is because now we want to find the exact solutions for the exact problem, and not to see how the algorithm evolves during the epochs.

We have attached also the error for the *Biomass* value in each execution to validate it.



(a) General structure with 2 layers



(b) General structure with 3 layers

Figure 4.27: General structure of neural networks for final solution

Two Layer Direct Method

In this variation of final solution, we follow the structure seen in 4.27a. This means that our neural network will have an input layer of i cells (i external metabolites) and an output layer with $i + 1$ cells (i metabolites plus *Biomass*). And it would have two hidden layers both with the same amount of cells n.

Algorithm explanation

So this algorithm 9 works in a really simple way. It just probes with an n number for the hidden layers, and it keeps increasing it till it reaches the best solution.

Algorithm 9 Two Layer Direct Algorithm

Input: n initial hidden layer size
Input: It_Max maximum iteration without improving
Output: best network

```
i ← 0
while i < It_Max do
    nn ← [input, n, n, input + 1]
    fit ← nn fitness
    if fit improves actual fitness then
        i ← 0
    else
        i ← i + 1
    end if
    n ← n + 1
end while
return best nn visited
```

Results

We can see in the figures 4.28 the different results for the execution of this algorithm.

Also in table 4.1 commented later, we can see the exact results for this executions.

In the figure 4.28a we see the execution for the *core* metabolism, that is the simplest one. We can see how the error decreases quickly in the early epochs, and once the layer is bigger than double of the input layer, the error stabilizes in approximate 4%, that is an accurate value assuming the few epochs and data used to train the network.

Also we can see how the number of connections increases quadratically as we increment the cells in the hidden layers.

We have shown also the error in *Biomass* metabolism, that it could be seen as the *most important* output for our network,. We see that it is at least as low as the rest of the outputs.

Even when *Salmonella* is a more complex model than the *core* model, in 4.28b we see that the error of this network, once the hidden layers arrive to bigger networks. The error gets stable in practically the same accuracy as in the model before.

We see in *Salmonella*, as in *Ecoli* model in figure 4.28c, how the error decreases once the hidden layers arrives to bigger number of cells, and the error gets stable even when we increase highly the number of connections.

Three Layer Direct Method

In this variation of final solution, we follow the structure seen in 4.27b.

We see that our network will have an input layer of i cells (i external metabolites) and an output layer with $i + 1$ cells (i metabolites plus *Biomass*). And it would have three hidden layers, the first and third with m cells and the second with $l < m$ cells.

Algorithm explanation

This algorithm 10 works similar to the one seen before. It establish a constant l, and try different m values for it. And once finished it updates $l \leftarrow l + 1$, and it stops when there is no improvement anymore.

Results

We can see in the figures 4.29 the results for the executions of this algorithm.

The behaviour of the different red lines allows us to know when a new value l has been taken, because the number of connection is reduced again. In many cases this change also gives a very bad solution.

There are also different sizes of red lines, because the step between l is taken depending on whether the network improves or not.

In the figures 4.29a and 4.29c we can see how the error starts very high, but it decreases quickly with the epochs, even if there are plenty of isolated networks that has a great error. In *Ecoli* it is clearer that the epochs converge to better solutions as l increases.

In the case of *Salmonella* shown in figure 4.29b, we can see clearly how the error is low in the early epochs, and as the layers increases the isolated worse errors are disappearing.

This is why the algorithm converges quickly and stops earlier than in the other cases.

Conclusions to Direct Methods

In the tables 4.1 are shown different results and data from the execution of the two variants of the algorithm.

Algorithm 10 Three Layer Direct Algorithm

Input: l initial hidden layer size
Input: It_Layer_Max maximum iteration without improving for layer
Input: It_Total_Max maximum iteration without improving in total
Output: best network

```
i ← 0
while i < It_Total_Max do
  i ← 0
  m ← l
  while j < It_Layer_Max do
    nn ← [input, m, l, m, input + 1]
    fit ← nn fitness
    if fit improves actual layer fitness then
      j ← 0
    else
      j ← j + 1
    end if
    m ← m + 1
  end while
  if fit l improves actual total fitness then
    i ← 0
  else
    i ← i + 1
  end if
end while
return best nn visited
```

The results on 4.2 are calculated by taking the best network seen in the algorithm, and training it with the same amount of data and with 2000 epochs.

In the first table we can see clearly how the *2 layer* approach spends much less time because the search is linear, and also we arrive faster to better solutions and with much less networks visited. But it also arrives to higher networks that we want to avoid.

In this table we can see some non expected values. In ^{*1} we probed more networks for the *core* than for the others, but this is just because, as it is the simplest model, we wanted to start the search from smaller networks. In ^{*2} the algorithm exceeded the maximum time given (half day) so it cuts the execution and returns the best solution found.

In the second table we see the accuracy that the best neural networks visited achieve. We can see that the *2 layer* approach gives faster and more reliable networks in general than the *3 layer* method. ^{*3} *mae* means *mean absolute error*.

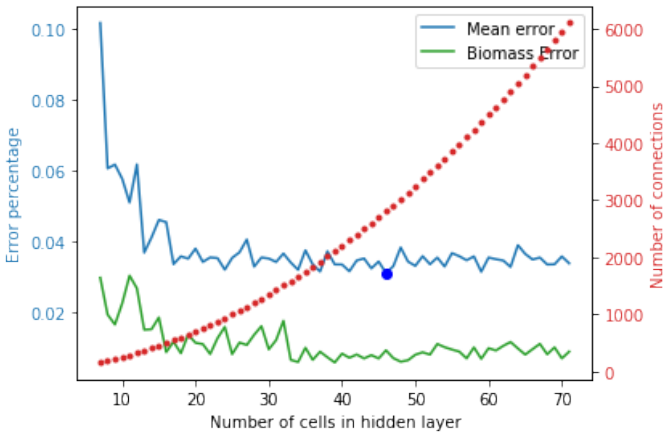
Method	**Dataset**	**Execution Time** (s)	**Networks visited**
2 Layers	*Core*	1890	65 *1
	Salmonella	1757	46
	Ecoli	1809	39
3 Layers	*Core*	43392 *2	344
	Salmonella	20300	146
	Ecoli	40915	337

Table 4.1: Final results for *Direct Methods* execution

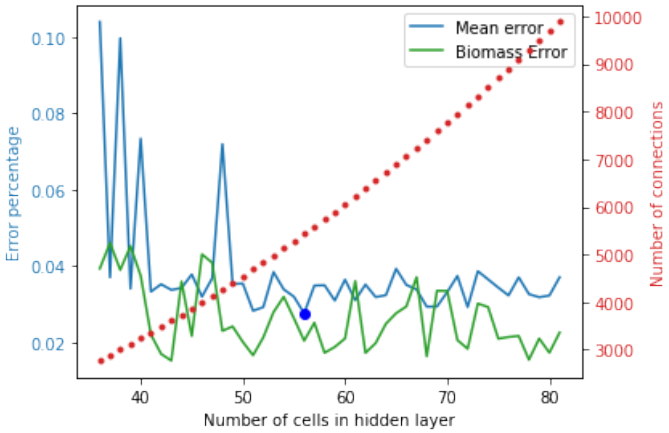
Method	**Dataset**	**Best Solution**	**accuracy**	**mae**[*3]	**connections**
2 Layers	*Core*	[7,46,46,8]	0.962	1.71	2806
	Salmonella	[20,56,56,21]	0.968	1.16	5432
	Ecoli	[25,62,62,26]	0.976	1.01	7006
3 Layers	*Core*	[7,38,29,38,8]	0.973	1.3	2774
	Salmonella	[20,40,38,40,21]	0.963	1.33	4680
	Ecoli	[25,63,57,63,26]	0.975	1.22	10395

Table 4.2: Best individuals for *Direct Methods* execution

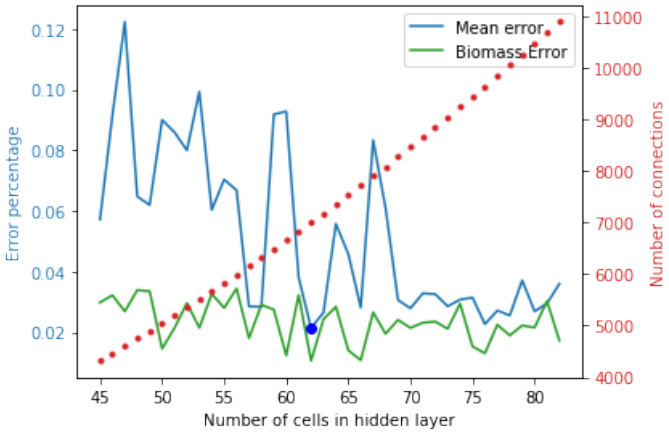
We can conclude from this experiments that, even when the *3 layer design* looks like more accurate and with better and faster networks, finally **the bounded amount of time that is spent to achieve a good solution with the *2 layer* method makes this algorithm the best choice** to focus the final implementation.



(a) Execution for *Ecoli core*

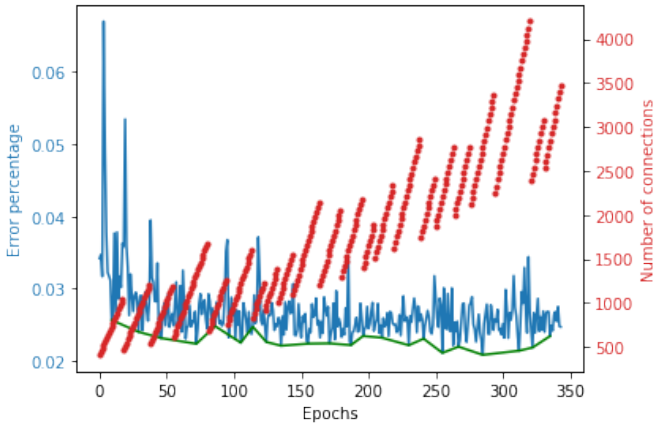


(b) Execution for *Salmonella*

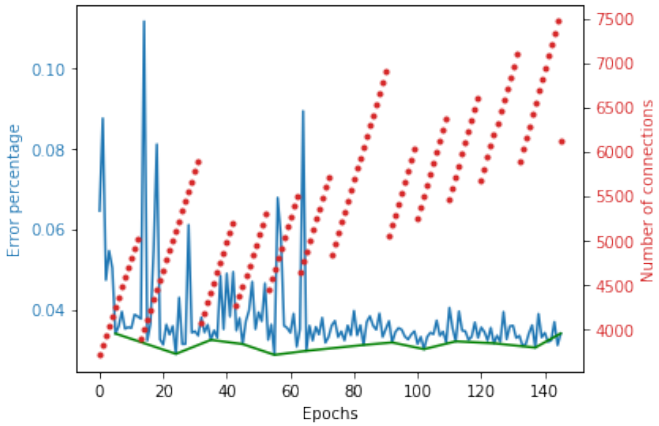


(c) Execution for *Ecoli*

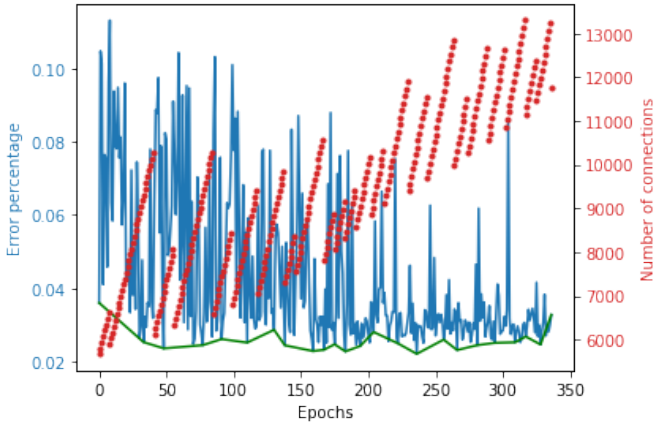
Figure 4.28: Execution for all the metabolism. Red dots indicates number of connection per network. Green line represents the error measured in *Biomass* metabolite



(a) Execution for *Ecoli core*



(b) Execution for *Salmonella*



(c) Execution for *Ecoli*

Figure 4.29: Execution for all the metabolism. The red dots indicates the number of connections. Green line represents the evolution of the best solutions visited for each l

4.4.4 Final Implementation

We have demonstrated that the most efficient method to find a correct neural network is by using a design as the one presented in figure 4.27a. This design needs enough hidden layers to achieve the desired accuracy, but not as much to have too much connections. This value is, by the few models that we could use in this project, variant with the model. This value is approximately between $2I$ and $4I$ being I the number of input variables.

By running random neural networks with *2 layers* design and with n number of cells in hidden layers, we have achieved the results shown in table 4.3. Here we can calculate approximately the accuracy that a network will have and how big it would be depending on the proportion between n and I, with the time spent in each network training.

We have implemented the **final solution** that follows the simple algorithm 11. It just takes values for n till it finds a network that is more accurate and smaller than the maximum given.

These *input* values are depending on the necessities of the network that we want. If we prefer a faster network, we would choose a smaller $maxConnections$ number, and so a higher $maxError$, and if we want more accuracy we will do the opposite.

For an approximate help in the inputs of this algorithm, we could use the table 4.2 where we can predict how much time it will cost to achieve a network with x maximum error and y maximum number of connections.

In table 4.2 we have summarized an experiment where we try 100 networks with random n for each result (each row in the table). This n will vary between *min proportion* and *max proportion*, this means that the n will be uniformly distributed between $minP \cdot I$ and $maxP \cdot I$ being I the number of input variables.

Easier to understand these results than looking into the table 4.2 (but also less accurate because it has less repetitions) would be to look up the graphs in the figure 4.30. Here we can see the evolution over different layer sizes of the percentage error for each model.

In 4.30a and 4.30c we can see clearly how the error decreases while the cells increase. But also we have to see how the minimum value raises slightly at the end. Also the number of connections increases quadratically, so these solutions may not be as good as desired.

In 4.30b we see how the error does not decreases with the epochs. In this case bigger networks may be worse than smaller layers.

In order to demonstrate that those results fulfilled our hypothesis, we will compare our results with the maximums set. For this we have the table 4.4 where we have set the worst scenarios that we have calculated with the results in 4.3.

Algorithm 11 Final implementation

Input: $maxError$ maximum error valid
Input: $maxConnections$ maximum number of connections
Input: α increasing ratio. *Default value as* 1
Input: I input number of cells
Output: trained network

```
                    // Maximum n that has less than maxConnections connections
nMax ← floor(positive_solution(N^2 + N(2I + 1) − max = 0))
                                                  // Proportion between I and nMax
proportionNI ← I/nMax
                                  // The algorithm starts not so far from the maximum
n ← (proportionNI − 1) ∗ I
                                // The algorithm will run a maximum time of I iterations
while n < maxConnections do
  nn ← [I, n, n, I + 1]
  fit ← nn fitness
  if fit < maxError then
    return full trained nn
  end if      // Values for n could be seen as random because the fitness of each
  network is also random                          // Lower α implies longer algorithm
  n ← ceil(n ∗ (maxError − fit) ∗ α)
end while
return Error: Not network found
```

Here we set the values that are related with all the datasets:

- **Maximum error allowed:** 10%.
- **Time per connection:** $4 \cdot 10^{-}7$.
- **Maximum time over the FBA:** 50 times shorter.
- **Maximum time to obtain the solution:** 600 seconds.

In table 4.4 the *maximum error, number of connections and solution time* have been calculated by adding the maximum mean error to its standard deviation.

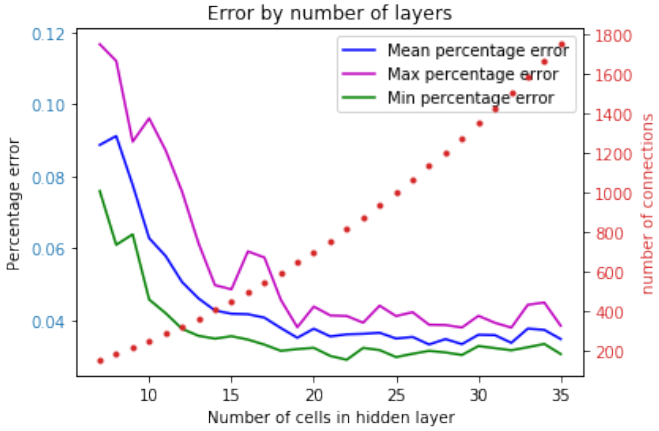
The time proportion has been calculated by taking the *time per connection* and multiply it by the maximum number of connections. Then we divided the *FBA mean time* by this value and we get how many times the network will be faster than the FBA.

Dataset	**Min proportion**	**Max Proportion**	**mean accuracy**	**mean connections**	**mean time** (s)
Core	2	3	**0.971** ± 0.012	**541.9** ± 107.0	**30.65** ± 5.25
	3	4	**0.971** ± 0.003	**931.0** ± 122.3	**52.04** ± 7.40
	4	5	**0.972** ± 0.002	**1434.8** ± 156.7	**86.84** ± 8.60
Salmonella	2	3	**0.965** ± 0.015	**4363.1** ± 823.8	**49.12** ± 15.01
	3	4	**0.970** ± 0.003	**7540.3** ± 1075.9	**97.38** ± 12.74
	4	5	**0.969** ± 0.002	**11696.4** ± 1242.3	**158.98** ± 19.03
Ecoli	2	3	**0.946** ± 0.023	**6936.7** ± 1324.5	**49.68** ± 13.40
	3	4	**0.966** ± 0.004	**12243.3** ± 1516.5	**97.88** ± 16.82
	4	5	**0.965** ± 0.003	**18276.6** ± 2063.2	**158.12** ± 17.46

Table 4.3: Results for final implementation along 100 random networks for every case

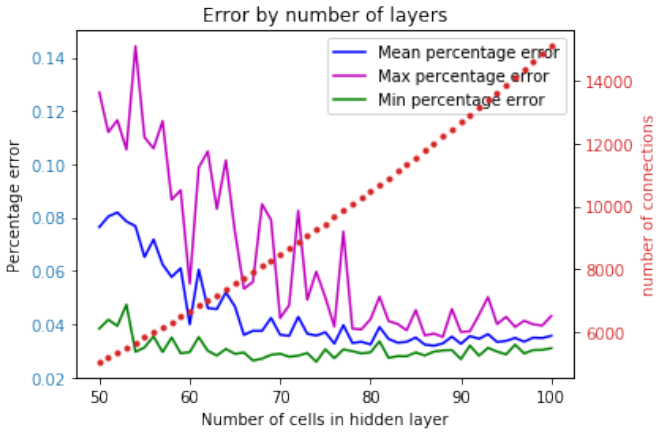
Dataset	**max error** $< 10\%$	**max connections**	**FBA mean time** (s)	**time proportion** $>$ 100	**max time to solution** $< 600s$
Core	4.1%	1054	0.022	52.18	$60s$
Salmonella	5.0%	8616	1.254	363.86	$110s$
Ecoli	7.7%	13760	1.367	248.36	$115s$

Table 4.4: Worst solution expected for final implementation



(a) Execution for *Ecoli core*

(b) Execution for *Salmonella*



(c) Execution for *Ecoli*

Figure 4.30: Repetitive executions for every model along different sizes. These values has been calculated repeating 10 times each hidden layer size with **random** datasets each time

4.4.5 Conclusions

After all the results shown in this work, and other results attached in annexes or intermediate results founded along this project, we can conclude that:

1. We have **not** found a fast and reliable heuristic method that allows to achieve a fast and accurate neural network for this specific kind of data.

2. The most efficient method to find a correct neural network is by using a design as the one presented in figure 4.27a with enough hidden layers to achieve the desired accuracy, but not as much to have too much connections.

So, we have calculated that, **using *2 layer design*** we can obtain in few time a reliable neural network that achieves accuracy or speed enough to satisfy the possible requirements that we have asked for. Using as hidden layer size n between $2I$ and $4I$ being I the number of input variables, we can conclude that we will find a solution **in a limited time**, with a **maximum number of connections** and with **boundary accuracy**.

Chapter 5

Conclusions

5.1 Conclusions

The work presented in this dissertation is framed within the area of synthetic biology, in which the use of simulators is crucial to reduce costs in time and in money.

This work is based on improving a specific part of the metabolism to allow the simulators to compute the metabolism of the cells in an fast way.

The theoretical metabolic model that we have chosen, that is *FBA*, is too slow to be implemented into an *AbM* simulator. So we had to find a faster model which allows us to approximate this FBA results.

The implementation that we have built is independent of the data that we choose to train it. This allows future projects to use other more complex models, or even to use real experimental data in order to create a reliable model. For example the model and data presented in [37].

Therefore, as we have demonstrated with the results in section 4.3, the use of neural networks for approximating *FBA* models is as fast as reliable, allowing us to achieve very fast networks with very accurate outputs.

Also, the use of neural networks creates a system that scales much better and more predictably than any other complex model. This allows to simulate great bacterial consortia, as the *microbiota*. This is something that any *AbM* is able to do nowadays.

Regarding the use of optimization algorithms, we have concluded that they are too slow and expensive to be used as our desired final implementation. Those methods require too much time of execution because of the amount of different tries that they need to converge to a useful solution.

But the implementation of this methods has been useful in order to learn about the neural network structures, and how they fit with our problem. From here we have obtained some patterns that has been used in the final implementation.

This leads us to the final solution, that was based not in an algorithm, but in a random selection that assures us (at least with the dataset that we have tried with and with small variants of them) to get a neural network already trained that fits the data chosen, and that it will be executed in short time and with a great accuracy.

This solution is understood as a *black box* where we do not see what is done inside, and we do not even care about. This is a beneficial point for this work, because it does not just solve our problem in GRO, but it is useful for every other simulator or every other system that needs to accelerate an FBA calculation, independently of the system itself.

To finish, we can present as the most important conclusion from this project the fact that it is possible to extrapolate this same model to other sources of data, and in every kind of simulator. This makes this solution widely scalable.

So finally we can conclude that we have arrived to a valid implementation that fixes the main problem that we wanted to solve:

- **H1:** The network will have a maximum error of **10%**:
  E. coli Core: Maximum error of 4.1%.
  E. coli: Maximum error of 5%.
  Salmonella: Maximum error of 7.7%.

- **H2:** The network execution must be shorter than **100 times** the FBA execution:
  E. coli Core: Minimum of 52.18 times faster than FBA.
  E. coli: Minimum of 363.86 times faster than FBA.
  Salmonella: Minimum of 248.36 times faster than FBA.

- **H3:** The time limit to find a network that fulfill **H1** and **H2** is **600 seconds**:
  E. coli Core: Maximum time of 60 seconds.
  E. coli: Maximum time of 110 seconds.
  Salmonella: Maximum time of 115 seconds.

Except for the case of the *E. coli Core* metabolism, where the time gained is not so significant, we fulfill the hypotheses established.

5.2 Future work

Even when the results for this project have been highly satisfactory, we know that this has been just a starting point for future research works and future improvements

in this area.

We now present some starting points to future work, in order to summarize the points that are still pending:

Implementation in GRO

The implementation in GRO is obviously an important need requirment for this work to be useful, and it is a branch of this project that has been already started and that will be achieved in few time from now.

The reason why we have not delved in the implementation of this project in GRO and why we have not discussed about it along the project is because this project was focused more in the theoretical part. This is the demonstration that this whole project was a useful approach. We have not focused this work on the incorporation in GRO, leaving this final step out from the project.

This has allowed us to focus more in the solution for the general problem as it is: ***FBA* acceleration**, than in solving a more concrete problem focused on GRO.

Final implementation scope

Following the explanation given before, there are some assumptions that we have taken because of the scope of the project. We wanted to probe that this would be a proper solution for our problem, but we have not focused on the improvement of this specific solution.

The use of optimization techniques when using neural networks, as intelligent initialization, the use of normalized data, the use of dropout techniques, etc. could lead to better final solutions that the ones given. So, this is an interesting future line .

Trying new methods

There are a whole world of heuristic methods that we could not try for this project, that could lead to better results for the optimization of the structure of the network.

This algorithms such as *Tabu Search* or other *Genetic Algorithms* among others, could be the answer for this problem, and could become a better solution than the one given.

Scaling the model with other data

This whole project has been based in the metabolic data extracted by the *FBA* model for the three species that *cobraPy* implemented (*E. coli*, *E. coli core* and *Salmonella*.

But this project is scalable to any other kind of data, as long as it follows the same structure.

This allows this solution to work with more complex models of *FBA*, with other theoretical models for metabolism or even to work with real experimental data.

Appendix A

FBA specification

In this annex we will show the different models that we use for the *FBA* execution following the *COBRApy* specification for them.

A.1 *E. coli* Core

This model is based in the *Ecoli* carbon metabolism. This is the perfect model to work with, because it is small enough to execute quickly and is simple enough to understand and follow its execution.

It is also a good model that we would like to implement in our simulator.

A.1.1 Model Summary

Name	e_coli_core
Memory address	0x021518c820f0
Number of metabolites	72
Number of reactions	95
Objective expression	1.0*Biomass_Ecoli_core - 1.0*Biomass_Ecoli_core_reverse_2cdba
Compartments	cytosol, extracellular

Figure A.1: *E. coli* Core metabolism summary

A.1.2 Reactions

```
ACALD : acald_c + coa_c + nad_c <=> accoa_c + h_c + nadh_c
ACALDt :  acald_e <=> acald_c
ACKr :  ac_c + atp_c <=> actp_c + adp_c
ACONTa :  cit_c <=> acon_C_c + h2o_c
```

```
ACONTb :  acon_C_c + h2o_c <=> icit_c
ACt2r :  ac_e + h_e <=> ac_c + h_c
ADK1 :  amp_c + atp_c <=> 2.0 adp_c
AKGDH : akg_c + coa_c + nad_c -> co2_c + nadh_c + succoa_c
AKGt2r :  akg_e + h_e <=> akg_c + h_c
ALCD2x :  etoh_c + nad_c <=> acald_c + h_c + nadh_c
ATPM : atp_c + h2o_c -> adp_c + h_c + pi_c
ATPS4r :  adp_c + 4.0 h_e + pi_c <=> atp_c + h2o_c + 3.0 h_c
Biomass_Ecoli_core :  1.496 3pg_c + 3.7478 accoa_c + 59.81 atp_c + 0.361
e4p_c + 0.0709 f6p_c + 0.129 g3p_c + 0.205 g6p_c + 0.2557 gln__L_c + 4.9414
glu__L_c + 59.81 h2o_c + 3.547 nad_c + 13.0279 nadph_c + 1.7867 oaa_c +
0.5191 pep_c + 2.8328 pyr_c + 0.8977 r5p_c -> 59.81 adp_c + 4.1182 akg_c
+ 3.7478 coa_c + 59.81 h_c + 3.547 nadh_c + 13.0279 nadp_c + 59.81 pi_c
CO2t :  co2_e <=> co2_c
CS : accoa_c + h2o_c + oaa_c -> cit_c + coa_c + h_c
CYTBD : 2.0 h_c + 0.5 o2_c + q8h2_c -> h2o_c + 2.0 h_e + q8_c
D_LACt2 :  h_e + lac__D_e <=> h_c + lac__D_c
ENO : 2pg_c <=> h2o_c + pep_c
ETOHt2r :  etoh_e + h_e <=> etoh_c + h_c
EX_ac_e :  ac_e ->
EX_acald_e :  acald_e ->
EX_akg_e :  akg_e ->
EX_co2_e :  co2_e <=>
EX_etoh_e :  etoh_e ->
EX_for_e :  for_e ->
EX_fru_e :  fru_e ->
EX_fum_e :  fum_e ->
EX_glc__D_e :  glc__D_e <=>
EX_gln__L_e :  gln__L_e ->
EX_glu__L_e :  glu__L_e ->
EX_h_e :  h_e <=>
EX_h2o_e :  h2o_e <=>
EX_lac__D_e :  lac__D_e ->
EX_mal__L_e :  mal__L_e ->
EX_nh4_e :  nh4_e <=>
EX_o2_e :  o2_e <=>
EX_pi_e :  pi_e <=>
EX_pyr_e :  pyr_e ->
EX_succ_e :  succ_e ->
FBA : fdp_c <=> dhap_c + g3p_c
FBP : fdp_c + h2o_c -> f6p_c + pi_c
FORt2 :  for_e + h_e -> for_c + h_c
FORti :  for_c -> for_e
FRD7 :  fum_c + q8h2_c -> q8_c + succ_c
```

```
FRUpts2 :  fru_e + pep_c -> f6p_c + pyr_c
FUM : fum_c + h2o_c <=> mal__L_c
FUMt2_2 :  fum_e + 2.0 h_e -> fum_c + 2.0 h_c
G6PDH2r :  g6p_c + nadp_c <=> 6pgl_c + h_c + nadph_c
GAPD : g3p_c + nad_c + pi_c <=> 13dpg_c + h_c + nadh_c
GLCpts :  glc__D_e + pep_c -> g6p_c + pyr_c
GLNS : atp_c + glu__L_c + nh4_c -> adp_c + gln__L_c + h_c + pi_c
GLNabc :  atp_c + gln__L_e + h2o_c -> adp_c + gln__L_c + h_c + pi_c
GLUDy :  glu__L_c + h2o_c + nadp_c <=> akg_c + h_c + nadph_c + nh4_c
GLUN : gln__L_c + h2o_c -> glu__L_c + nh4_c
GLUSy :  akg_c + gln__L_c + h_c + nadph_c -> 2.0 glu__L_c + nadp_c
GLUt2r :  glu__L_e + h_e <=> glu__L_c + h_c
GND : 6pgc_c + nadp_c -> co2_c + nadph_c + ru5p__D_c
H2Ot :  h2o_e <=> h2o_c
ICDHyr :  icit_c + nadp_c <=> akg_c + co2_c + nadph_c
ICL : icit_c -> glx_c + succ_c
LDH_D : lac__D_c + nad_c <=> h_c + nadh_c + pyr_c
MALS : accoa_c + glx_c + h2o_c -> coa_c + h_c + mal__L_c
MALt2_2 :  2.0 h_e + mal__L_e -> 2.0 h_c + mal__L_c
MDH : mal__L_c + nad_c <=> h_c + nadh_c + oaa_c
ME1 :  mal__L_c + nad_c -> co2_c + nadh_c + pyr_c
ME2 :  mal__L_c + nadp_c -> co2_c + nadph_c + pyr_c
NADH16 :  4.0 h_c + nadh_c + q8_c -> 3.0 h_e + nad_c + q8h2_c
NADTRHD : nad_c + nadph_c -> nadh_c + nadp_c
NH4t :  nh4_e <=> nh4_c
O2t :  o2_e <=> o2_c
PDH : coa_c + nad_c + pyr_c -> accoa_c + co2_c + nadh_c
PFK : atp_c + f6p_c -> adp_c + fdp_c + h_c
PFL : coa_c + pyr_c -> accoa_c + for_c
PGI : g6p_c <=> f6p_c
PGK : 3pg_c + atp_c <=> 13dpg_c + adp_c
PGL : 6pgl_c + h2o_c -> 6pgc_c + h_c
PGM : 2pg_c <=> 3pg_c
PIt2r :  h_e + pi_e <=> h_c + pi_c
PPC : co2_c + h2o_c + pep_c -> h_c + oaa_c + pi_c
PPCK : atp_c + oaa_c -> adp_c + co2_c + pep_c
PPS : atp_c + h2o_c + pyr_c -> amp_c + 2.0 h_c + pep_c + pi_c
PTAr :  accoa_c + pi_c <=> actp_c + coa_c
PYK : adp_c + h_c + pep_c -> atp_c + pyr_c
PYRt2 :  h_e + pyr_e <=> h_c + pyr_c
RPE : ru5p__D_c <=> xu5p__D_c
RPI : r5p_c <=> ru5p__D_c
SUCCt2_2 :  2.0 h_e + succ_e -> 2.0 h_c + succ_c
SUCCt3 :  h_e + succ_c -> h_c + succ_e
```

```
SUCDi :  q8_c + succ_c -> fum_c + q8h2_c
SUCOAS : atp_c + coa_c + succ_c <=> adp_c + pi_c + succoa_c
TALA : g3p_c + s7p_c <=> e4p_c + f6p_c
THD2 :  2.0 h_e + nadh_c + nadp_c -> 2.0 h_c + nad_c + nadph_c
TKT1 :  r5p_c + xu5p__D_c <=> g3p_c + s7p_c
TKT2 :  e4p_c + xu5p__D_c <=> f6p_c + g3p_c
TPI : dhap_c <=> g3p_c
```

A.1.3 Metabolites

```
13dpg_c; 2pg_c; 3pg_c; 6pgc_c; 6pgl_c; ac_c; ac_e; acald_c; acald_e; accoa_c;
acon_C_c; actp_c; adp_c; akg_c; akg_e; amp_c; atp_c; cit_c; co2_c; co2_e;
coa_c; dhap_c; e4p_c; etoh_c; etoh_e; f6p_c; fdp_c; for_c; for_e; fru_e;
fum_c; fum_e; g3p_c; g6p_c; glc__D_e; gln__L_c; gln__L_e; glu__L_c; glu__L_e;
glx_c; h2o_c; h2o_e; h_c; h_e; icit_c; lac__D_c; lac__D_e; mal__L_c; mal__L_e;
nad_c; nadh_c; nadp_c; nadph_c; nh4_c; nh4_e; o2_c; o2_e; oaa_c; pep_c;
pi_c; pi_e; pyr_c; pyr_e; q8_c; q8h2_c; r5p_c; ru5p__D_c; s7p_c; succ_c;
succ_e; succoa_c; xu5p__D_c
```

A.1.4 External metabolites

Those are the external metabolites that we will use as input and output for this model.

```
  7:  EX_co2_e; EX_glc__D_e; EX_h_e; EX_h2o_e; EX_nh4_e; EX_o2_e; EX_pi_e;
xu5p__D_c
```

A.2 *E. coli*

This model is based in the *E. coli* whole metabolism. This is the most complex and large model that *COBRApy* implements. This is a good model to represent a complex possible metabolism that we will want to simulate.

In order to avoid excessive use of pages, we will show just the summary and the external metabolism for the next two models.

A.2.1 Model Summary

Name	iJO1366
Memory address	0x021518e4da20
Number of metabolites	1805
Number of reactions	2583
Objective expression	1.0*Ec_biomass_iJO1366_core_53p95M - 1.0*Ec_biomass_iJO1366_core_53p95M_reverse_e94eb
Compartments	Cytoplasm, Extracellular, Periplasm

Figure A.2: *E. coli* metabolism summary

A.2.2 External metabolites

25: EX_ca2_e; EX_cbl1_e; EX_cl_e; EX_co2_e; EX_cobalt2_e; EX_cu2_e; EX_fe2_e; EX_fe3_e; EX_glc_e; EX_h_e; EX_h2o_e; EX_k_e; EX_mg2_e; EX_mn2_e; EX_mobd_e; EX_na1_e; EX_nh4_e; EX_ni2_e; EX_o2_e; EX_pi_e; EX_sel_e; EX_slnt_e; EX_so4_e; EX_tungs_e; EX_zn2_e; xu5p__D_c

A.3 *Salmonella*

This model is based in the *Sallmonella* whole metabolism. This is simple but large metabolism that is very useful to contrast the difference with the other two models.

A.3.1 Model Summary

Name	Salmonella_consensus_build_1
Memory address	0x021518e57470
Number of metabolites	1802
Number of reactions	2546
Objective expression	1.0*biomass_iRR1083_metals - 1.0*biomass_iRR1083_metals_reverse_00f08
Compartments	Cytoplasm, Periplasm, Extracellular

Figure A.3: *Salmonella* metabolism summary

A.3.2 External metabolites

22: EX_ca2_e; EX_cit_e; EX_cl_e; EX_co2_e; EX_cobalt2_e; EX_cu2_e; EX_fe3_e; EX_glyc_e; EX_h2o_e; EX_h_e; EX_k_e; EX_mg2_e;

```
EX_mn2_e; EX_mobd_e; EX_nh4_e; EX_o2_e; EX_pi_e; EX_so4_e;
EX_thm_e; EX_zn2_e; xu5p__D_c
```

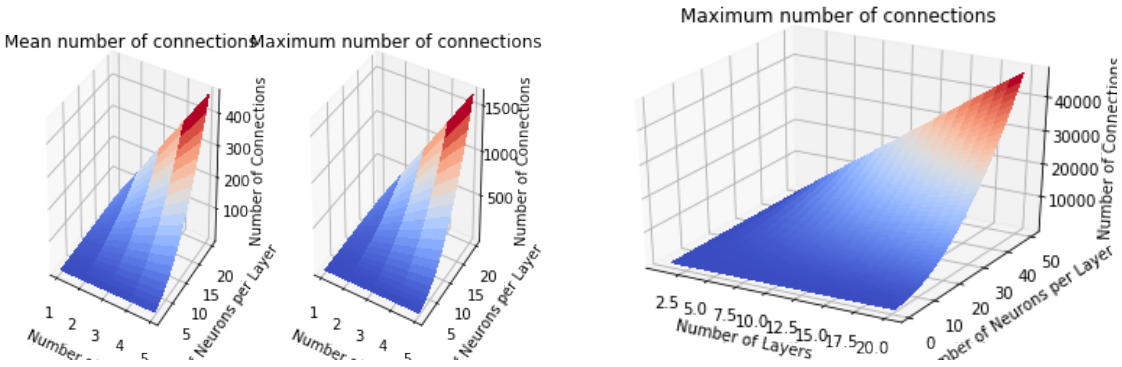
Appendix B

Results

In this annexed, we present the results that has not been attached in the body of this work because they were not as relevant as the others, or because they do not affect the decisions taken.

For example, here we have all the results for the models of *Ecoli Core* and *Salmonella*, because the *Ecoli* ones has been presented already in the body.

B.1 Number of connections



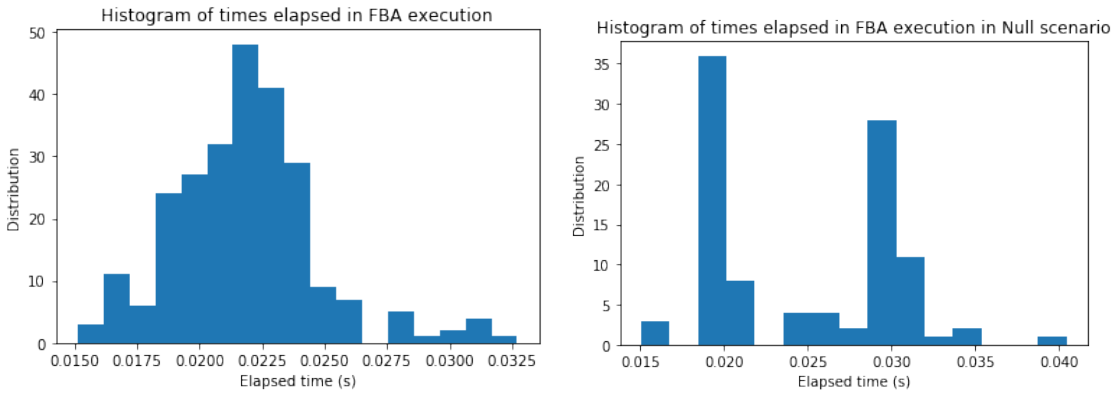
(a) Mean and Max number of connections for limit given

(b) Max number of connections for limit given

Figure B.1: Mathematical number of connections

B.2 Time

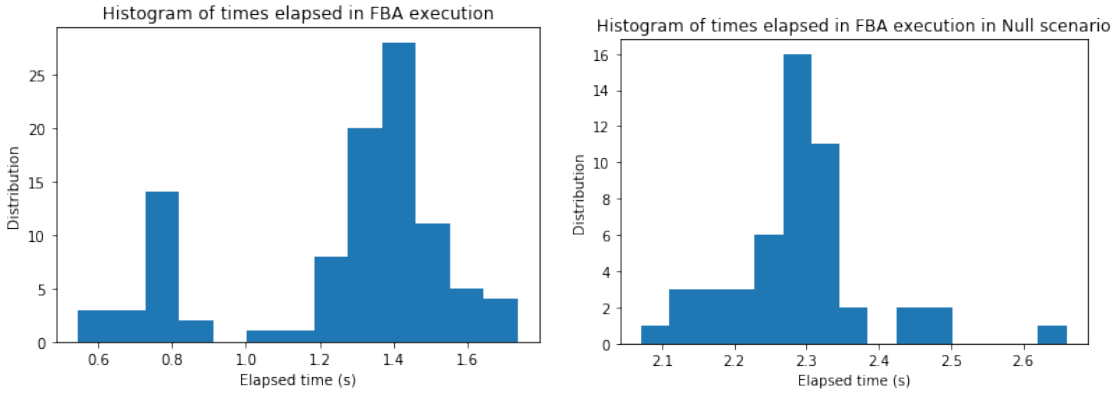
B.2.1 *Core*



(a) Mean time (b) Null input

Figure B.2: Core mean execution

B.2.2 *Salmonella*

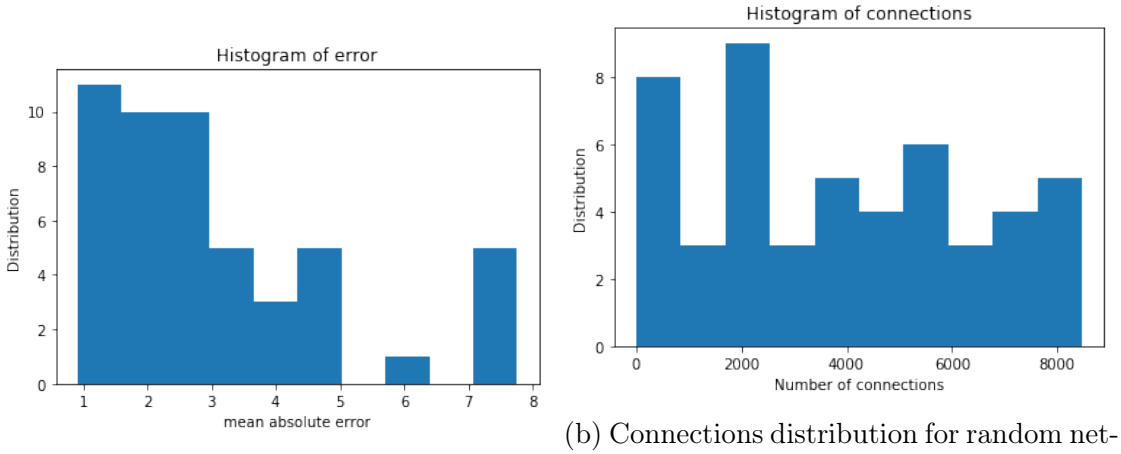


(a) Mean time (b) Null input

Figure B.3: Salmonella mean execution

B.3 Accuracy

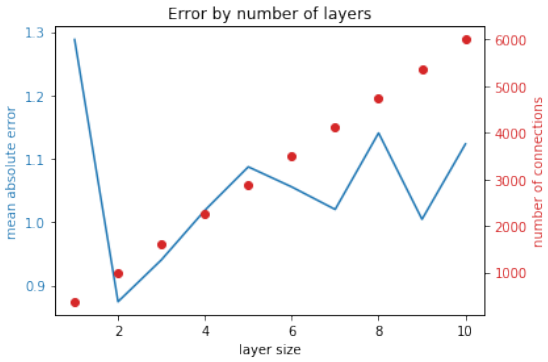
B.3.1 *Core*



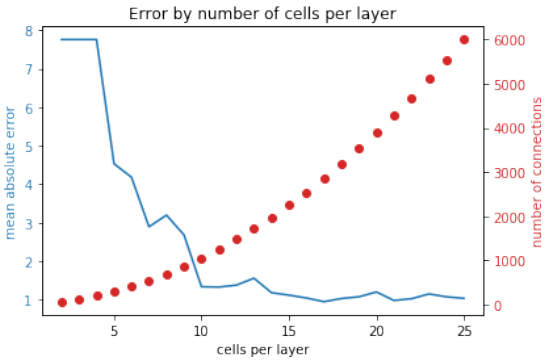
(a) Error for random networks

(b) Connections distribution for random networks

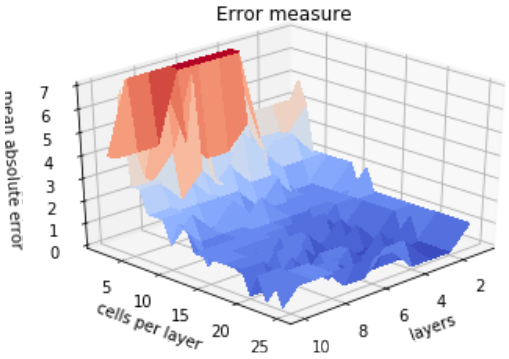
Figure B.4: Error and time estimated for random networks



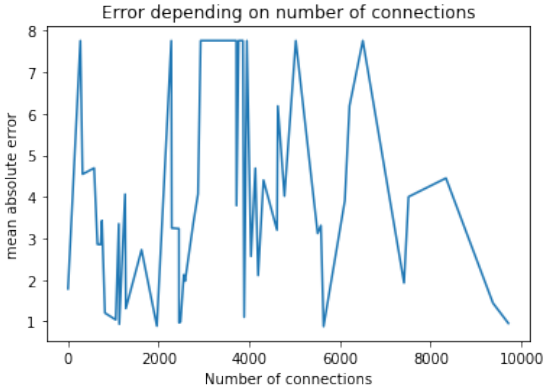
(a) Error depending on the number of layers for 25 cells per layer



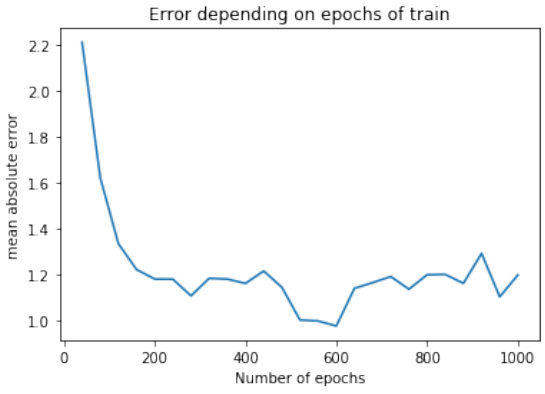
(b) Error depending on the number of cells per layer for 10 layers



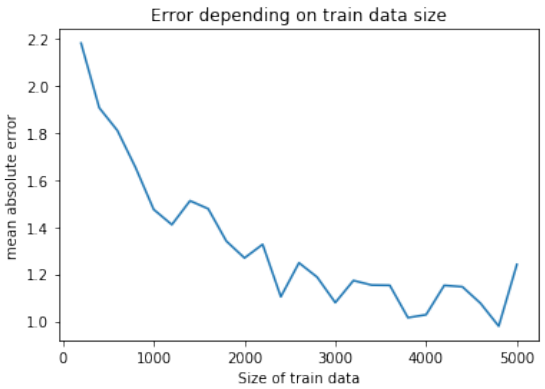
(c) Error depending on the number of layers and the cell per layer



(d) Error depending on the number of connections



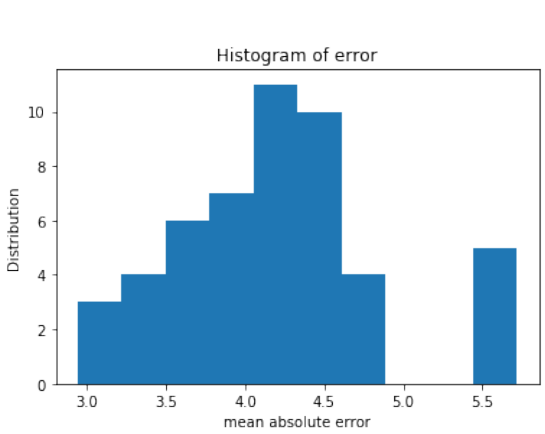
(e) Error depending on the number of epochs of training



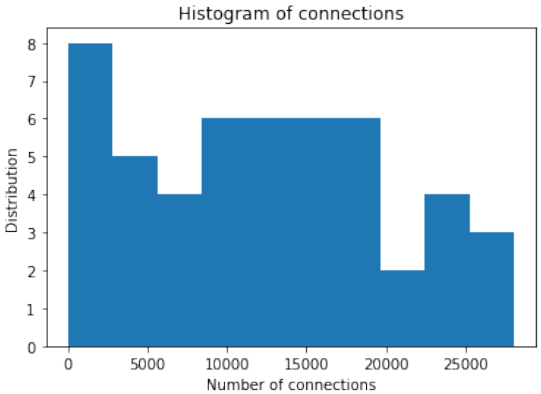
(f) Error depending on the size of the dataset

Figure B.5: Results modifying parameters

B.3.2 *Salmonella*

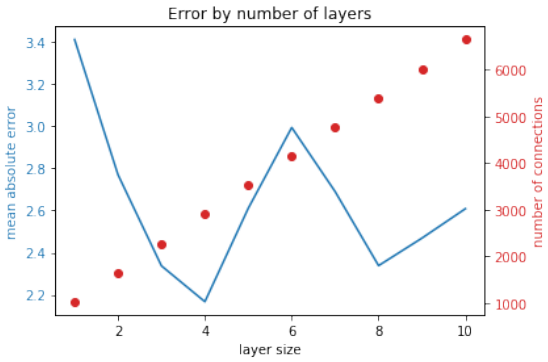


(a) Error for random networks

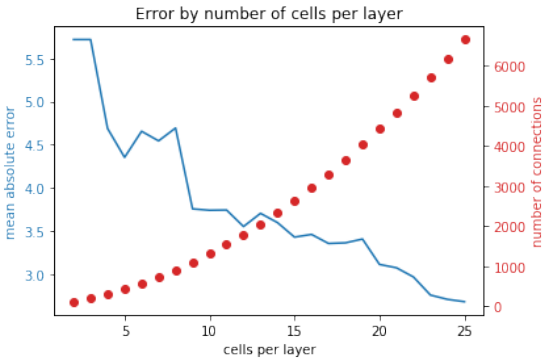


(b) Connections distribution for random networks

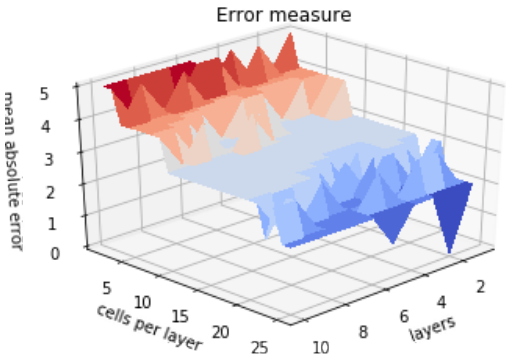
Figure B.6: Error and time estimated for random networks



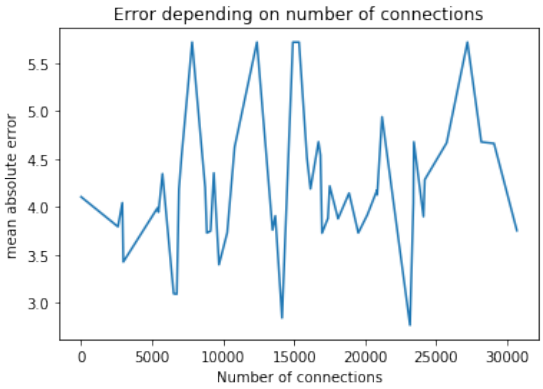
(a) Error depending on the number of layers for 25 cells per layer



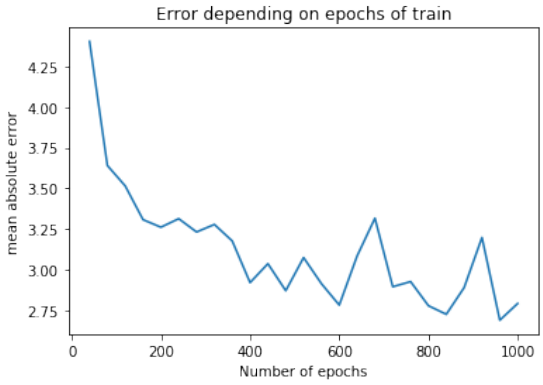
(b) Error depending on the number of cells per layer for 10 layers



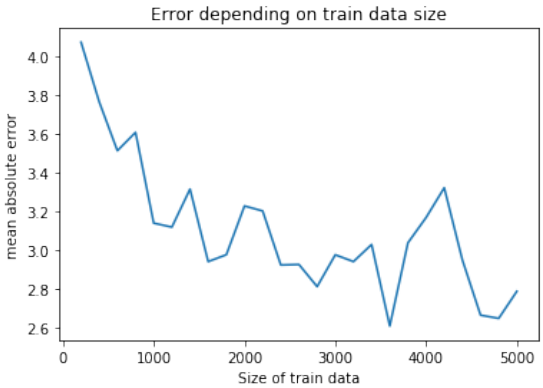
(c) Error depending on the number of layers and the cell per layer



(d) Error depending on the number of connections



(e) Error depending on the number of epochs of training

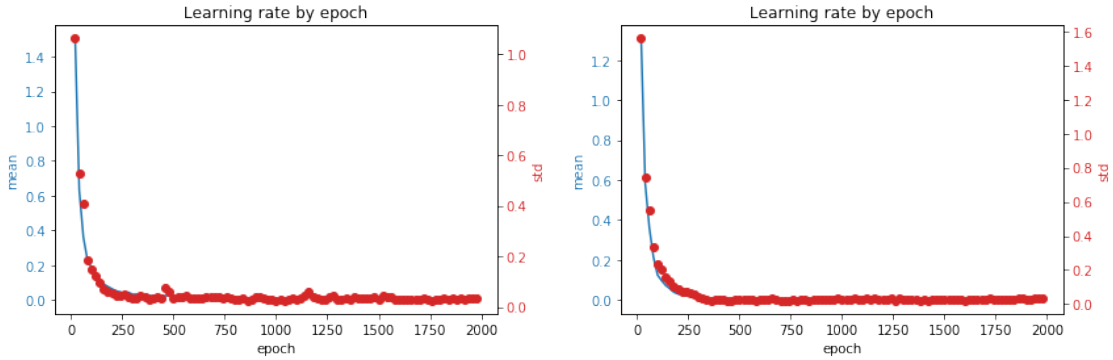


(f) Error depending on the size of the dataset

Figure B.7: Results modifying parameters

B.4 Training

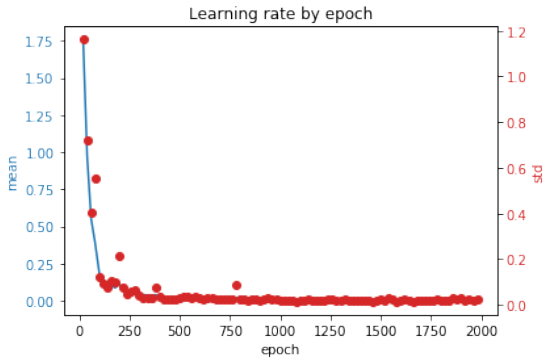
B.4.1 *Core*



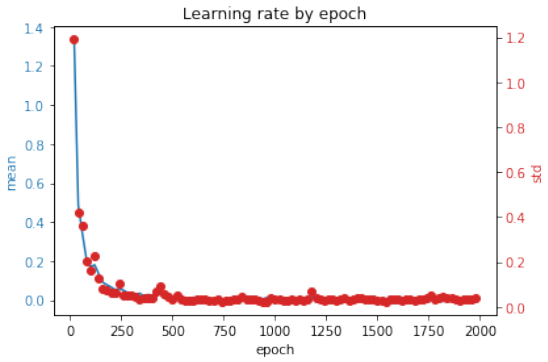
(a) Error decreased over epochs for several networks with 5000 data

(b) Error decreased over epochs for several networks with 2500 data

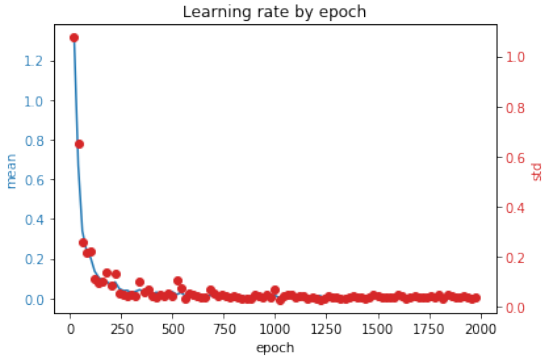
Figure B.8: Execution of 100 random neural networks to measure the error decreasing ratio.



(a) Error decreased over epochs for several networks under 1000 connections



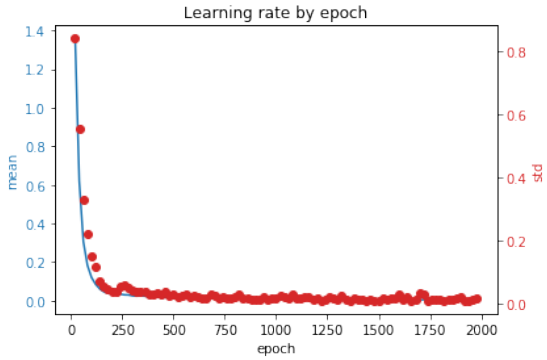
(b) Error decreased over epochs for several networks between 1000 and 2000 connections



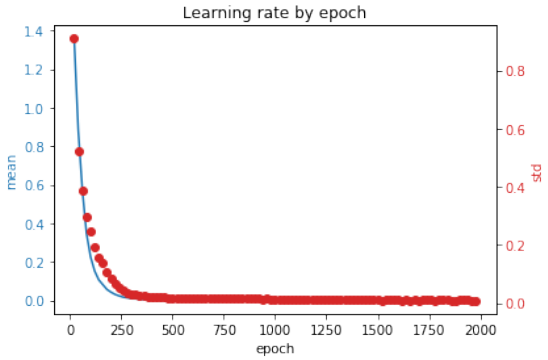
(c) Error decreased over epochs for several networks over 2000 connections

Figure B.9: Execution of 100 random neural networks to measure the error decreasing ratio depending on the number of connections.

B.4.2 *Salmonella*



(a) Error decreased over epochs for several networks with 5000 data



(b) Error decreased over epochs for several networks with 2500 data

Figure B.10: Execution of 100 random neural networks to measure the error decreasing ratio.

Appendix C

Neural Network implementation using Keras

In this annex we show the general implementation for all the neural networks used along this project.

There are some native inherited methods as *fit* or *evaluate*, and other new methods implemented as *percentageEvaluation* or *connectionNumber*.

```
import tensorflow as tf
import numpy as np


# neural network keras wrapper
class NeuralNetworkModel():

    def __init__(self, layers, inputCells, outputCells):

        model = tf.keras.Sequential()
        for i, l in enumerate(layers):
            if i == 0:
                model.add(tf.keras.layers.Dense(l,
                                                input_shape=(inputCells,),
                                                activation='relu'))
            else:
                model.add(tf.keras.layers.Dense(l,
                                                activation='relu'))

        model.add(tf.keras.layers.Dense(outputCells,
                                        kernel_initializer='normal'))

        model.compile(optimizer='adam',
                      loss='mse',       # mean squared error
                      metrics=['mae'])  # mean absolute error

        self.model = model
```

```
28          self.array = [inputCells] + layers + [outputCells]
29
30
31      def predict(self, data):
32          return self.model.predict(data)
33
34      def fit(self, data, value, epochs, batchSize, verbose=0):
35          return self.model.fit(data.values,
36                                value.values,
37                                epochs=epochs,
38                                batch_size=batchSize,
39                                verbose=verbose)
40
41      def evaluate(self, data, value, batchSize, verbose=0):
42          return self.model.evaluate(data.values, value.values,
    batch_size=batchSize, verbose=verbose)
43
44      def connectionNumber(self):
45          val = 0
46          for i in range(len(self.array)-1):
47              val += self.array[i] * self.array[i+1]
48          return val
49
50      def percentageEvaluation(self, data, value, verbose=0):
51
52          result = self.predict(data)
53
54          outCells = self.getOutputCells()
55          maxMet = [0.001 for i in range(outCells)]
56          totalError = [[] for i in range(outCells)]
57          totalErrorMean = []
58          totalErrorStd = []
59
60          for i, d in enumerate(value.values):
61              for met in range(outCells):
62                  if d[met] > maxMet[met]:
63                      maxMet[met] = d[met]
64                  totalError[met].append(np.abs(d[met] - result[i][met]))
65
66
67          for i in range(outCells):
68              totalErrorMean.append(np.mean(totalError[i])/maxMet[i])
69              totalErrorStd.append(np.std(totalError[i])/maxMet[i])
70
71          return totalErrorMean, totalErrorStd, maxMet,
```

Listing C.1: neural network implementation

Appendix D

FBA implementation using Cobra

In this annex we show the general implementation for all the *FBA* data used along this project.

These datasets have been calculated by uniform random distributions of every metabolite.

```
# cobra libraries
import cobra.test
from cobra import Model, Reaction

# std math libraries
import numpy as np
import tensorflow as tf
import pandas as pd

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# ------------------------------------------------------------------------

# COBRA
# return core ecoli model
def ecoliCoreModel():
    return cobra.test.create_test_model("textbook")

# return ecoli complex model
def ecoliModel():
    return cobra.test.create_test_model("ecoli")

# return salmonella model
def salmonellaModel():
    return cobra.test.create_test_model("salmonella")

# ------------------------------------------------------------------------

# DATA
```

```
30 # create random data of input
31 def getRandomDataAndValues(minValue, maxValue, size, model, metabolites
       , pond=100):
32
33     inputCells = len(metabolites)
34
35     data = []
36     value = []
37
38     for i in range(size):
39         data.append(np.random.rand(inputCells) * (maxValue - minValue)
       + minValue)
40         value.append(_modelPointModifying(model, data[-1], metabolites,
        pond))
41
42     return pd.DataFrame(data=data), pd.DataFrame(data=value)
43
44
45 # solution for a model and some limits in metabolites
46 def _modelPointModifying(model, point, metabolites, pond=100):
47
48     for i, met in enumerate(metabolites):
49         model.reactions.get_by_id(met).lower_bound = -point[i]
50
51     res = []
52     for i, met in enumerate(metabolites):
53         res.append(point[i] + model.optimize().fluxes.get(met))
54
55     res.append(model.optimize().objective_value*pond)
56
57     return res
```

Listing D.1: FBA data implementation

Bibliography

[1] Jong Min Lee, Erwin P. Gianchandani, and Jason A. Papin. Flux balance analysis in the era of metabolomics. *Briefings in Bioinformatics*, 7(2):140–150, 04 2006.

[2] Martín Eduardo Gutiérrez Pescarmona. A new agent-based platform for simulating multicellular biocircuits with conjugative plasmids. 2017.

[3] Jayesh Bapu Ahire. The artificial neural network handbook, part 1. https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4, 2018.

[4] Jeffrey D. Orth, Ines Thiele, and Bernhard Ø Palsson. What is flux balance analysis? *Nature Biotechnology*, 28:245 EP –, Mar 2010.

[5] Xin-She Yang. *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010.

[6] Jeremy S. Edwards, Rafael U. Ibarra, and Bernhard O. Palsson. In silico predictions of escherichia coli metabolic capabilities are consistent with experimental data. *Nature Biotechnology*, 19(2):125–130, 2001.

[7] Romilde Manzoni, Arturo Urrios, Silvia Velazquez-Garcia, Eulàlia de Nadal, and Francesc Posas. Synthetic biology: insights into biological computation. *Integrative Biology*, 8(4):518–532, 04 2016.

[8] I. Cody MacDonald and Tara L. Deans. Tools and applications in synthetic biology. *Advanced Drug Delivery Reviews*, 105:20 – 34, 2016. Synthetic Biology: Innovative approaches for pharmaceutics and drug delivery.

[9] Yiannis N. Kaznessis. Computational methods in synthetic biology. *Biotechnology Journal*, 4(10):1392–1405, 2009.

[10] Seunghee S. Jang, Kevin T. Oishi, Robert G. Egbert, and Eric Klavins. Specification and simulation of synthetic multicelled behaviors. *ACS Synthetic Biology*, 1(8):365–374, 2012. PMID: 23651290.

[11] Martín Gutiérrez, Paula Gregorio-Godoy, Guillermo Pérez del Pulgar, Luis E. Muñoz, Sandra Sáez, and Alfonso Rodríguez-Patón. A new improved and extended version of the multicell bacterial simulator gro. *ACS Synthetic Biology*, 6(8):1496–1508, 2017. PMID: 28438021.

[12] William?R Harcombe, William?J Riehl, Ilija Dukovski, Brian?R Granger, Alex Betts, Alex?H Lang, Gracia Bonilla, Amrita Kar, Nicholas Leiby, Pankaj Mehta, Christopher?J Marx, and Daniel Segrè. Metabolic resource allocation in individual microbes determines ecosystem interactions and spatial dynamics. *Cell Reports*, 7(4):1104–1115, May 2014.

[13] Eugen Bauer, Johannes Zimmermann, Federico Baldini, Ines Thiele, and Christoph Kaleta. Bacarena: Individual-based metabolic modeling of heterogeneous microbes in complex communities. *PLOS Computational Biology*, 13(5):1–22, 05 2017.

[14] D. R. Hush and B. G. Horne. Progress in supervised neural networks. *IEEE Signal Processing Magazine*, 10(1):8–39, Jan 1993.

[15] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[16] D. F. Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, Nov 1991.

[17] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[18] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, 1998.

[19] F. Z. Brill, D. E. Brown, and W. N. Martin. Fast generic selection of features for neural network classifiers. *IEEE Transactions on Neural Networks*, 3(2):324–328, March 1992.

[20] *TensorFlow* developers. *TensorFlow* api documentation. `https://www.tensorflow.org/api_docs`.

[21] *Keras* developers. *Keras* documentation. `https://keras.io/`.

[22] *COBRApy* developers. Documentation for *COBRApy*. `https://cobrapy.readthedocs.io/en/latest/`.

[23] *Numpy* developers. *Numpy* and *Scipy* documentation. `https://docs.scipy.org/doc/`.

[24] *Pandas* developers. *Pandas* documentation. `https://pandas.pydata.org/pandas-docs/stable/`.

[25] Ali Ebrahim, Joshua A. Lerman, Bernhard O. Palsson, and Daniel R. Hyduke. Cobrapy: Constraints-based reconstruction and analysis for python. *BMC Systems Biology*, 7(1), Aug 2013.

[26] D Pham and Dervis Karaboga. *Intelligent optimisation techniques. Genetic algorithms, tabu search, simulated annealing and neural networks*. 01 2000.

[27] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477 Vol. 2, July 1999.

[28] Krzysztof Socha and Christian Blum. An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training. *Neural Computing and Applications*, 16(3):235–247, May 2007.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[30] L. Ingber. Simulated annealing: Practice versus theory. *Mathematical and Computer Modelling*, 18(11):29 – 57, 1993.

[31] Dolores Barrios Rolanía, Guillermo Delgado Martínez, and Daniel Manrique. Multilayered neural architectures evolution for computing sequences of orthogonal polynomials. *Annals of Mathematics and Artificial Intelligence*, 84(3):161–184, Dec 2018.

[32] H. K. Lam, S. H. Ling, F. H. F. Leung, and P. K. S. Tam. Tuning of the structure and parameters of neural network using an improved genetic algorithm. In *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.37243)*, volume 1, pages 25–30 vol.1, Nov 2001.

[33] Randall S. Sexton, Robert E. Dorsey, and John D. Johnson. Optimization of neural networks: A comparative analysis of the genetic algorithm and simulated annealing. *European Journal of Operational Research*, 114(3):589 – 601, 1999.

[34] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3):365–396, Sep 2010.

[35] Daniel Junkle. A summary and comparison of moea algorithms. http://www.ccs.neu.edu/home/kunkle/papers/techreports/moeaComparison.pdf, May 2005.

[36] J. Knowles and D. Corne. The pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimisation. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 1, pages 98–105 Vol. 1, July 1999.

[37] Jeffrey D. Orth, Tom M. Conrad, Jessica Na, Joshua A. Lerman, Hojung Nam, Adam M. Feist, and Bernhard Ø Palsson. A comprehensive genome-scale reconstruction of escherichia coli metabolism–2011. *Molecular systems biology*, 7:535–535, Oct 2011. 21988831[pmid].