Zébulon Goriely

Simulating Natural Language Learning and Evolution

Computer Science Tripos – Part II

Queens' College

May 2020

Declaration of Originality

I, Zébulon Goriely of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Zébulon Goriely of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Zébulon Goriely

Date: May 8, 2020

Acknowledgements

Thanks to Paula Buttery and Andrew Caines for supervising me, providing amazing feedback on my project and for giving me an incredible insight into the world of Computer Science research. Thank you to Alice Wenban for putting up with my enthusiasm about this project. Thank you to Alastair Beresford for Directing my Studies throughout the last three years and providing feedback on my work. Thank you to my parents for their continuous and constant support through my studies.

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

Proforma

Candidate number:	**2400G**
Project Title:	**Simulating Language Learning and Evolution**
Examination:	**Computer Science Tripos – Part II, May 2020**
Word Count:	**11150**[1]
Final Line Count:	**3125**[2]
Project Originator:	The Author
Project Supervisors:	Prof. Paula Buttery and Dr. Andrew Caines

Original Aims of the Project

Language has evolved and therefore probably gave an evolutionary advantage to the individuals that exhibited it. According to Cangelosi and Parisi (1998), it is difficult to investigate the evolutionary origin of language and the selective pressures that may have originated language due to the limited evidence available. In the referenced paper, they describe using a simulated world to investigate the effect of introducing language to a population of entities controlled by neural networks. These entities interact with an environment of mushrooms that are edible and poisonous and must correctly categorise them to survive. Exploring this simulation is the basis of my project.

Work Completed

I successfully implemented the simulation; comparing three different populations. To simulate evolution, I implemented a genetic algorithm. Through algorithmic analysis, I discovered key optimisations. The state of the simulation was saved at each generation in order to: i) plot the fitness of the populations; ii) plot the quality of the language produced; iii) investigate behavioural tests. I compared these findings with Cangelosi and Parisi (1998) to confirm the evolutionary advantage of language. I then conducted my own analysis to determine the robustness of the simulation, criticise arbitrary decisions made by Cangelosi and Parisi (1998) and demonstrate a correlation between population behaviour and fitness.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`, counting just the five main chapters.

[2] This line count was computed using: ``cat `find . -name "*.py"` | wc -l``

Special Difficulties

None.

Contents

1 Introduction **1**

1.1 Contributions 1

1.2 Project Overview 2

2 Preparation **3**

2.1 Mushroom World 3

2.2 Entities 4

2.2.1 Feed-Forward Neural Networks 4

2.2.2 Genetic Algorithm 5

2.2.3 Population Types 6

2.3 Simulation Analysis 7

2.4 Requirements Analysis 8

2.5 Starting Point 9

2.6 Software Engineering 9

2.6.1 Languages and Libraries 9

2.6.2 Project Management 10

2.6.3 Version Control 10

2.6.4 Development Tools 10

2.7 Summary 11

3 Implementation **12**

3.1 High-level Overview 12

3.2 Environment 12

3.2.1 Mushrooms 13

3.2.2 Representation of the World 13

3.2.3 Environment Functionality 15

3.2.4 Testing 15

3.3 Entities 15
3.3.1 Abstract Specification 16
3.3.2 Neural Network Entity 16
3.4 Simulation 18
3.4.1 Initialising the Simulation 18
3.4.2 Single-Entity Simulation Run 18
3.4.3 Genetic Algorithm 19
3.4.4 Running the Simulations Using HPC 20
3.4.5 Optimisations 20
3.4.6 Interactivity 22
3.4.7 Data Recording 22
3.5 Simulation Analysis 23
3.5.1 QI Score 23
3.6 Summary 24

4 Evaluation **25**
4.1 Initial Analysis 25
4.1.1 Population Fitness 25
4.1.2 Language Analysis 27
4.2 Behavioural Analysis 28
4.3 Exploring Simulation Parameters 31
4.3.1 Neural Network Depth 32
4.3.2 Epoch Length 33
4.3.3 Genetic Parameters 33
4.4 Summary 35

5 Conclusion **36**
5.1 Lessons Learnt 36
5.2 Further Work 37

Bibliography **37**

A World Representation Benchmark **39**
A.1 world_benchmark.py 39

B Simulation Interactivity **41**

C Project Proposal 44

List of Figures

2.1 An entity (▲) in the simulation environment with edible (E) and poisonous (P) mushrooms. 4

2.2 Commits to the Github repository over the duration of the project. 10

3.1 Core modules for my project 13

3.2 UML Diagram for the Simulating Module 14

3.3 Fully-connected ANN to control entity behaviour 17

3.4 Time taken to run 1000 generations of the genetic algorithm on a population with no language for each optimisation, averaged over ten runs. 22

3.5 Time taken to run each generation of the genetic algorithm on a population with no language for each optimisation, averaged over ten runs. 23

4.1 Average fitness of each population, averaged over 10 replications, with a **ReLU** hidden layer activation. 26

4.2 Average fitness of each population, averaged over 5 replications. Source: Cangelosi and Parisi (1998). 26

4.3 Average fitness of each population, averaged over 10 replications, with an **identity** hidden layer activation. 26

4.4 Average fitness of each population, averaged over 10 replications, with a **sigmoid** hidden layer activation. 26

4.5 Frequency distribution of the possible signals produced by all individuals in 10 generations in one replica of a simulation with an **evolved language.** 28

4.6 Average fitness for one replica of a population with an **evolved language**. Also shown is the Quality Index score of the language produced by this population. 29

4.7 Average fitness of 10 replications of the **no language** population, with an **identity** hidden layer activation. 30

4.8 Average fitness of 10 replications of the **no language** population, with a **ReLU** hidden layer activation. 30

4.9 Heatmap of the weights and biases of 100 entities taken from generation 2000. 30

4.10 Heatmap of a randomly chosen entity taken from increasing generations. 30

4.11 Heatmaps of the flattened neural networks of a randomly chosen entity from generation 2000 of ten replicas. 31

4.12 The behaviour of two entities approaching a poisonous mushroom (P). The entities are randomly chosen from generation 2000 of two **no language** populations with an average fitness of 140 (left) and 250 (right). 32

4.13 Average fitness of each population, averaged over 10 replications, with two hidden layers of five units each. 33

4.14 Average fitness of each population, averaged over 10 replications, with three hidden layers of ten units, five units and ten units. 33

4.15 Average fitness of each population, averaged over 10 replications, with simulations consisting of 30 epochs of length 15. 34

4.16 Average fitness of each population, averaged over 10 replications, with simulations consisting of 10 epochs of length 75. 34

4.17 Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 20%. 34

4.18 Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 50%. 34

4.19 Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 10%. 35

4.20 Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 5%. 35

B.1 Result of running `python3 -m simulating.simulation -h` to display the help page for the command-line interface of the simulation module. The required parameters are the foldername (for storing results) and the language type, all other parameters have a default value. 42

B.2 Result of running `python3 -m analysis.plotting -h` to display the help page for the command-line interface of the analysis module. The required parameters are the foldername (where the results are found in order to plot) and the type of graph to display. 42

B.3 Visualisation of running a default simulation for a **no language** population using the interactivity flag (`-i`). The left side shows the terminal output and the right side shows the live fitness graph. 42

B.4 Visualisation of watching the behaviour of a single entity in a population, selected from an interactive simulation by entering `watch 0`. 43

Chapter 1

Introduction

Humans evolved to understand and produce language, and according to theories of natural selection we therefore infer that the ability to use language gave an evolutionary advantage to those individuals who exhibited it. The field of language evolution is diverse and spans many scientific disciplines, from evolutionary biology and neuroscience to psycholinguistics and cultural anthropology. One of the main problems with studying the evolution of language is the relative abundance of theories compared to the limited empirical evidence since anatomically-modern humans emerged in the fossil record 200,000 years ago (Fleagle et al., 2008). It is not known when, during or prior to this period, language actually emerged. To our knowledge, language evolution is a once-occurring, long-term and complex system making it very difficult to study, but computer simulations may provide a means by which to examine the theories surrounding it.

Computer simulations are *"theories of the empirical phenomena that are simulated"* (Cangelosi and Parisi, 2002). Simulations encompass the hypotheses of the theory and produce empirical predictions that can then be evaluated against known phenomena (Cavalli-Sforza, 1997). Simulations can act as virtual laboratories where impossible experiments (such as language evolution) can be run, controlling parameters that could never be controlled in real life, such as evolutionary or environmental factors. Simulations can also act as quantitative verifiers for the internal validity of vague and ambiguous theories, requiring explicit definitions of key assumptions. Finally, simulations can be used as a tool for studying complex systems. These are composed of many interacting entities that produce global properties that cannot be predicted even with complete knowledge of the system. A noteworthy example is Conway's Game of Life, a cellular automaton with very simple rules in which it is possible to build a universal Turing Machine (Rendell, 2002).

Language itself is a complex system and it has been shown that the bottom-up approach of simulations can generate key insights (Langton, 1997). In this dissertation, I present how simulation can be used to explore the genetic advantage of populations of entities who exhibit language over those who do not.

1.1 Contributions

Computer simulations have been applied to a variety of interesting questions in the field of language evolution. Seeking to investigate the emergence of syntactic universals, Kirby and Hurford (2002) presented the Iterated Learning Model (ILM), a means of simulating cultural transmission of language.

They discuss the intersection of learning, cultural evolution and biological evolution as defining the emergence of language and use the ILM to explain the emergence of compositionality, irregularity and frequency properties of language.

To study the emergence of shared vowel systems, De Boer (1997) created a population-based language game model involving language games and genetic algorithms. He explores an apparent bias towards vowel systems that reflect the structure of human vowel systems, discussing how this is explained by the "optimisation of acoustic distinctiveness" from an information-theoretic perspective.

Parisi and Cangelosi (2002) discuss the use of a single unified simulation for the investigation of research questions surrounding the evolution of language. In particular, Cangelosi and Parisi (1998) introduce the "toy mushroom world" simulation for investigating how the evolution of categorisation abilities is linked to the evolution of communication signals for distinguishing these categories. I choose to replicate this work because of one of the broad questions that it tackles: how language can evolve when it has a purely informative function and so is advantageous only to the receiver and not the producer.

1.2 Project Overview

In my project I re-implement the "toy mushroom world" simulation described by Cangelosi and Parisi (1998). This involved the following contributions:

- Researching relevant algorithms and data structures (Chapter 2).
- Taking a professional approach to managing my project (Chapter 2).
- Creating three populations of feed-forward neural networks from scratch (Chapter 3).
- Implementing the genetic algorithm used to evolve the populations (Chapter 3).
- Develop optimisations to reduce the time taken to run the simulation (Chapter 3).
- Creating a suite of data gathering tools, interactivity features and analysis methods (Chapter 3).
- Comparing population fitness and languages production to the results of Cangelosi and Parisi to verify that language is a useful adaptation (Chapter 4).
- Criticising the arbitrary design decisions made by Cangelosi and Parisi by altering simulation parameters (Chapter 4).
- Conducting original analyses of the simulation with respect to convergence to different exploratory behaviours (Chapter 4).
- Reflecting on the importance of tests and the utility of scripting (Chapter 5).
- Discussing future directions that this work could take (Chapter 5).

Chapter 2

Preparation

This chapter covers the background to my project in Sections 2.1 to 2.3, then evaluates the requirements for the project in Section 2.4. I discuss the starting point of my project in Section 2.5 and the software engineering techniques used in Section 2.6.

2.1 Mushroom World

To explore how the introduction of language may affect a population's fitness, we need a simulated environment in which to observe the effect of these changes. The "mushroom world" described by Cangelosi and Parisi (1998) was inspired by the use of signals to communicate information about food location and quality present in many species.

In the simulation, organisms are represented by *entities*. The entities live in an environment populated by two different types of mushroom; edible and poisonous. The entities reproduce based on their ability to eat edible mushrooms and avoid the poisonous ones. They need to learn to categorise the two mushrooms and respond accordingly by moving towards and eating the edible mushrooms and moving away from the poisonous mushrooms.

To ensure this categorisation is not trivial for the entities, the mushrooms have different properties. Edible mushrooms resemble each other but are not identical and likewise for poisonous mushrooms.

Each entity will live in an environment of 20×20 cells containing 20 randomly distributed mushrooms; 10 of which are edible and 10 of which are poisonous. They are able to explore this world during 15 epochs of 50 simulation cycles each. Between each epoch the world is reset; the entity is placed again in a new environment with 20 new randomly distributed mushrooms. An example of this world can be seen in Figure 2.1.

The constants chosen by Cangelosi and Parisi are fairly arbitrary. In later sections I will explore the effects of changing them, but for now I can intuitively explain some of these choices. The 15 epochs provide a large enough sample size of environments for entity behaviour to be significant. The 50 cycle constraint captures limited lifetimes, forcing entities to learn optimal strategies to maximise the number of edible mushrooms eaten. A full list of constants can be seen in Table 2.1.

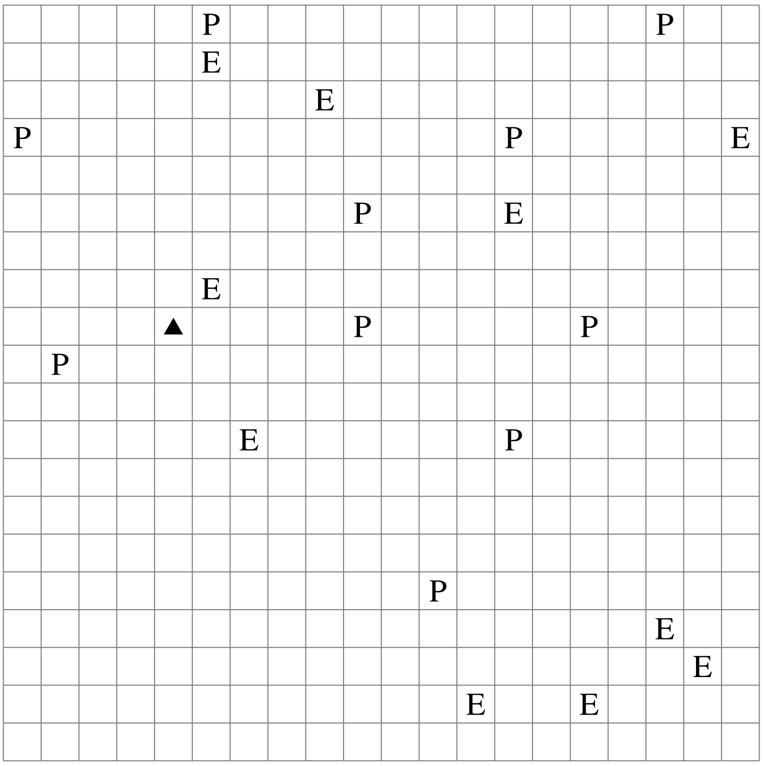


Figure 2.1: An entity (▲) in the simulation environment with edible (E) and poisonous (P) mushrooms.

2.2 Entities

Entities can *perceive* their surroundings and *act* accordingly. Perception is composed of three senses; the entity can sense the *direction* of the nearest mushroom (a 'smelling' sense), can see the *properties* of mushrooms it is adjacent to (a 'visual' sense) and can receive *signals* from other entities (an 'auditory' sense). The adjacency restriction to the visual sense means that without additional signals, entities must approach mushrooms in order to be able to categorise them.

Action consists of two possible responses; the *movement* of the entity and the *signal* it produces. The movement is restricted to four options; moving one cell forwards, turning 90° left, turning 90° right and doing nothing. To eat a mushroom, the entity moves into the mushroom's cell. The signal is used to communicate information to other entities when we add language to the simulation.

The simulation should also incorporate evolution; some process by which the fittest entities of a species reproduce to pass on their behaviour to a new generation, with some degree of mutation. This will allow a population's average fitness to improve over many generations.

2.2.1 Feed-Forward Neural Networks

Consistent with Cangelosi and Parisi (1998), I use *Artificial Neural Networks* to model the entities. Neural networks are composed of nodes (artificial neurons) which loosely model the neurons in a biological brain. Each node processes an output signal by computing a non-linear function of the sum of inputs. By organising these nodes into layers, signals can travel through from the input layer to the output layer. This is a *feed-forward* neural network as the signal flows forward through the layers (can't loop back). A *fully-connected* neural network feeds the output of every node in a layer into the input of every node in the next layer.

Each node calculates a weighted average of the inputs offset by a bias term. An *activation function* can then be applied to produce the output of the node. As Cangelosi & Parisi *do not* describe which activation function was used in their paper[1], I explore the use of three common functions:

- Identity: $f(x) = x$
- Sigmoid: $f(x) = (1 + e^{-x})^{-1}$
- ReLU: $f(x) = \begin{cases} 0 \text{ for } x \leq 0 \\ x \text{ for } x > 0 \end{cases}$

The entities can be represented by a fully-connected feed-forward neural network. The *perception* of the entity acts as the input of the neural network with the output treated as the *action* chosen by the entity. To increase the computational abilities of the entity, I also include a layer of hidden nodes to allow for more complex decision making (De Villiers and Barnard, 1993). I initially use five hidden nodes, as in Cangelosi and Parisi (1998). By altering the weights and biases used by the network, the entity responds differently to the perceptual inputs and produces different behaviours. Note that when using the identity function, hidden layers are redundant as multiple layers can be linearly combined through matrix multiplication.

2.2.2 Genetic Algorithm

The *genetic algorithm* is a natural implementation for simulating the population dynamics of a group of such entities (Holland et al., 1992). Just as artificial neural networks are inspired by animal brains, the genetic algorithm is inspired by the process of natural selection, the very process that we want to simulate.

Generally, a genetic algorithm takes a *population* of candidate solutions to an optimisation problem and evolves the population towards a better solution. Each candidate has a set of properties; the *genetic representation* which can be mutated and altered. The evolution starts from a population of randomly generated individuals and proceeds iteratively. For each *generation*, the *fitness* of each individual is calculated. The fitter individuals are selected and their genomes modified to form a new generation, used in the next iteration.

This maps neatly to our problem. The *population* used by the algorithm is simply a group of entities. The *genetic representation* of the entities is the set of weights and biases used by each neural network. The *fitness* score, F is calculated according to the number of edible and poisonous mushrooms eaten by the entity during the simulation, E and P respectively:

$$\mathrm{F} = 10E - 11P \tag{2.1}$$

My interpretation of this equation is that it rewards entities that eat edible mushrooms and punishes those that eat poisonous mushrooms. The difference in weight between these two scores gives a greater reward to the entities that eat more edible mushrooms than poisonous mushrooms as it punishes the greedy strategy of simply eating as many mushrooms as possible.

[1] I contacted the authors concerning this but no further details were forthcoming.

Constant	Description	Default Value
dim_x, dim_y	The width and height of the simulation environment	20×20
num_mushroom	The number of mushrooms placed in the environment	20
num_epochs	Number of epochs for each simulation	15
num_cycles	Number of simulation cycles per epoch	50
num_entities	Number of entities in a population	100
num_generations	Number of times a population will reproduce	2000
mutate_percentage	The percentage of weights to mutate in reproduction	10
percentage_keep	The percentage of entities chosen to reproduce	20

Table 2.1: A description of the constants and default values used for the simulation.

Finally, reproduction occurs by selecting a percentage of the population with the highest fitness score and producing a small set of offspring for each of these individuals. In Cangelosi and Parisi (1998), the top 20% of entities are selected from a population of 100 at the end of the generation. Each of these entities produces five entities by randomly mutating 10% of their weights and biases, producing a new generation of 100 entities. This mutation consists of adding a sample of the uniform distribution from -1 to 1. The experimental constants are listed in Table 2.1.

2.2.3 Population Types

To carry out the investigation of whether introducing language to a population increases its fitness, I implement three different populations in the simulation. As described by Cangelosi and Parisi (1998), these three populations differ in how the communication signals are used in the simulation environment. The three population types are:

- No Language.
- External Language.
- Evolved Language.

The population with **no language** acts as a baseline. The linguistic input to the neural network controlling the entity is set to a constant and the linguistic output is ignored. These entities cannot perceive the properties of the mushrooms unless they are adjacent, unlike the other populations they are not assisted by a linguistic signal.

Entities in the **external language** population are not given a constant linguistic input signal. Instead, at every step the entity is provided with a linguistic signal corresponding with whether the edible mushroom is edible or poisonous, as if another entity adjacent to that mushroom can see that mushroom's properties and communicates them through this signal. The language is “externally provided" because exactly one signal is used for edible mushrooms and another is used for poisonous mushrooms without actually involving another entity in the simulation.

The population with an **evolved language** is similar to the **external language** population, but without enforced signals. Instead, the population derives its own signals. Each entity is paired with a randomly chosen entity from the population at each simulation cycle (a ‘speaking’ entity). This second entity labels the mushrooms for the listening entity. Both entities are provided with the same *direction* input but the speaking entity additionally receives the properties of the closest mushroom, no matter the distance. The linguistic output of this second entity is used as the linguistic input to the primary entity; simulating a one-word utterance.

2.3 Simulation Analysis

To analyse their simulation, Cangelosi and Parisi (1998) first compared the three populations with respect to the average fitness achieved across 1000 generations. As discussed in Section 2.2.2, the fitness score for each entity describes its success in distinguishing between edible and poisonous mushrooms, correctly eating the former and avoiding the later.

Cangelosi and Parisi (1998) also described the *efficiency* of the language produced by these entities. They gave three requirements for a population having an efficient language, based on principles that Clark (1995) argues govern a child's acquisition of a lexicon:

1. Functionally distinct categories (e.g. mushroom type) are labelled with distinct signals.
2. A single signal tends to be used to label all instances within a category.
3. All the individuals in the population tend to use the same signal to label the same category.

Note that the **external language** satisfies all three requirements and is thus an upper bound for language efficiency.

To investigate the language produced by different populations, Cangelosi & Parisi used a *naming task*. In this controlled experiment, each entity in a population is exposed to the entire set of mushrooms (10 edible and 10 poisonous) in four locations (forward, left, backwards and right). When summed over the whole population, the output signals associated with these 80 conditions describe a frequency distribution over the language's possible signals, allowing the evolution of the population's language to be visualised. Cangelosi and Parisi use 3 bits for their signal language; giving 8 possible signals for the entities to choose from when communicating.

To quantify the efficiency of a population's language, Cangelosi and Parisi (1998) introduced a Quality Index (QI) score, calculated as follows:

$$d_{\text{poisonous}} = \sum_{i=1}^{8} |x_i - x_e| \tag{2.2}$$

$$d_{\text{edible}} = \sum_{i=1}^{8} |y_i - y_e| \tag{2.3}$$

$$\text{QI} = \sum_{i=1}^{8} |x_i - y_i| - k \times \min(d_{\text{poisonous}}, d_{\text{edible}}) \tag{2.4}$$

x_i is the percentage of signal i for poisonous mushrooms and y_i is the percentage of signal i for edible mushrooms, as calculated from the naming task. x_e and y_e are the expected percentages in the case of a flat distribution, in this case $\frac{1}{8}$. k is a constant to weight the effect of the internal dispersion values $d_{\text{poisonous}}$ and d_{edible} and is typically 1.

These dispersion values measure the variance of the distribution of signals used for the same category (edible or poisonous). This captures the use of synonyms, as these values are highest (equal to 1.75) when only one signal is used for the category and are lowest (equal to 0) when all eight signals are used with equal frequency.

The first part of the QI equation captures the principle of contrast (use of one word for each class of mushrooms) as it is highest when different signals are used for each category. By combining this with the smaller of the two dispersion values, we get a score that captures the idea of an efficient language.

In Section 4.1, I conduct this same analysis to compare the generational fitness, language distribution and correlation between language efficiency and population fitness to the results achieved by Cangelosi and Parisi (1998).

2.4 Requirements Analysis

My project involves re-implementing the simulation described in Sections 2.1 - 2.2 and analysing this simulation with regards to the metrics described in Section 2.3. The requirements for this project can then be divided into two parts: implementing the simulation; and constructing the means of comparing my implementation against the findings in Cangelosi and Parisi (1998).

Simulation Construction

1. Implement the simulation environment with a world grid populated by poisonous and edible mushrooms as described in Section 2.1. The simulation loop should be divided into regular ‘epochs’.
2. Have entities capable of navigating the environment, taking actions in each simulation cycle controlled by feed-forward neural networks; with the structure described in Section 2.2.1.
3. Introduce a genetic algorithm that executes after all entities complete the simulation, as described in Section 2.2.2.
4. Develop three populations, one without language, one with an externally imposed language and one with an evolved language involving speaker-listening pairs, as described in Section 2.2.3.

*5. (Extension) Develop optimisations to reduce the runtime of the simulation.

Simulation Analysis

1. Plot the average fitness over the number of generations to compare between the three populations.
2. Plot the frequency distribution of the different signals produced by entities with the evolved language using a naming task.

3. Calculate the Quality Index (QI) of the language produced by the population without language and the population with an evolved language to investigate the genetic advantage of producing productive signals.

4. Investigate the correlation between QI of the language and the fitness of the species to determine if change in the language or in the categorisation skill of the entities affects the linguistic ability.

*5. (Extension) Investigate the behaviour of entities at specific generations and whether convergence of behavioural patterns corresponds to generational fitness.

*6. (Extension) Explore the robustness of Cangelosi and Parisi's simulation by altering their arbitrary design choices.

2.5 Starting Point

The implementation of the project is based on Cangelosi and Parisi (1998) which I familiarised myself with before beginning the project and in the early preparatory phases.

This project builds on concepts of simulations; I had a small amount of experience in programming simulations from an A-Level project in 2016. Before the project, I also read Simulating the Evolution of Language (Cangelosi and Parisi, 2002) to give me an overview of the techniques used in this field.

This project builds on some of the content covered in the Artificial Intelligence course and the Formal Models of Language course from Part 1B. In particular, I had no experience with neural networks before beginning this project and all the code for the project was written from scratch within the official timeline.

2.6 Software Engineering

2.6.1 Languages and Libraries

I chose Python for my project due to the ease of programming, my experience with it and the availability of good libraries for numerical analysis and plotting. To avoid code duplication, I made use of a few libraries:

1. To perform some scientific computing, I used `SciPy`[2].

2. For the plotting of the analysis of the simulation, I used `Matplotlib`[3].

3. For producing unit tests, I used `pytest`[4].

[2]https://www.scipy.org/
[3]https://matplotlib.org/
[4]https://docs.pytest.org/

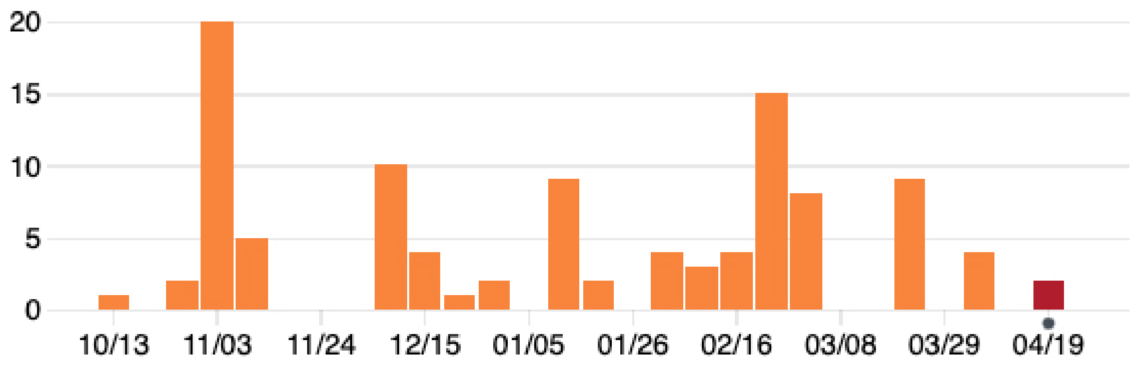


Figure 2.2: Commits to the Github repository over the duration of the project.

2.6.2 Project Management

During implementation, I followed an agile development process. An initial project plan was formulated at the beginning of the project with a list of tasks to complete. These tasks were compiled into a Kanban board, with sections for To-do, In Progress, Testing/Documenting and Completed. My project plan divided the timeline of my project into a series of 2-3 week sprints, each with associated deadlines and milestones. At the beginning of each sprint I selected the tasks to complete in order to meet these deadlines and successfully pass each milestone, often completing additional tasks when work was finished early.

This agile process allowed me to ensure I was on track with my project. In fact, the success criteria proposed at the start of the project were met very early on in the timeline, allowing me to focus on refining the project and complete my proposed extensions.

2.6.3 Version Control

For version control, I used Git to track change with a remote repository on GitHub[5]. In total, 104 commits were made over the course of the project as seen in Figure 2.2. My source code is publicly available on GitHub with a README to explain how to run it. The project is licensed under the MIT license, for free modification and reuse while limiting my liability.

The repository served as a backup, allowed me to revert to previous iterations of my code and allowed me to work on multiple computers. In particular, it allowed me to deploy my project to the University's High Performance Computing (HPC) service for running the simulations.

2.6.4 Development Tools

I made use of a number of tools to streamline my development process:

1. I used Travis[6] for continuous integration. With each commit, a script would automatically run all my unit tests and check my code for correct formatting.

[5] https://github.com/
[6] https://travis-ci.org/

2. I used pylint[7] to lint my code against Google's Python style guide[8].
3. I used yapf[9] to format my code consistently. I implemented a git hook to automatically format my code with each commit; the formatting was further checked by Travis.

2.7 Summary

The evolution of language is a challenging problem that can be explored using simulations. In this chapter, I discussed the design of the "mushroom world" environment populated by entities controlled by neural networks with a genetic algorithm to model evolution. Three populations can be considered; one without language, one with an external language and one with an evolved language. By comparing these populations in terms of fitness and language production, this simulation can be analysed to determine the parallel ability of the entities to categorise mushrooms and their ability to name them. I concluded the chapter by discussing the professional approach I took to develop, maintain and test my code.

[7] https://www.pylint.org/
[8] http://google.github.io/styleguide/pyguide.html
[9] https://github.com/google/yapf

Chapter 3

Implementation

Section 3.1 of this chapter introduces the modular structure of my implementation. In Sections 3.2 to 3.4, I detail the Environment, Entity and Simulation modules. I discuss my analysis tools in Section 3.5.

3.1 High-level Overview

The implementation of my project is split into three modules as seen in Figure 3.1. The Simulating module contains all the code relevant to creating the simulation. This covers the "Implementing the Simulation" requirements for the project described in Section 2.4. The Analysis module contains functions for plotting different graphs and carrying out the "Analysis of the Simulation" requirements for the project.

Within the Simulating module, the classes correspond to the core concepts detailed in Chapter 2. A full UML diagram of this module (excluding tests) can be seen in Figure 3.2. The `Entity` class and sub-classes correspond to the entities described in Section 2.2. The `Environment` class corresponds to the "mushroom world" simulation environment described in Section 2.1. The `Simulation` class corresponds to the actual simulation, including the genetic algorithm (§2.2.2) and the different populations (§2.2.3).

The Analysis module contains the code to analyse the simulation. This includes the plotting of generational fitness, language frequency and quality index as described in Section 2.3. It also contains another file to produces heatmap plots from the neural networks of a population.

3.2 Environment

The Environment module contains the `Direction` enumeration and the `Environment` class. These are used to represent the state of the mushroom world including the position of the mushrooms and the entity and the direction that the entity is facing. It contains methods for populating the world with mushrooms, managing the state of the world and querying the world, totalling 385 lines of code.

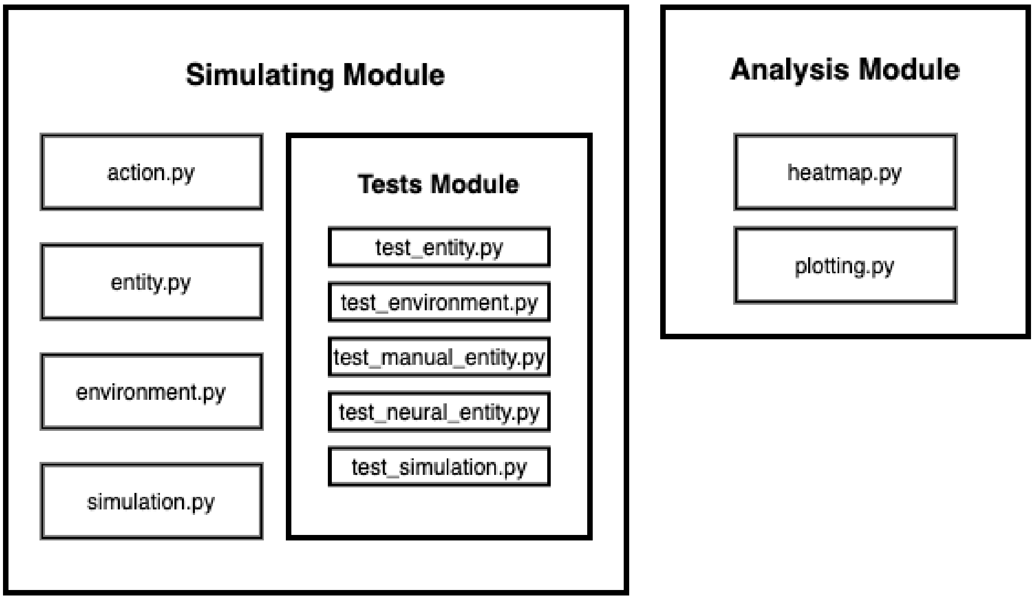


Figure 3.1: Core modules for my project

3.2.1 Mushrooms

Mushrooms are represented as 10-bit integers. Conceptually, each bit corresponds to a binary property of the mushroom. Poisonous mushrooms are represented as the bit-string `0b1111100000` with one randomly-chosen bit flipped. Similarly, edible mushrooms are represented by the bit-string `0b0000011111`, again with one bit flipped. This means that all mushrooms within a class share the same prototype but have some variations to simulate visual inconsistencies.

3.2.2 Representation of the World

The 20x20 mushroom world is stored as a dictionary, mapping from (x,y) coordinates to mushroom values. Since there are at most 20 mushrooms at any point in the simulation, this dictionary holds a maximum of 20 values at once.

This data structure is chosen over an array representation as the world is sparse. Since the `closest_mushroom()` function is called every simulation frame and involves iterating over every mushroom, we want this to be efficient. To compare the two data structures, I ran a benchmark. This benchmark created worlds of both types, populated each with 20 mushrooms and timed how long it took to iterate through all 20 mushrooms. By taking an average of 10000 worlds, I found that the dictionary representation was 14.1 times faster, with an average time of 3.3µs (with a standard deviation of 0.7µs) compared to 46.3µs (with a standard deviation of 9.1µs) for the array representation. The benchmark script can be found in Appendix A.

Two run-time exceptions are implemented, `WorldFull` and `MushroomNotFound`, thrown when the world is full and empty respectively. Positions in the world are passed between methods as integer pairs (x, y), represented as the type `pos` in the UML diagram.

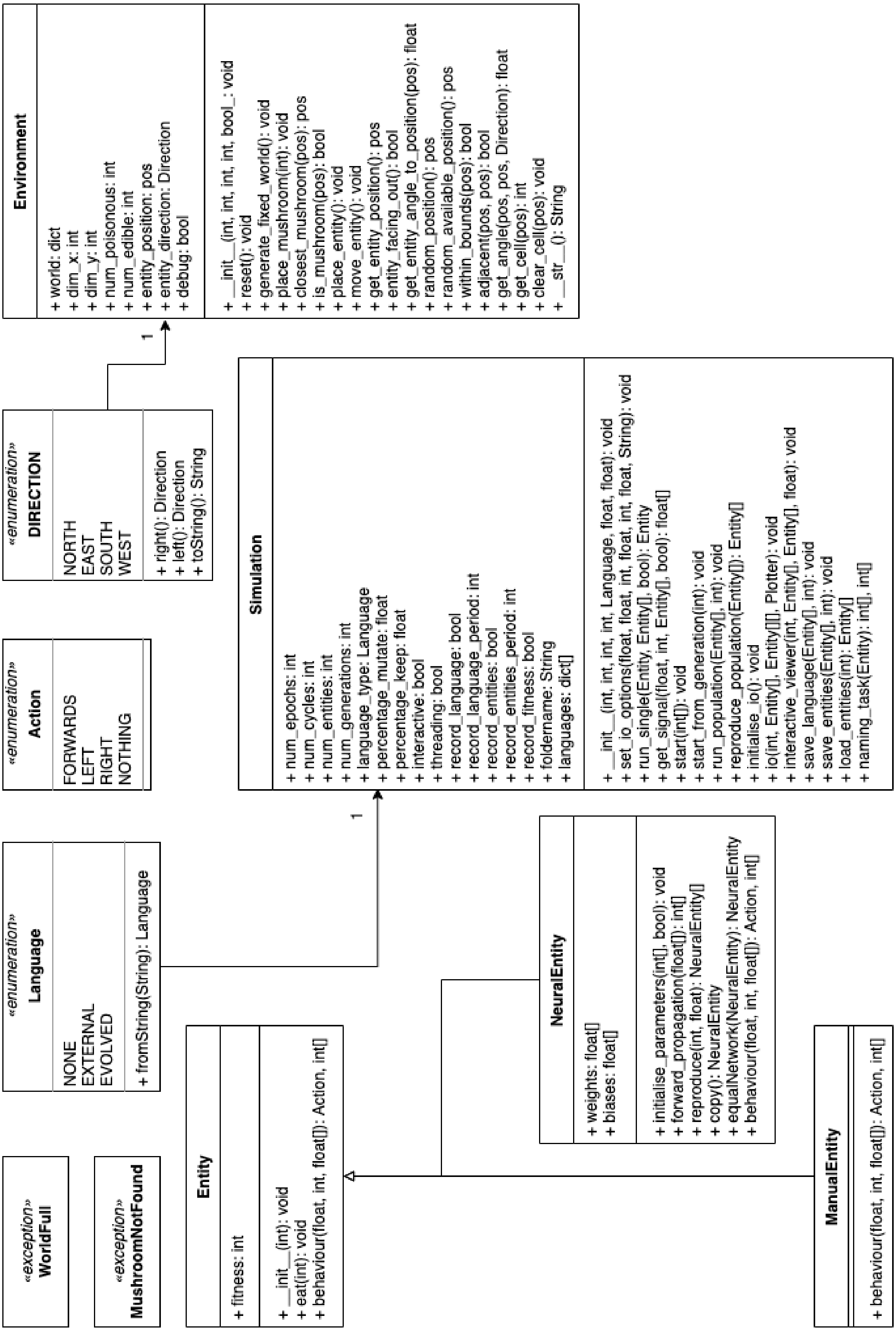


Figure 3.2: UML Diagram for the Simulating Module

3.2.3 Environment Functionality

The methods in the Environment module can be split into four groups; world initialisation, utilities, mushroom manipulation and entity manipulation.

World Initialisation: An `Environment` object is created by specifying the dimensions of the world and the number of edible and poisonous mushrooms expected. These are three of the 'arbitrary' parameters described in Table 2.1; by default the world is 20x20 and 10 of each mushroom type is placed randomly at the start of each epoch. The `reset()` method is called to place the mushrooms initially and at the start of each epoch.

Utility Methods: Utilities were implemented to calculate adjacency information, find angles between positions, query individual cells and find available cells.

Mushroom Manipulation: Along with methods to place and test for mushrooms, the `closest_mushroom()` method loops through the position keys in the dictionary and for each of these calculate the Manhattan distance. The Manhattan distance is used because it is faster to calculate, it is always an overestimate (or equal to) the Euclidian distance and because the entity can only move rectilinearly.

Entity Manipulation: The world keeps track of the position and direction of the entity within it. This introduces a layer of abstraction between the `Entity` class and the `Environment` class; the `Entity` class is only concerned with the behaviour of the entity independently of the actual representation of its position and movement within the virtual world. The `Simulation` class acts as an interface between instances of these objects, discussed in Section 3.4. Relevant methods in the `Environment` class are used to move and query the position and direction of the entity, where the direction is stored using the Direction enumeration. In particular, the `entity_facing_out()` method is used for an optimisation discussed in Section 3.4.5.

3.2.4 Testing

I produced a total of 68 unit tests using `pytest` to assert the behaviour of the Environment module. This was done early to ensure that any subsequent refactoring of the code would retain the correct behaviour of the program. Upon discovering bugs during the development process, I produced regression tests to prevent the reemergence of bugs later in development.

Unit tests were also produced for the Entities and Simulation module, totalling 914 lines of code.

3.3 Entities

The Entities module, totalling 342 lines of code, contains the `Entity` class and its two subclasses; `ManualEntity` and `NeuralEntity`. Separately, the Action module contains the Action enumeration to describe the possible actions taken by the entity; `FORWARDS` to move once cell forwards, `LEFT` to turn 90° left, `RIGHT` to turn 90° right and `NONE` to do nothing. The full UML diagram can be seen in Figure 3.2.

Signal	Datatype	Description	Input / Output
location	`float`	Angle to the nearest mushroom	Input
perception	`int`	Properties of the adjacent mushroom	Input
listening	`float[]`	A 3-bit linguistic signal (heard)	Input
vocal	`int[]`	A 3-bit linguistic signal (spoken)	Output
action	`Action`	The action taken by the Entity	Output

Table 3.1: Inputs and Outputs of the `behaviour()` method

3.3.1 Abstract Specification

The `Entity` class acts as an empty parent class. As state, it has an `fitness` value which is used to determine the fitness of the entity according to equation 2.1. The `behaviour()` method describes the expected input-output behaviour of entities discussed in Section 2.2. When passed an angle, a mushroom and an input signal, this method returns an action and an output signal. The angle is passed as a float ranging from 0 to 1. The mushroom's properties are passed as a single integer, as described in Section 3.2.1. The input and output communication signals are represented by three bits. The outputted action is an instance of the `Action` enumeration described above. These inputs and outputs are detailed in Table 3.1. In the `Entities` class, this method always returns `Action.NOTHING` and the vocal signal `[0,0,0]`.

Before implementing the feed-forward neural networks to control the entities, I created a rule-based entity in the `ManualEntity` class which extends `Entity`. It overrides the `behaviour()` method and implements a simple algorithm to always rotate and move towards the nearest mushroom. On average, this strategy will produce a negative fitness score due to the poisonous mushroom penalty being higher than the edible mushroom reward. This behaviour is not analysed but was useful for the purpose of testing the simulation in the early stages of development due to the deterministic choices made.

3.3.2 Neural Network Entity

In Section 2.2.1 I gave the general structure for the fully-connected neural network I wanted to use to control the behaviour of the entities. Now that I have defined the datatypes for these inputs and outputs, I can produce a more concrete description of this neural network.

The neural network has fourteen input units. The first unit is the float value `location` which describes the angle to the closest mushroom. The next ten units describe the bit-string representation of the mushroom. Note that these values are 0 if the entity is not adjacent to a mushroom (this is controlled by the simulation). The final three units are used for the linguistic input signal. Five hidden units are used, as in Cangelosi and Parisi (1998). As discussed in Chapter 2, I implemented the *identity*, *sigmoid* and *ReLU* activation functions.

There are five output units; two of which encode the action taken by the entity (as described above) and the remaining three encode the outputted communication signal (one of eight). The full neural network structure can be seen in Figure 3.3.

The `NeuralEntity` class introduces a few additional methods and state. In particular, two dictionaries store the weights and biases associated with each node in the neural network. Creating a new `NeuralEntity` calls the `initialise_parameters()` method to set the weights and biases to ran-

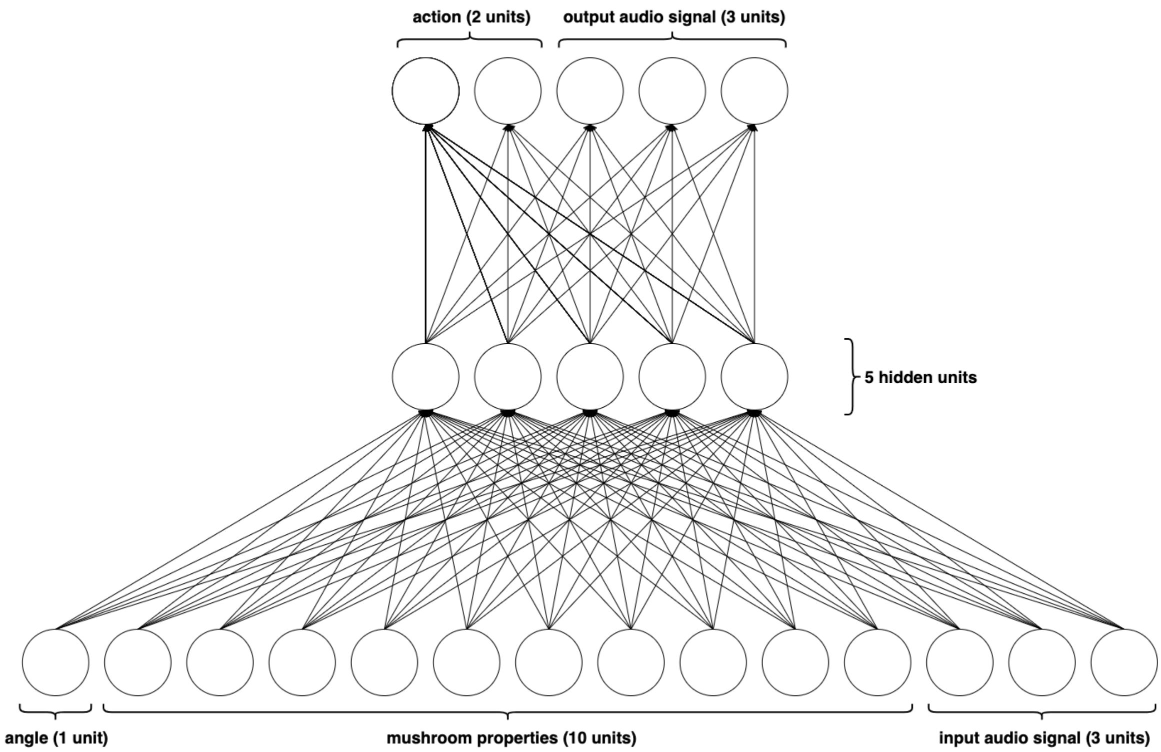


Figure 3.3: Fully-connected ANN to control entity behaviour

dom values chosen from the rectangular distribution over [-1, 1], as specified by Cangelosi and Parisi (1998).

The `forward_propagation()` method carries out the forward propagation algorithm. The following calculation is made at each layer, where W_n is the weight matrix, B_n is the bias matrix, f is the activation function and Z_n is the resulting activation for layer n:

$$Z_n = f(W_n Z_{n-1} + B_n)$$

Cangelosi and Parisi (1998) states that "for all output units, continuous values are thresholded to either 0 or 1" but is not clear how this thresholding takes place. I have decided to perform a sigmoid activation on the final layer and round the result to return integer values of 0 or 1.

`NeuralEntity` overrides `behaviour()` in order to apply the neural network to the inputs. Given the inputs, it constructs a 14-element float vector as specified in Figure 3.3, in particular converting the integer mushroom to an array of bits. It then feeds this input vector to the `forward_propagation()` method to get the output vector. The first two bits of this output vector are parsed as an instance of `Action` and the last three bits are returned as the output signal.

The weights and biases also constitute the *genetic representation* of the entity and are used to produce children between each generation in the genetic algorithm, as described in Section 2.2.2. Two parameters specify the number of children, n, to produce and the percentage of weights, p, in the neural network to mutate. When called, the entity produces n deep copies of itself and for each child, adds a sample from the uniform distribution [-1, 1] to p% of the weights and biases. A deep copy was required to prevent aliasing of the dictionaries.

3.4 Simulation

The Simulation module, seen in Figure 3.2, contains the `Simulation` class and the `Language` enumeration for implementing the simulation. The `Simulation` class is responsible for implementing the per-entity simulation (750 cycles in the Environment described above) and the population behaviour, including the genetic algorithm. The `Language` enumeration describes the differences in language between the populations as described in Section 2.2.3; `NONE` is for no language, `EXTERNAL` for the external language and `EVOLVED` for the evolved language. Together, these total 685 lines of code.

3.4.1 Initialising the Simulation

An instance of a `Simulation` object stores the parameters of the simulation and provides methods that run single-entity simulations and the genetic algorithm. The initialisation method sets the number of epochs, cycles per epoch, population size and the number of generations to run the simulation for as outlined in Section 2.1.

Further specified are two parameters used by the genetic algorithm; the percentage of the population chosen to reproduce (20% by default) and the percentage of weights to mutate (10% by default). These were discussed in Section 2.2.2 and are listed with the other simulation parameters in Table 2.1.

3.4.2 Single-Entity Simulation Run

The `run_single()` method performs the main simulation for each entity. The pseudocode for this function is shown below. Taking a single entity as a parameter, it performs the relevant operations required to allow the entity to 'live' in an instance of the `Environment` class for `num_epochs` of `num_cycles` time steps each. During this time, the entity's `behaviour()` method is passed the appropriate inputs according to its current position in the environment (lines 14 to 19). The outputs of the `behaviour()` method are interpreted accordingly, moving the entity in the environment (line 20). When the entity shares a cell with a mushroom, it eats it, changing its fitness score accordingly and the mushroom is removed from the environment (lines 23 to 25).

```
1  function run_single(entity):
2
3          # Create a random environment
4          env := new Environment
5
6          for epoch in num_epochs:
7
8              # Reset mushroom positions and entity position between each epoch
9              env.reset()
10             env.place_entity()
11
12             for cycle in num_cycles:
13                     # Gather inputs for the entity
14                     location := angle from entity to closest mushroom
15                     perception := closest mushroom if adjacent else 0
```

```
16          listening := input signal according to language type
17
18          # Get behaviour of the entity and move accordingly
19          action := entity.behaviour(location, perception, listening)
20          env.move_entity(action)
21
22          # Entity eats a mushroom if on top of one
23          if env.entity_position is a mushroom:
24              entity.eat(mushroom)
25              env.remove(mushroom)
```

Line 16 is implemented as three cases according to the `language_type` property of the simulation. For no language, the signal is just set to `[0.5, 0.5, 0.5]`. For the external language, this signal is set to `[1, 0, 0]` if the closest mushroom is edible and `[0, 1, 0]` if it is poisonous.

For the evolved language, this step is slightly more complicated. The `run_single()` method is also passed an array of the 99 other entities in the population and at each simulation step, one of these is randomly chosen to be the ‘partner entity’ to name the closest mushroom for the primary entity. This other entity is given the same `location` parameter and a constant value of `[0.5, 0.5, 0.5]` as its `listening` signal but always receives the properties of the mushroom as its `perception` input, regardless of distance to the mushroom. The `action` output of the call to this partner entity’s `behaviour()` call is ignored but the outputted signal is used as the inputted `listening` signal for our primary entity.

3.4.3 Genetic Algorithm

The `run_population()` method implements the genetic algorithm used for this simulation. Given an initial population of entities, it runs the algorithm for `num_generations` generations. The algorithm has three steps; performing the `run_single()` method for each entity of the population, sorting the population to find the entities with the highest fitness and finally choosing a certain percentage of the fittest entities to reproduce to create the next population. The pseudocode for this process is shown below.

Lines 7-9 perform this first step by running the simulation for each entity in the current population. This simulation will update the fitness of each entity each time that an entity eats a mushroom. If the language being used is the evolved language, the `run_single()` method is also passed a list of the other entities in the population to perform the appropriate naming as discussed in Section 3.4.2.

Lines 12-13 perform the second step by simply sorting the list of entities according to the fitness achieved in each simulation. The top percentage of these entities is then selected for reproduction.

Lines 16-19 create the next population by calling the `reproduce()` method on each of the fittest parent entities identified. This method is passed the number of children to produce (the reciprocal of the percentage of parents selected) and the percentage of weights to adjust in the mutation process.

Two key constants in this algorithm are `mutate_percentage` and `percentage_keep` which set the percentage of weights to mutate and the percentage of the population chosen for reproduction respectively. The setting of these constants was discussed in Section 3.4.1.

```
function run_population(entities):

        for generation in num_generations:

            # Step 1: Run the simulation for each entity
            # Pass the remaining population for the evolved language
            for entity in entities:
                    population := entities - {entity}
                    run_single(entity, population)

            # Step 2: Sort the entities by their fitness and select the best
            sort(entities, entity.fitness)
            best_entities := top percentage_keep of entities

            # Step 3: Create a new population from the fittest entities
            children := []
            for entity in best_entities:
                    children += entity.reproduce(1/percentage_keep, mutate_percentage)
            population := children
```

3.4.4 Running the Simulations Using HPC

To run experiments, I acquired access to the University of Cambridge's High Performance Computing (HPC) facility. This gave me access to an environment where I could set up experiments and schedule runs of my program using the SLURM workload manager[1]. I created a series of bash scripts that would allow me to schedule a batch of jobs at once (for example 10 independent runs of the genetic algorithm for each language type). I used the `rsync` command to copy computed data to my local file system. This work cycle allowed me to work quickly and effectively, all while avoiding having to use my own laptop for computationally demanding jobs.

To further speed up running experiments, I used the `argparse` library to create a command-line interface for my system. This allowed me to run the genetic algorithm, a single-entity simulation or load a previously run simulation from the command line and set any of the simulation and I/O parameters. Examples of running my system from the command line can be seen in Appendix B.

3.4.5 Optimisations

Having finished the core implementation of the project early, I implemented optimisations to increase the rate at which I could run experiments. Throughout implementation I made decisions that would favour a faster runtime, such as using `numpy` arrays for matrix operations or using the dictionary world representation, as discussed in Section 3.2.2.

Considering a run of the genetic algorithm informs us that the `run_single()` method is called 200000 times when operating on a population of 100 entities for 2000 generations. Each call to this method

[1] `https://slurm.schedmd.com/`

involves 15 epochs of 50 cycles, or 750 iterations of the perception-action loop described in Section 3.4.2. Giving us a total of 150 million total iterations, this is a good site for optimisation.

The most expensive operation in this loop is the call to `behaviour()` which performs forward propagation, involving matrix multiplications. The other operations are less expensive, involving simple variable manipulation. Furthermore, the `behaviour()` method is called twice for the population with the evolved language (once for the speaker and once for the listener). The average runtimes for the **no language** and **evolved language** populations are 108 minutes and 180 minutes respectively, calculated by comparing the average runtime of ten runs of the genetic algorithm for 1000 generations. We can therefore estimate that calls to the the `behaviour()` method are responsible for approximately 67% of the runtime for when the language type is `NONE` and 80% when the language type is `EVOLVED`. My optimisations were thus focused around this call.

Parallelisation: The first optimisation was made by taking advantage of the fact that the 100 calls to `run_single()` are large (750 simulation cycles each) and do not interact (each entity exists in its own world). These calls are thus *embarrassingly parallel* and so are worthy of parallelisation (Mycroft, 2019).

Using the `multiprocessing` library,[2] I replaced the for loop with a `Pool` object which represents a pool of worker processes. The size of this pool is automatically set according to the number of hardware threads available. Using the `Pool`, I call the `starmap` method to dynamically offload the 100 calls to `run_single()` to these processes. On my laptop, with 8 hardware threads available, this gave a speedup of 3.96× when running the simulation for 100 generations, averaged over 10 runs. Unfortunately, I could not use this optimisation on the HPC as hyperthreading is disabled on the Skylake nodes I used to run my experiments.

Epoch skipping: By noting that (1) the `behaviour()` method is deterministic and that (2) the environment is static (besides the entity), I was able to reduce the number of simulation loops required in the `run_single()` method. If at any stage the call to `behaviour()` returns the `NOTHING` action then due to these two properties, it will continue to do so for the rest of the epoch. Thus, we can simply skip the remainder of the current epoch; the "Skip None" optimisation.

A similar optimisation can be made when the entity attempts to move `FORWARD` through the edge of the world. The response to this behaviour was not defined in Cangelosi and Parisi (1998) so I had the entity remain in the same position. This is arbitrary, and I explore alternative choices in Section 4.3. As the entity remains in the same position, it will return the same response in the next cycle so we can again skip the epoch. This is the "Skip Edge" optimisation. Similar epoch-skipping optimisations can also be considered; such as the "Detect Looping" optimisation which determines if the entity is stuck spinning on the spot in either direction by examining the last four actions.

Figure 3.4 shows how each epoch-skipping optimisation decreases the average runtime of the genetic algorithm. The "Skip None", "Skip Edge" and "Detect Looping" optimisations give speedups of 1.03×, 1.17× and 1.38× respectively. When all three of these optimisations are used, there is a total speedup of 1.78×, significantly reducing the time taken to run experiments.

It is worth noting that *by the nature of the genetic algorithm*, the population will learn to avoid the very behaviours that lend themselves to optimisations. This can be seen in the time taken to simulate each generation; in Figure 3.5 we can see that "Skip None" optimisation is very effective in early generations but gradually has less effect. The population learns to avoid producing a `NONE` action, as

[2] `https://docs.python.org/3.8/library/multiprocessing.html`

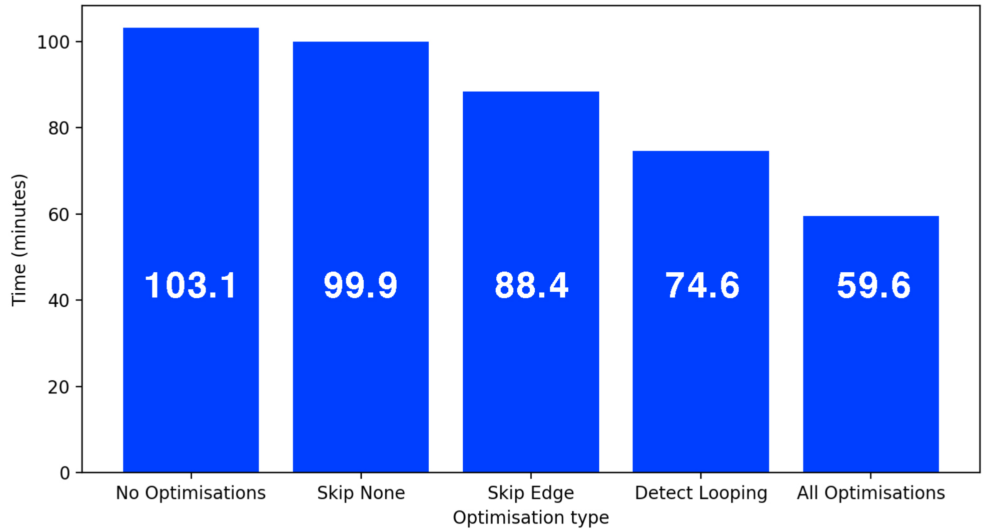


Figure 3.4: Time taken to run 1000 generations of the genetic algorithm on a population with no language for each optimisation, averaged over ten runs.

this wastes precious time that could be spent searching for mushrooms. A similar effect is seen for the “Detect Looping” optimisation but the inverse is seen for the “Skip Edge” optimisation; the entities do not know where the edge of the world is and so this behaviour occurs more frequently as the entities learn to explore.

3.4.6 Interactivity

During development, I implemented interactive behaviour to allow me to watch the simulation and genetic algorithm occur step-by-step. This was also useful for evaluation as it allowed me to analyse the behaviour of each population by replaying saved simulations from different generations. This behaviour is activated using a ‘`-i`’ flag in the command-line interface.

Enabling this behaviour produces an average fitness graph, updated continuously with the latest average fitness of the population. Running the simulation in this interactive mode also displays useful information at each generation and even allows the user to watch the behaviour of a single entity in a visual representation of the `Environment` object. Examples and descriptions of the live graph, generational information and single-simulation interactivity can be found in Appendix B.

3.4.7 Data Recording

For the analysis of the simulation, I implemented a few utility methods to collect data as it ran. The command-line interface is used to set a group of I/O parameters to control the storage of fitness, language and population data. The parameters control whether, where and how frequently this data is stored.

By default, at each generation this method records the the average fitness, the language and the neural networks of the population. The language is saved as a frequency table, calculated by calling the `naming_task()` method for each entity in the population, which calls `behaviour()` for 80 different

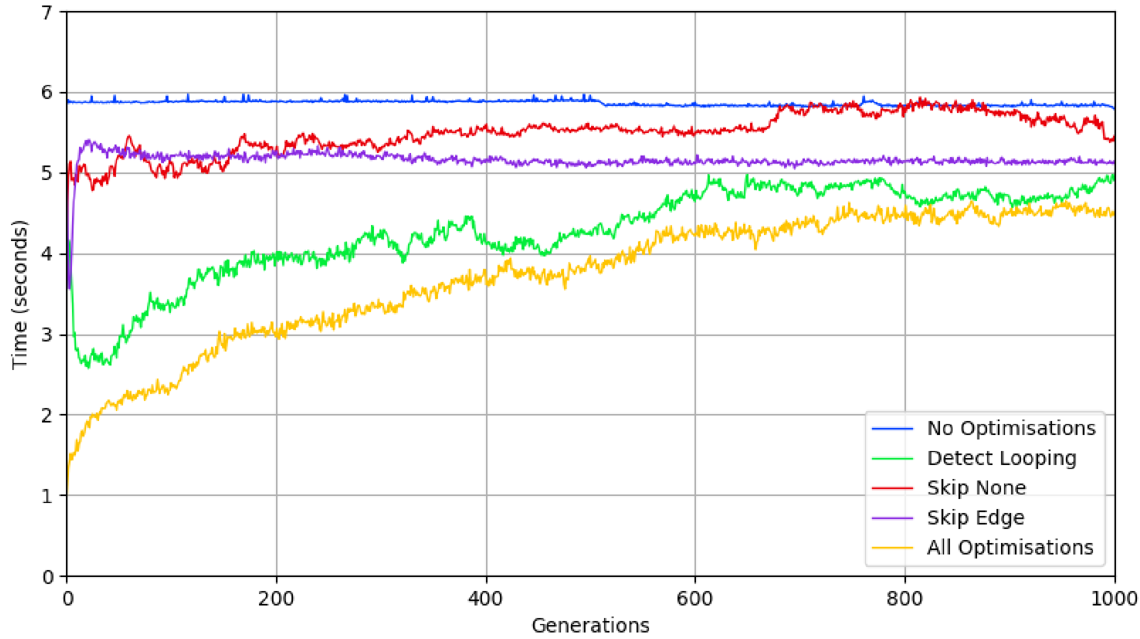


Figure 3.5: Time taken to run each generation of the genetic algorithm on a population with no language for each optimisation, averaged over ten runs.

possible inputs and returns the outputted signals. The neural networks are saved in a binary file, used to load previously-run simulations and to analyse the weights and biases of the population.

3.5 Simulation Analysis

The Analysis module seen in Figure 3.1 contains the `Plotting` class and a variety of methods to analyse the simulation in the same manner as Cangelosi and Parisi (1998). The `Plotting` class is used for displaying a live fitness graph, discussed in Section 3.4.6. A command-line interface was also implemented for this module, allowing me to easily generate different fitness graphs and language frequency distribution graphs as seen in Chapter 4.

3.5.1 QI Score

A particular set of methods were used to calculate the QI score of the language produced by the population as given in equation 2.4 in Section 2.3. Cangelosi and Parisi (1998) used this equation to correlate language efficiency with fitness and found that even in a population that did not use language, there was a high correlation between fitness and QI score over 1000 generations. Upon initial analysis of my simulation however, I only achieved low and negative correlation scores between fitness and the QI score defined by equation 2.4, as seen in Table 3.2.

Reviewing this equation further we can see that this equation does not match Cangelosi and Parisi's definition of a productive language. The first half of the equation, $\sum_{i=1}^{8} |x_i - y_i|$ has a maximum value of 2 if no signal is used for both edible and poisonous mushrooms, so is large if the language is efficient. The second half of this equation, $\min(d_{\text{poisonous}}, d_{\text{edible}})$, has a maximum value of 1.75 if only one signal is used for each mushroom type, thus is also large if the language is efficient.

Replication	**1**	**2**	**3**	**4**	**5**
QI Equation A	0.28	0.27	0.13	0.21	0.17
QI Equation B	0.54	0.69	0.41	0.57	0.52

Table 3.2: Pearson Correlation of QI Score with Average Fitness for five replications of the simulation. **QI Equation A** is equation 2.4 and **QI Equation B** is equation 3.1.

This means that if the QI equation is meant to capture language efficiency, these two values should be *added* instead of subtracted. I thus implemented the following equation:

$$\mathrm{QI} = \sum_{i=1}^{8} |x_i - y_i| + k \times \min(d_{\mathrm{poisonous}}, d_{\mathrm{edible}}) \tag{3.1}$$

As well as having logical justification, the preliminary results in Table 3.2 show that this equation produces a better correlation with average fitness than equation 2.4. From this I assume that there was a printing mistake in the published equation that Cangelosi and Parisi gave.

3.6 Summary

In this chapter I covered my implementation of the "mushroom world" simulation. I described the Environment, Entity and Simulation classes and gave details of how I implemented the genetic algorithm and the simulation loop. I discussed my testing framework, how I made the simulation interactive and how I recorded data for analysing the simulation. Through careful analysis, I was able to produce key optimisations for my simulation and also discover a printing error published in Cangelosi and Parisi (1998).

Chapter 4

Evaluation

In Section 4.1.1, I begin by examining the average fitness of different populations with default simulation parameters. In Section 4.1.2 I then examine the language produced by the evolved language population and investigate the QI of this language.

In Section 4.2, I conduct my own, deeper analysis of the simulation by examining the convergence of the neural networks to different behavioural states. Finally I explore the robustness of the simulation to the choice of simulation parameters in Section 4.3, and discuss whether the conclusions of Cangelosi and Parisi (1998) still hold.

4.1 Initial Analysis

4.1.1 Population Fitness

The first analysis that we can conduct is how the average fitness of each population compares across many iterations of the genetic algorithm.

Cangelosi and Parisi (1998) performed this experiment by running the simulation 5 times for each language type for 1000 generations. They chose to stop the simulation at 1000 generations because by this point all three populations were capable of discriminating between the two mushroom types and had further learnt the correct behaviour (avoiding the poisonous and eating the edible ones). The result of their experiment can be seen in Figure 4.2.

In my analysis I chose to run each experiment for 2000 generations and average across 10 replications for each population type. The choice of 2000 generations was made to explore potential behaviour that Cangelosi and Parisi (1998) may have missed and the choice of 10 runs was made to increase the statistical validity of my results due to high variance observed between runs. I was also able to run these experiments due to my access to the HPC facility; computing power that Cangelosi and Parisi may not have had.

The first result of this experiment can be seen in Figure 4.1. The simulation parameters used were the defaults detailed in Table 2.1, with the ReLU activation function used in the hidden layer of the neural networks and a sigmoid activation function used at the final layer. Unless otherwise specified, these parameters will be assumed for the rest of this chapter.

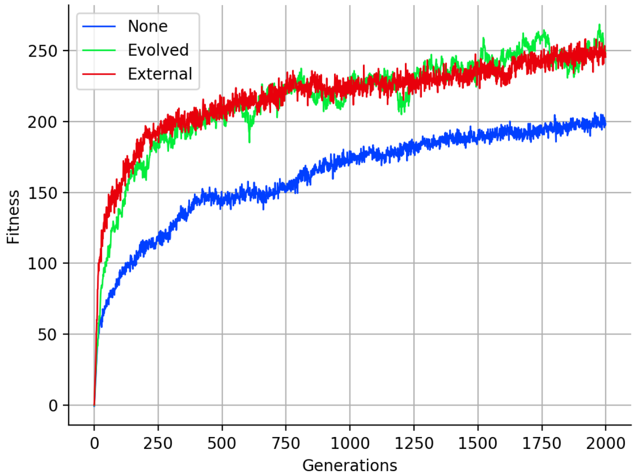


Figure 4.1: Average fitness of each population, averaged over 10 replications, with a **ReLU** hidden layer activation.

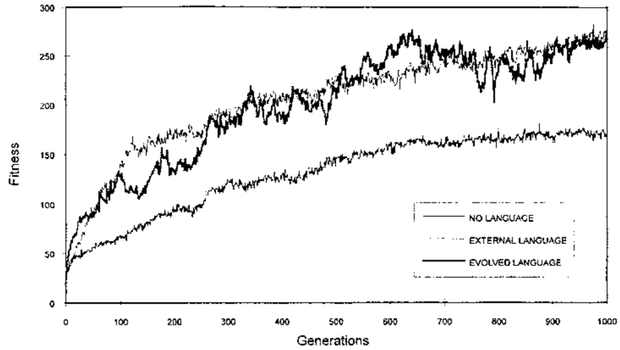


Figure 4.2: Average fitness of each population, averaged over 5 replications. Source: Cangelosi and Parisi (1998).

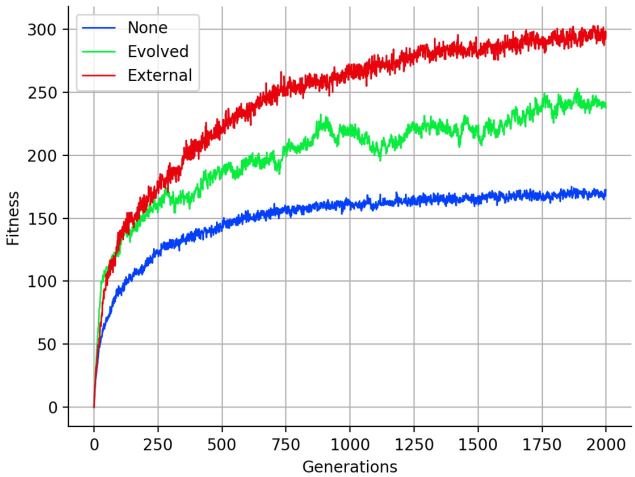


Figure 4.3: Average fitness of each population, averaged over 10 replications, with an **identity** hidden layer activation.

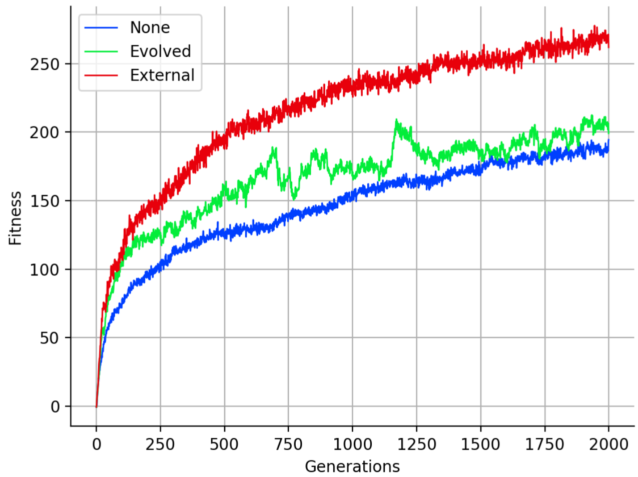


Figure 4.4: Average fitness of each population, averaged over 10 replications, with a **sigmoid** hidden layer activation.

From this experiment we can conclude that language is a useful adaptation for our entities. The population with **no language** achieves an average fitness of 200 by the end of the 2000 generations whereas the two populations with language achieve an average fitness of 250 by generation 2000. A behavioural test shows that entities in these population use the signals provided to move away from poisonous mushrooms regardless of distance and approach edible mushrooms to eat; the population without language must first approach each mushroom in order to categorise them and behave accordingly. This takes additional simulation cycles, explaining the lower fitness achieved.

As can be seen by Figure 4.2, these were the same conclusions reached by Cangelosi and Parisi (1998). However, they didn't specify which activation functions were used in their implementation. In Figure 4.3 and 4.4 we can see the results of the same experiment repeated using the identity activation function and sigmoid activation function respectively, with no other parameters changed. We see similar results as when using the ReLU activation function but with slight differences. For the case of the identity function, the **evolved language** populations perform the same as with ReLU but the **external**

language populations perform better and the **no language** populations perform worse. With the sigmoid activation function the results are very similar to with ReLU except that the **evolved language** performance is closer to the **no language** populations than the **external language** populations. Since my first experiment achieved the closest result to Cangelosi and Parisi (1998), I concluded that this was the activation function that they used, without referring to it as ReLU since the term was not coined until 2010 (Nair and Hinton, 2010).

Despite these differences, the populations with language still perform better than those without. We can therefore still conclude that language is a useful adaptation for these entities. It therefore seems that this experimental setup is robust to this structural change in the neural networks. In Section 4.3, I will further explore robustness to other changes that can be made to the simulation.

The way that the **evolved language** populations differ in performance in these two later experiments does however tell us that we cannot assume that the **evolved language** and **external language** populations are equivalent. Instead, the **external language** acts as an upper bound in terms of evolutionary fitness, as previously discussed in Section 2.3.

4.1.2 Language Analysis

In Section 2.3, I described the efficiency of the language produced by the entities according to three criteria:

1. Functionally distinct categories (e.g. mushroom type) are labelled with distinct signals.
2. A single signal tends to be used to label all instances within a category.
3. All the individuals in the population tend to use the same signal to label the same category.

I also mentioned that the **external language** is maximally efficient with respect to these criteria as only one distinct signal is used to measure each mushroom type. Given these criteria, we can now explore the efficiency of the language produced by the **evolved language** populations from my first experiment.

Figure 4.5 shows the frequency distribution of the signals produced by one of the ten populations from the experiment. The frequency distribution is calculated using the naming task described in Section 2.3. The first observation is that at generation 0, the distribution of signals for edible and poisonous mushrooms is flat. This is to be expected, as this results from random weights chosen for all entities in the population. The language at this point provides no information to the listeners in the simulation. Over time, the population converges to using a single signal to label all mushrooms in the same category (criteria 2) and all individuals of the population use the same signals (criteria 3). By generation 400, the majority of the population uses only signals `100` and `101` but still struggles to distinguish between the two mushroom types. By generation 1000, criteria 1 is satisfied with signal `100` used for poisonous mushrooms and signal `101` for edible.

In Figure 4.6 we can see how the QI score of the language changes across 2000 generations. Initially the QI score is close to 0, corresponding with the flat distribution seen for generation 0 in Figure 4.5. This is because this is the least efficient language, providing no information to listeners. The language quality quickly rises as the language converges to a more efficient state, peaking at 80%. A score of

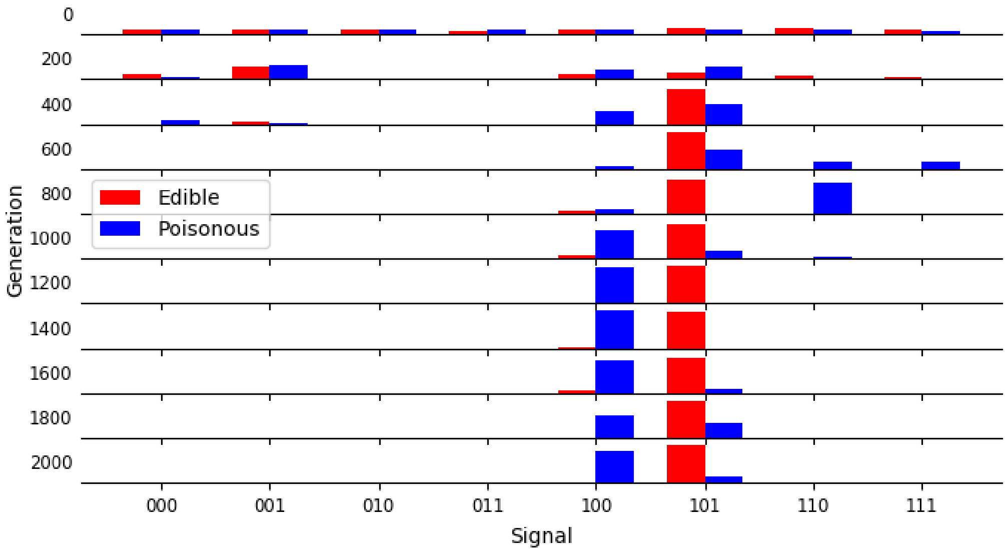


Figure 4.5: Frequency distribution of the possible signals produced by all individuals in 10 generations in one replica of a simulation with an **evolved language.**

100% would correspond to a maximally efficient language, like the **external language**. Note that the Quality Index seems to be correlated to the average fitness of the population, also seen in Figure 4.6. For this replica, the Pearson correlation between fitness and quality was 0.568. This suggests that not only is language a useful addition, but the quality of the language directly affects the resulting evolutionary success of the population.

4.2 Behavioural Analysis

In Section 4.1.1 I presented the average fitness of each population over ten replications of the simulation. By examining these ten individual replications we can gain some interesting insight into the evolutionary convergence of different behaviours.

Figure 4.7 shows these ten replications for the **no language** population for the experiment using an **identity** activation function on the hidden layer. It seems that the ten populations are converging to two distinct states; four converge to an average fitness of 250 and the other six converge to an average fitness of 125. One of the replications (seen in red) even seems to jump from one state to the other. Figure 4.8 reveals the ten replicas for the experiment using a **ReLU** activation function which seems to produce more of a continuum of states.

Examining the weights of the neural networks in each population helps provide some understanding. Figure 4.9 presents a heatmap of the weights and biases of 100 entities taken from generation 2000 of one of these populations, clamped to [-2, 2]. It is clear from the uniformity that by generation 2000 the population has converged to a single state and that by increasing the absolute values of these weights, the state has been protected from the disrupting effects of mutations, thus preventing the population from converging to a different state. Figure 4.10 shows this convergence process for the same population; weights are initially random but quickly converge to a single state.

It seems therefore that one set of populations has converged to a better state than the others. Attempting to compare the heatmaps of these different populations does not yield any clear result. Recalling from

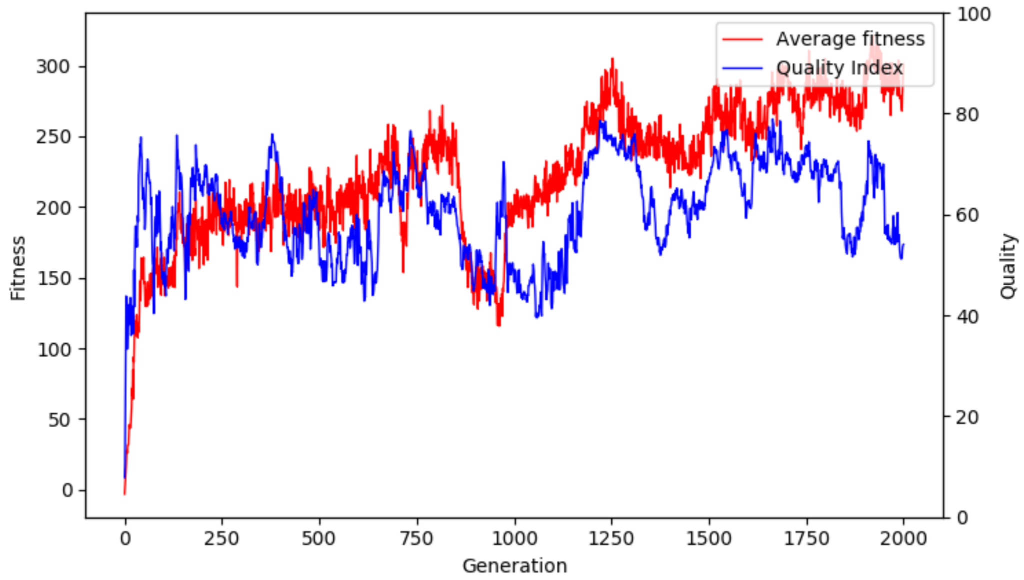


Figure 4.6: Average fitness for one replica of a population with an **evolved language**. Also shown is the Quality Index score of the language produced by this population.

Section 2.2.1 that the hidden layer is redundant when the identity is used as an activation function, we can produce an equivalent representation of these neural networks by multiplying through the weights and biases of the hidden layer:

$$W' = W_1 W_0$$
$$B' = W_1 B_0 + B_1$$

where W_1, B_1, W_0 and B_0 are the weights and biases for the output layer and hidden layer respectively and W' and B' are the weights and biases for the new, equivalent neural representation. Figure 4.11 compares these heatmaps, displaying just the weights and biases that map to the two movement outputs (ignoring the weights and biases that map to the signal outputs). The top set of networks achieved a fitness of close to 250 whereas the bottom six only achieved a fitness of close to 125 but simply examining the neural networks reveals no clear discernible difference between them.

Instead, it is productive to examine the actual behaviour exhibited by these ten populations. I examined how each population behaved when faced with a poisonous mushroom. I selected a randomly chosen entity from generation 2000 of each population and placed a poisonous mushroom two cells in front of it. As expected, since all ten of these populations have no language, all ten entities moved towards the mushroom until adjacent (they otherwise have no means of categorising the mushroom). Once the mushroom is seen, however, two very different sets of behaviours are exhibited (Figure 4.12).

The six populations that achieved a maximum fitness of 150 all exhibited "stuck" behaviours. Four of these populations spun in place (either clockwise or anti-clockwise) and the other two produced the `None` action indefinitely. Whilst valid means of avoiding the poisonous mushrooms, I call these behaviours "stuck" as it prevents the entity from discovering more mushrooms.

The other four populations, those that achieve a fitness of 250, exhibit "exploratory" behaviours. All four entities moved in the symmetric path seen in Figure 4.12, again either clockwise or anti-clockwise. This behaviour also allows the entity to avoid the mushroom but also allows it to potentially discover a different mushroom and continue searching; explaining how a higher fitness is achieved.

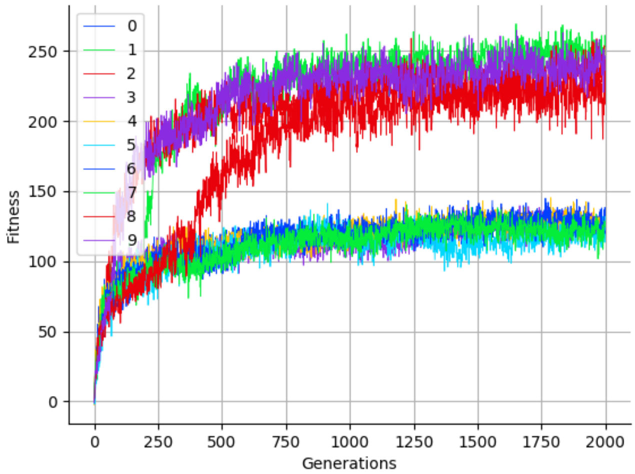


Figure 4.7: Average fitness of 10 replications of the **no language** population, with an **identity** hidden layer activation.

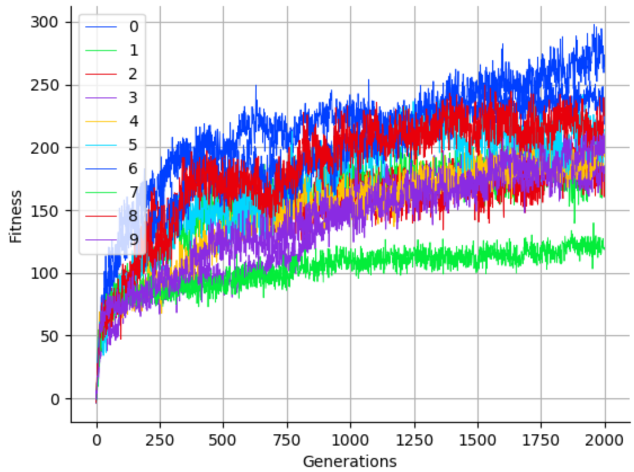


Figure 4.8: Average fitness of 10 replications of the **no language** population, with a **ReLU** hidden layer activation.

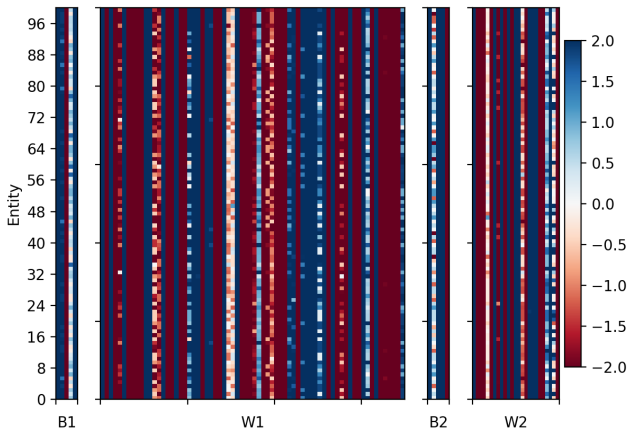


Figure 4.9: Heatmap of the weights and biases of 100 entities taken from generation 2000.

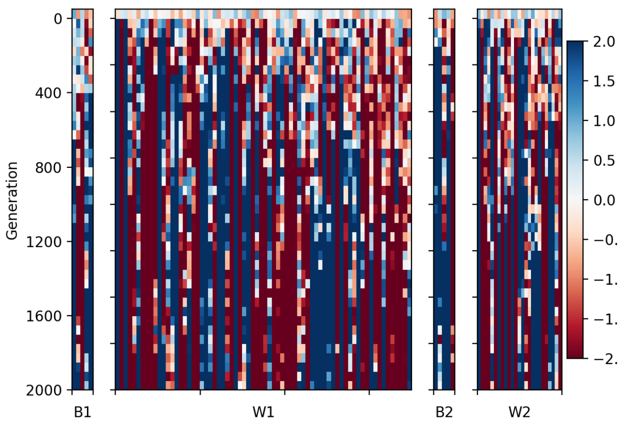


Figure 4.10: Heatmap of a randomly chosen entity taken from increasing generations.

Examining these populations at generation 200 reveals the same behaviours except for one population; replica 8 now exhibits the "stuck" behaviour. At generation 250 however, replica 8 has switched to the "exploratory" behaviour. This seems to correspond with the 'jump' to the higher state seen in Figure 4.7. From this we can conclude that early in the evolution, some populations converge to more productive behaviours than others and the gradual strengthening of weights prevents the lesser populations from escaping these states.

The same behavioural analysis can also be applied to the ten replicas seen in Figure 4.8. Although these form more of a continuum of states, similar behaviours are noticed. The green line corresponding with the 120-fitness population exhibits "stuck" behaviour whilst the blue line corresponding with the 280-fitness population exhibits "exploratory" behaviour with an even broader search path than seen in Figure 4.12. It seems that for the populations without language, it is the *search strategy* that determines genetic success.

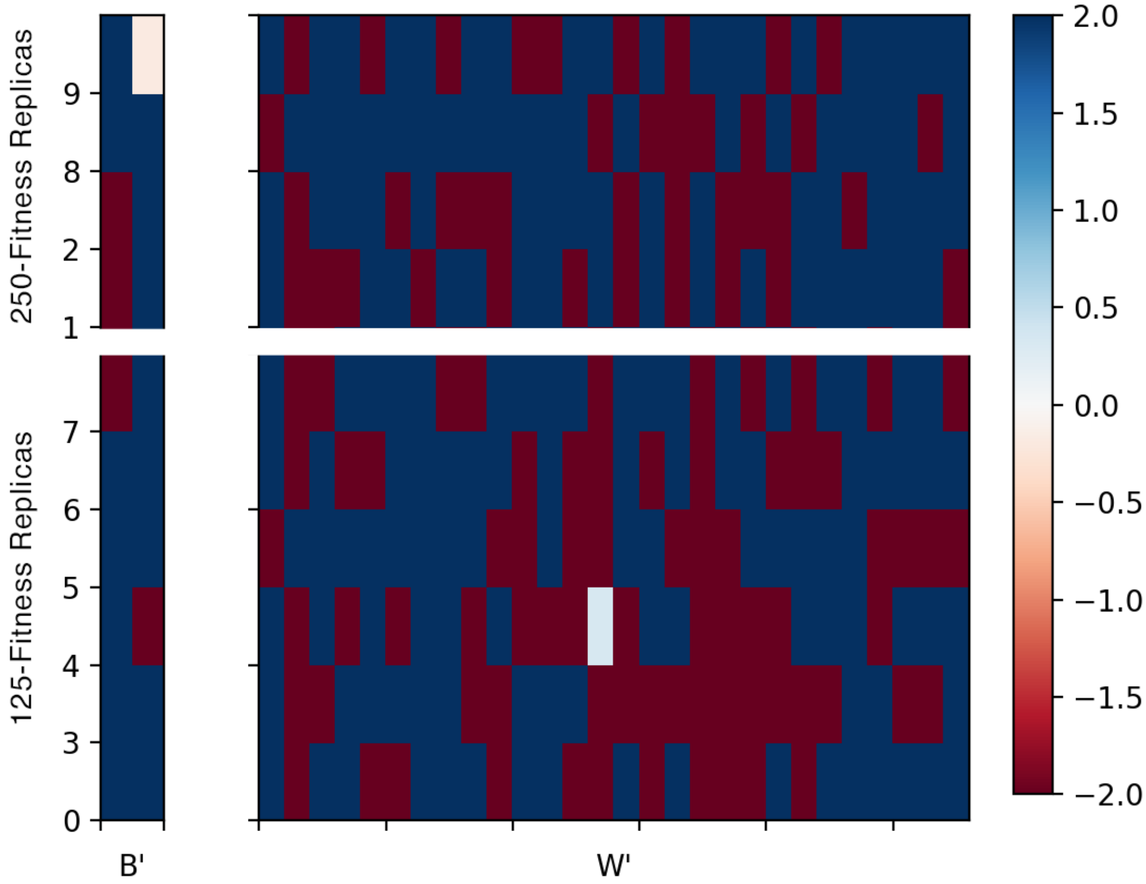


Figure 4.11: Heatmaps of the flattened neural networks of a randomly chosen entity from generation 2000 of ten replicas.

4.3 Exploring Simulation Parameters

Throughout Chapter 2 and 3, I discussed how the many design elements of the simulation were fairly arbitrary. One example is the equation to calculate fitness:

$$\mathrm{F} = 10E - 11P$$

Although the scaling parameters of the equation seem arbitrary, they do not make a difference. This is because by generation 1000 all populations successfully avoid poisonous mushrooms and so it is the searching ability that determines the differences in fitness. Removing the P term entirely would however make a difference as it would remove the penalty of eating a poisonous mushroom, leading to alternative behaviours.

Another arbitrary choice is the implementation decision I made for when an entity attempts to walk through the edge of the world. Instead of the entities getting stuck, I could have them ‘bounce-back’ or loop to the other side. When these alternative behaviours were implemented, no difference was found in the results besides an overall scaling of each curve. This is because the entities cannot detect the edge of the world; running into an edge is a random event so allowing them to pass through or bounce back only increases their exploration time.

The majority of arbitrary choices can be found in the simulation parameters specified by Cangelosi and Parisi (1998), listed in Table 2.1. In Section 4.1.1 we found that the simulation was robust to the

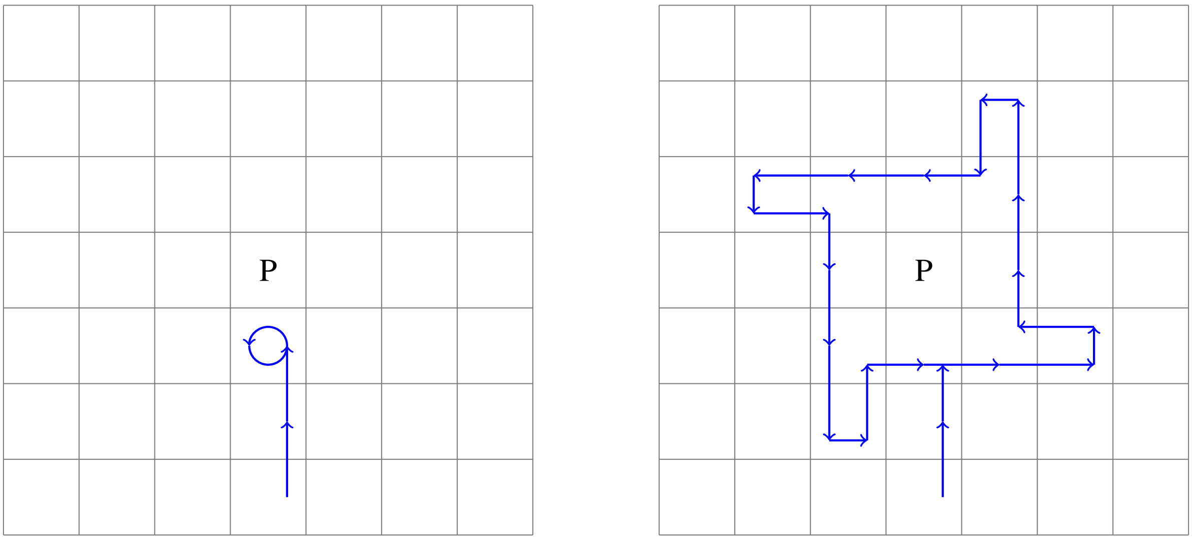


Figure 4.12: The behaviour of two entities approaching a poisonous mushroom (P). The entities are randomly chosen from generation 2000 of two **no language** populations with an average fitness of 140 (left) and 250 (right).

choice of activation function; in all cases, the populations with language performed better than those without. Observing the effect of the other parameters on the outcome of the simulation can produce some insight into whether these parameters are in fact arbitrary and the simulation is robust to these changes or whether they were chosen to produce good results.

4.3.1 Neural Network Depth

I first wanted to explore whether increasing the depth of the neural networks would affect the performance of the populations. Taking the exact same parameters as my initial experiment and increasing the depth of the neural network to contain *two* hidden layers of five units each gives us Figure 4.13. Comparing these results to Figure 4.1 does not reveal much difference; the **evolved language** and **no language** populations still reach fitnesses of 250 and 200 respectively, with the **evolved language** populations also reaching 250. Examining an even deeper network (Figure 4.14) presents a situation where the population without language performs much better, even surpassing the population with an evolved language. Performing behavioural analysis of these populations, as in Section 4.2, reveals that this is just occurring because nine out of the ten replicas are reaching the "exploratory" state with only one in the "stuck" state, thus increasing the average shown in this figure. The low performance of the **evolved language** populations is likely due to the hugely increased number of weights and biases introduced by this experiment (320 instead of 105), increasing the learning time required for these populations to develop an efficient language and reach good evolutionary performance. Examining the language produced by these populations reveals that convergence is much slower. It seems that adding in more layers has allowed these populations to escape local minima but has not allowed them to discover more sophisticated behaviours.

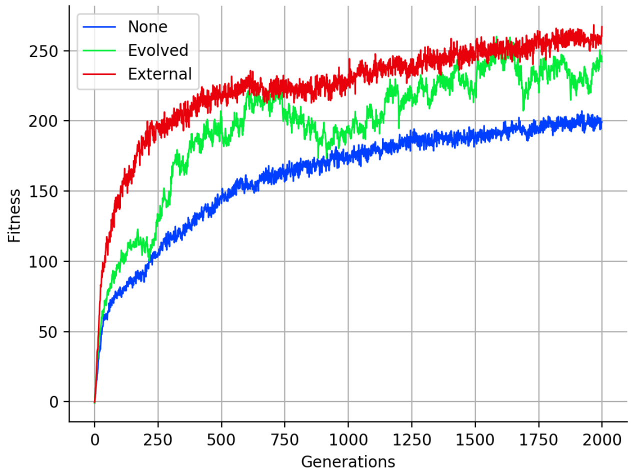


Figure 4.13: Average fitness of each population, averaged over 10 replications, with two hidden layers of five units each.

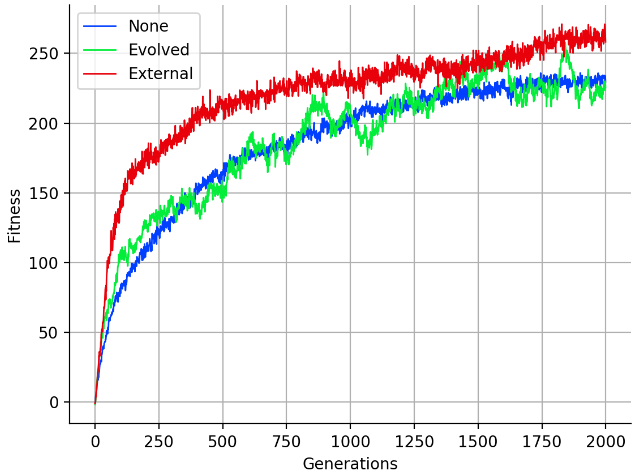


Figure 4.14: Average fitness of each population, averaged over 10 replications, with three hidden layers of ten units, five units and ten units.

4.3.2 Epoch Length

Table 2.1 lists the seemingly arbitrary simulation parameters specified by Cangelosi and Parisi (1998). Observing the effect of these parameters on the outcome of the simulation can produce some insight into whether these parameters are in fact arbitrarily chosen, perhaps to produce good results or whether the simulation is robust to these changes.

Another alteration involves changing the duration of each epoch while keeping the total number of cycles constant. In Figure 4.15 we do not observe much change in the comparison between the three populations although the fitness scores achieved are lower for all three populations. With only 15 cycles per epoch, it is likely that the entities are prevented from reaching more than one edible mushroom in each epoch. Increasing the number of cycles per epoch to 75 gives us Figure 4.16. Here we get very different results; the three populations achieve very similar fitness scores and the population with **no language** performs better than the population with the **evolved language**. This can similarly be explained from the duration of the epochs; with more time to explore, the advantage of having mushrooms labelled before approach is reduced, resulting in similar fitness scores. This does, however, show that the simulation setup is not fully robust to changes in parameters.

4.3.3 Genetic Parameters

Following the behavioural analysis in Section 4.2, it is worth exploring how changing the parameters of the genetic algorithm may affect the observed convergence to two different behaviours in the **no language** populations.

Figures 4.17 and 4.18 present the effect of increasing the mutation percentage to from 10% to 20% and 50% respectively and Figures 4.19 and 4.20 present the effect of reducing the percentage of entities chosen for reproduction from 20% to 10% and 5% respectively. It seems as if each population type is affected by these changes differently, for example the reduction in the percentage of entities kept between generations negatively affects the two populations with language but does not seem to

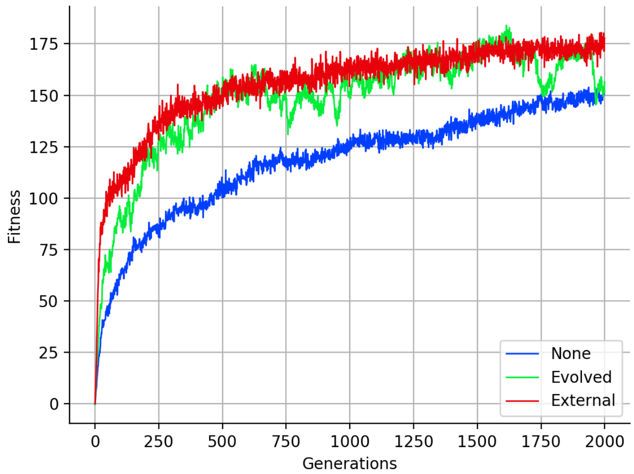


Figure 4.15: Average fitness of each population, averaged over 10 replications, with simulations consisting of 30 epochs of length 15.

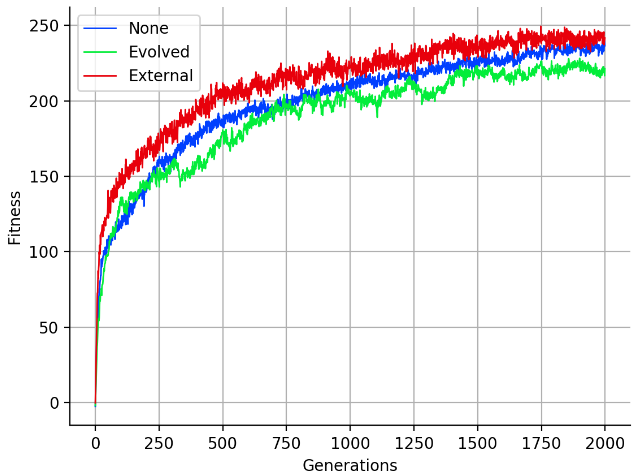


Figure 4.16: Average fitness of each population, averaged over 10 replications, with simulations consisting of 10 epochs of length 75.

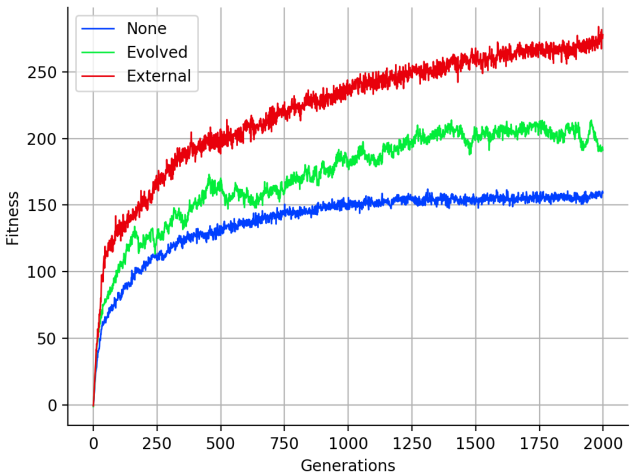


Figure 4.17: Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 20%.

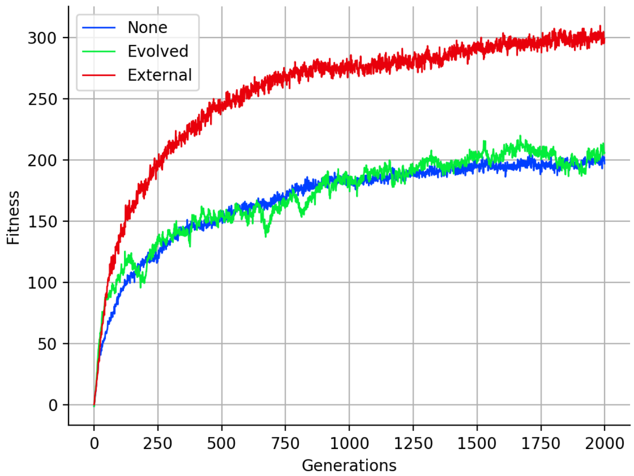


Figure 4.18: Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 50%.

have an effect on the population without language; resulting in the **evolved language** population performing worse than the population with **no language**. My view is that this is because lowering this parameter prevents the emergence of new behaviours and the two populations with language require the emergence of more complex behaviours to take advantage of the additional information. The main observation is that a slight change in these parameters results in very different outcomes to our initial experiment seen in Figure 4.1, suggesting that these parameters may have been tuned to maximise the gap between populations with language and those without.

It is worth pointing out that in all of of these experiments, we always see that the **external language** populations perform better than the **no language** populations. What seems to vary between these parameter changes is the performance of the **evolved language** population with respect to these two. With certain changes to the parameters, such as increasing the neural network depth or increasing the mutation percentage, the **evolved language** populations do not seem to converge to a productive language and do not gain an advantage in the mushroom world. Thus, although the simulation does demonstrate

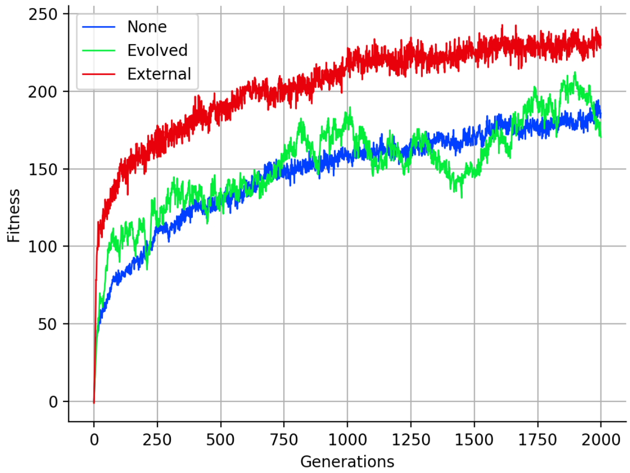


Figure 4.19: Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 10%.

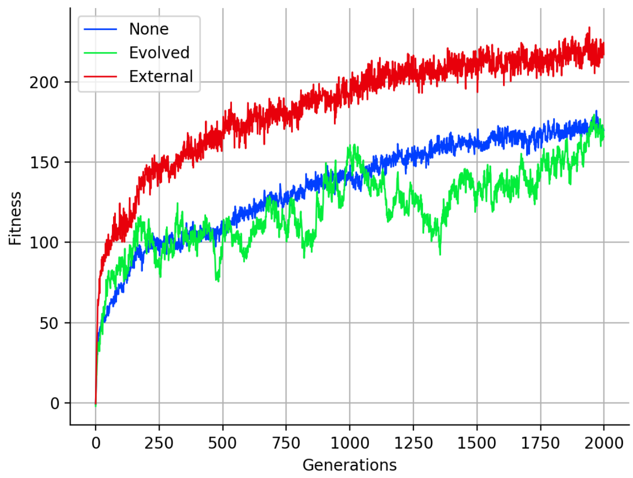


Figure 4.20: Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 5%.

that language is always beneficial, it is not fully robust to changes in simulation parameters.

4.4 Summary

Cangelosi and Parisi demonstrated that language is a useful adaptation in the "mushroom world" simulation. By implementing this simulation and through the analysis in Section 4.1.1, we have seen that this conclusion holds and we have demonstrated that an efficient language emerges through the simulated evolution.

What Cangelosi and Parisi did not analyse was how robust these conclusions were with respect to the many arbitrary design decisions made. Through the analysis made in Section 4.3, we have seen that although the main conclusion still holds, the simulation is not robust to these changes and in particular the **evolved language** and **external language** are not equivalent as was suggested by Cangelosi and Parisi in Figure 4.2.

Finally, Cangelosi and Parisi also seemed to have missed the convergence of different exploratory behaviours. In Section 4.2 we saw that two clear behaviours emerge in the population without language and that these can directly explain the fitness achieved, suggesting that exploration ability may be as important as mushroom categorisation to achieve evolutionary success in this model.

Chapter 5

Conclusion

This project was a success. All points listed in the success criteria and all items in the requirements analysis (§2.4), including extensions, were completed.

The purpose of this project was to implement the "toy mushroom world" simulation described in Cangelosi and Parisi (1998) and use it to investigate the evolution of a simple one-utterance language used to distinguish between edible and poisonous mushrooms. Entities controlled by neural networks exhibit either no language, an externally provided language or an evolved language and a genetic algorithm simulates evolution over many generations. These entities, the toy world they exist in and the genetic algorithm were all implemented successfully, along with key optimisations to reduce runtime. Throughout the project, good software engineering practices were adhered to, including:

- Issue tracking.
- Schedule management.
- A suite of unit tests.
- Continuous integration to prevent the re-emergence of bugs.

I investigated the behaviour of the simulation according to the metrics described by Cangelosi and Parisi (1998). I was able to reproduce the conclusion that populations perform better with language and furthermore that there is a correlation between language quality and fitness. I then performed additional analysis beyond this, analysing the weights of the neural networks and conducting behavioural tests to explain why different populations exhibit two seemingly distinct sets of behaviours. Finally, I explored the robustness of the system to changes made to the simulation parameters, seeking to see if these were tuned by Cangelosi and Parisi to exhibit particular results. From this I discovered that the resulting populations are in fact sensitive to these changes but that the main conclusions still hold.

5.1 Lessons Learnt

Despite having deployed an extensive suite of unit tests, a small underlying bug eluded me for a long time. This bug was in a method to generate mushrooms; the default parameter was set using a random number generator which incorrectly only evaluated once during the program. Once discovered, I created a regression test to prevent this bug's re-emergence. In the future, I will be more wary of randomness and produce more thorough tests.

Another lesson learnt was the utility of a command-line interface. Earlier in development, I created hard-coded methods for each experiment that required code changes between each run on the HPC. This slowed the evaluation cycle and resulted in many failed attempts when parameters were not set correctly or were changed before the jobs were scheduled. Introducing the command-line interface allowed me to schedule a dozen experiments at once without touching the code at all. I will therefore prioritise creating such interfaces in future projects.

Finally, during the course of the project I found that Python's dynamic typing introduced very subtle bugs that could not be discovered at compilation-time. This led me to conclude that Python would not be suitable if the aim of this project was to produce industry software, however for the scale of this project I believe that Python was a good choice. The powerful libraries lended themselves well and the dynamic typing lended itself to quick early implementation.

5.2 Further Work

Through the implementation and resulting analysis of this simulation I have demonstrated that computer simulations are a useful tool in the field of language evolution and that interesting behaviours can emerge from even the most basic neural networks. With further time I would seek to implement more expansive simulations to attempt to explain the emergence of linguistic phenomena including verb-object structures and compositionality, such as in Cangelosi (2001). I would also seek to create a visual interface for the system to allow the simulation to be used as a learning tool for students interested in learning more about agent-based modelling.

Bibliography

Cangelosi, A. (2001). Evolution of communication and language using signals, symbols, and words. *IEEE Transactions on Evolutionary Computation*, 5(2):93–101.

Cangelosi, A. and Parisi, D. (1998). The emergence of a 'language' in an evolving population of neural networks. *Connection Science*, 10(2):83–97.

Cangelosi, A. and Parisi, D. (2002). *Simulating the evolution of language*. Springer Science & Business Media.

Cavalli-Sforza, L. L. (1997). Genes, peoples, and languages. *Proceedings of the National Academy of Sciences*, 94(15):7719–7724.

Clark, E. V. (1995). *The lexicon in acquisition*, volume 65. Cambridge University Press.

De Boer, B. (1997). Generating vowel systems in a population of agents. In *Fourth European Conference on Arficial Life, Brighton*. Citeseer.

De Villiers, J. and Barnard, E. (1993). Backpropagation neural nets with one and two hidden layers. *IEEE transactions on neural networks*, 4(1):136–141.

Fleagle, J. G., Assefa, Z., Brown, F. H., and Shea, J. J. (2008). Paleoanthropology of the Kibish Formation, southern Ethiopia: Introduction. *Journal of Human Evolution*, 55(3):360–365.

Holland, J. H. et al. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

Kirby, S. and Hurford, J. R. (2002). The emergence of linguistic structure: An overview of the iterated learning model. In *Simulating the evolution of language*, pages 121–147. Springer.

Langton, C. G. (1997). *Artificial life: An overview*. Mit Press.

Mycroft, A. (2019). Concepts in programming languages.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

Parisi, D. and Cangelosi, A. (2002). A unified simulation scenario for language development, evolution and historical change. In *Simulating the evolution of language*, pages 255–275. Springer.

Rendell, P. (2002). Turing universality of the game of life. In *Collision-based computing*, pages 513–539. Springer.

Appendix A

World Representation Benchmark

A.1 world_benchmark.py

```
""" This is a benchmark test to show that the dictionary representation
of the toy world is more efficienct than the array representation for the
closest_mushroom() method, which involves iterating through all the mushrooms
in the world
"""

import time
import random
import matplotlib.pyplot as plt


def list_of_positions():
    """ Generate a list of world positions """
    positions = []
    while len(positions) != 20:
        x = random.randrange(0, 20)
        y = random.randrange(0, 20)
        if (x, y) not in positions:
            positions.append((x, y))
    return positions


def time_dictionary():
    """ Create a dictionary world and iterate through to find each mushroom """
    # Each mushroom is a key in the dictionary
    world = {}
    positions = list_of_positions()
    for position in positions:
        world[position] = random.randrange(1, 1000)

    time_start = time.time()

    # Iterate through the world,
    # incrementing each "mushroom"
    for position in world:
        world[position] += 1

    time_end = time.time()
    return time_end - time_start


def time_array():
    """ Create an array world and iterate through to find each mushroom """
    # Each mushroom is a non-zero value in the 2D array
    world = [[0 for _ in range(20)] for _ in range(20)]
    positions = list_of_positions()
```

```
    for (x, y) in positions:
        world[x][y] = random.randrange(1, 1000)

    time_start = time.time()

    # Iterate through the world,
    # incrementing each "mushroom"
    for x in range(20):
        for y in range(20):
            if world[x][y] > 0:
                world[x][y] += 1

    time_end = time.time()
    return time_end - time_start


total = 10000

d_times = [time_dictionary() for _ in range(total)]
d_mean = sum(d_times) / total
d_var = sum([(t - d_mean)**2 for t in d_times]) / total
a_times = [time_array() for _ in range(total)]
a_mean = sum(a_times) / total
a_var = sum([(t - a_mean)**2 for t in a_times]) / total

print("DICTIONARY REPRESENTATION")
print("Mean: {} \nVariance: {}\n".format(d_mean, d_var))
print("ARRAY REPRESENTATION")
print("Mean: {} \nVariance: {}".format(a_mean, a_var))
```

Appendix B

Simulation Interactivity

This section provides evidence of the interactivity features of the system and examples of how the simulation can be run using the command-line interface.

Figures B.1 and B.2 present the command-line interface information screens for the simulating and analysis modules respectively; giving the parameters used to configure these processes.

Figure B.3 shows the result of running an interactive simulation. Each generation is run following a press of ENTER by the user, or *n* generations will run if the user enters a number *n*. With each generation, the live average fitness graph is updated and the exact fitness score for each entity in the population is displayed as a sorted list in the terminal.

If the user enters `watch` *n* for some index *n* into the array of entities, this will allow the user to watch a run of the simulation for a particular entity, as seen in Figure B.4. This displays the current epoch, cycle number, then a grid representing the world that the entity exists within. Mushrooms as displayed as emojis and the entity is displayed as a triangle with the tip of the triangle giving the direction that the entity is facing. Below this grid, some information about the simulation is given along with the inputs passed to the `behaviour()` method and the outputs received.

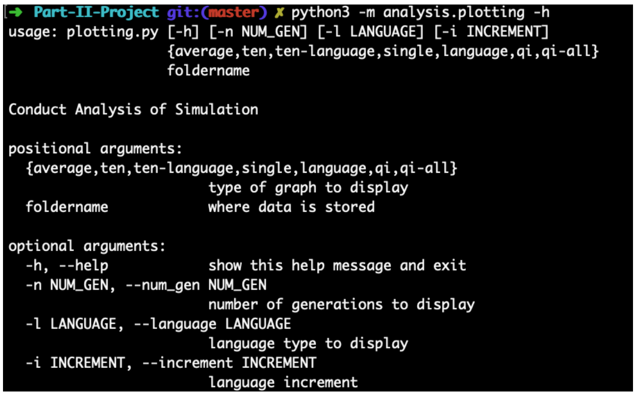
```
➜ Part-II-Project git:(master) ✗ python3 -m simulating.simulation -h
usage: simulation.py [-h] [--single] [--interactive] [--num_epo NUM_EPO]
                     [--num_cyc NUM_CYC] [--num_ent NUM_ENT]
                     [--num_gen NUM_GEN] [--per_mut PER_MUT]
                     [--per_keep PER_KEEP] [--no_rec_lang]
                     [--rec_lang_per REC_LANG_PER] [--no_rec_ent]
                     [--rec_ent_per REC_ENT_PER] [--no_rec_fit]
                     [--activation {identity,sigmoid,relu}] [--linear]
                     [--hidden_units HIDDEN_UNITS] [--start_from START_FROM]
                     {None,Evolved,External} foldername

Run the full genetic algorithm

positional arguments:
  {None,Evolved,External}
                        language type used in the simulation
  foldername            where results are stored

optional arguments:
  -h, --help            show this help message and exit
  --single, -s          run a single simulation for one entity
  --interactive, -i     run with interactivity
  --num_epo NUM_EPO     number of epochs per single simulation run
  --num_cyc NUM_CYC     number of cycles per epoch
  --num_ent NUM_ENT     number of entities in the population
  --num_gen NUM_GEN     number of generations to run for
  --per_mut PER_MUT     percentage of weights to mutate in reproduction
  --per_keep PER_KEEP   percentage of population that reproduces
  --no_rec_lang         don't store the language
  --rec_lang_per REC_LANG_PER
                        how frequently to store the language
  --no_rec_ent          don't store the population
  --rec_ent_per REC_ENT_PER
                        how frequently to store the population
  --no_rec_fit          don't store the fitness
  --activation {identity,sigmoid,relu}
                        the activation function used in the internal layer
  --linear              don't use an activation on the final layer
  --hidden_units HIDDEN_UNITS
                        nodes in hidden layers of the neural network
  --start_from START_FROM
                        generation to start the simulation from
```

Figure B.1: Result of running `python3 -m simulating.simulation -h` to display the help page for the command-line interface of the simulation module. The required parameters are the foldername (for storing results) and the language type, all other parameters have a default value.



```
➜ Part-II-Project git:(master) ✗ python3 -m analysis.plotting -h
usage: plotting.py [-h] [-n NUM_GEN] [-l LANGUAGE] [-i INCREMENT]
                   {average,ten,ten-language,single,language,qi,qi-all}
                   foldername

Conduct Analysis of Simulation

positional arguments:
  {average,ten,ten-language,single,language,qi,qi-all}
                        type of graph to display
  foldername            where data is stored

optional arguments:
  -h, --help            show this help message and exit
  -n NUM_GEN, --num_gen NUM_GEN
                        number of generations to display
  -l LANGUAGE, --language LANGUAGE
                        language type to display
  -i INCREMENT, --increment INCREMENT
                        language increment
```

Figure B.2: Result of running `python3 -m analysis.plotting -h` to display the help page for the command-line interface of the analysis module. The required parameters are the foldername (where the results are found in order to plot) and the type of graph to display.

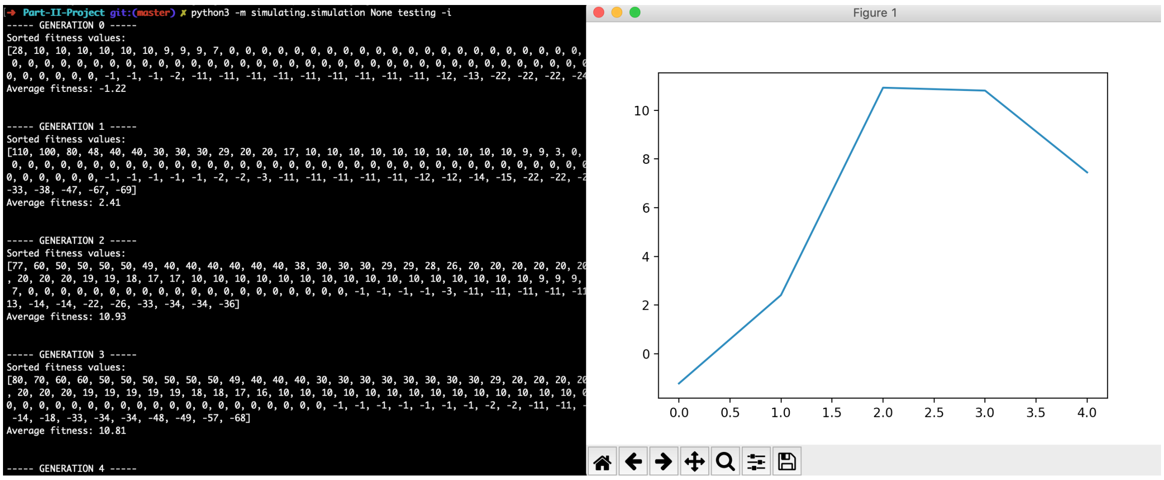


Figure B.3: Visualisation of running a default simulation for a **no language** population using the interactivity flag (`-i`). The left side shows the terminal output and the right side shows the live fitness graph.

Figure B.4: Visualisation of watching the behaviour of a single entity in a population, selected from an interactive simulation by entering `watch 0`.

Appendix C

Project Proposal

Introduction

Language has evolved and therefore probably gave an evolutionary advantage to the individuals that exhibited it. As Angelo Cangelosi and Domenico Parisi described in a 1998 paper[1], it is difficult to investigate the evolutionary origin of language and the selective pressures that may have originated language due to the limited evidence available. They propose using computer simulations of evolutionary scenarios to investigate this. In the paper referenced, they describe a simulated toy world where agents controlled by neural-networks interact with an environment of mushrooms that are edible and poisonous. This simulation and the ideas explored in the paper will be the basis for my project.

In the paper, Cangelosi and Parisi use small feed-forward neural networks to control the behaviour of each agent. The weights are initially random; a genetic algorithm is used to improve the fitness of the species over many generations. The agents are also given linguistic abilities; input and output nodes of the neural networks produce signals that allow for communication.

By creating three different populations (one without language, one with an externally imposed language and one with an evolved language) we can investigate the evolutionary advantage of language. Furthermore, it allows us to investigate a key question posed in the paper: *"Since language requires the parallel evolution of linguistic production and linguistic comprehension, how can language evolve when it has a purely informative function and therefore it is advantageous to the receiver but not the producer?"*

For this project, I will re-implement the simulation described. I will then create analysis tools to investigate the findings of the paper to see if I observe the same results.

Starting Point

I have a small amount of experience in programming simulations; for my A-Level project in 2016, I created a simulation of virus propogation between mosquito and human agents in the Unity game engine.

[1] `https://doi.org/10.1080/095400998116512`

I do not have any experience programming neural networks, however, I am confident that I understand the backpropagation algorithm and basic neural network structure through the Artificial Intelligence course I took last year. In the papers I plan to reference, Cangelosi very clearly describes the structure of the neural networks he uses and I am confident that I will be able to follow his work.

Over the summer I read a book titled Simulating the Evolution of Language which gave me an overview of the techniques used in this field. Alongside the Formal Models of Language course that I took last year, I now have a sufficient base of understanding to begin this project.

Work to be Done

The work for this project can be roughly divided into two stages; implementing the simulation and constructing the means of evaluating my implementation against the findings in the original paper. I will also regularly be creating tests to evaluate my simulation.

Implementing the Simulation

1. Set up the simulation environment by creating the world grid and implementing the properties of poisonous and edible mushrooms. Create the simulation loop divided into regular ‘epochs’.
2. Create the agents for the simulation, giving them position and energy properties.
3. Implement feedforward neural networks to control the behaviour of the agents; input units to identify the location of the nearest mushroom, visual perception units to observe mushroom properties (only when close enough) and signal perception units for when language is implemented. The output units control the movement of the agent and production of signals. There will also be hidden units.
4. Implement the genetic algorithm that runs after all agents complete the simulation. The fittest agents are determined by the energy level (based on eating edible mushrooms and avoiding poisonous mushrooms). The fittest agents are then chosen for asexual reproduction, producing offspring that have genetic mutations in the form selecting a percentage of the weights to change by a random amount.
5. Create two different populations, one without language (where the signal perception units are always set to the same, constant value) and one with an externally imposed language (where the signal perception units are set to one of two signals depending on the type of the nearest mushroom).
6. Create a third population with an evolved language. Instead of externally imposed signals, in each simulation cycle one of the other agents is randomly selected as the ‘speaker’ and its output is connected to the input signal perception units of the ‘listener’.

Analysis

1. Plot the average fitness over the number of generations to compare between the three populations.

2. Produce some behavioural tests to investigate the behaviour of random individual organisms at specific generations.

3. Plot the frequency distribution of the different signals produced by the individuals with the evolved language using a 'naming task'.

4. Calculate the Quality Index (QI) of the language produced by the population without language and the population with an evolved language to investigate the genetic advantage of producing productive signals. The QI evaluates the efficiency of a language based off of three criteria; (1) functionally distinct categories are labeled with distinct signals, (2) a single signal tends to be used to label all the instances within a category and (3) all the individuals in the population tend to use the same signal to label the same category.

5. Investigate the correlation between QI of the language and the fitness of the species to determine if change in the language or in the categorisation skill of the agents affects the other ability.

Testing

To evaluate my project and ensure that my simulation implementation is functional, I will create an ensemble of unit tests for each of the tasks above. These will be created in parallel as I develop each part of the implementation. For the simulation, this will involve small examples or scenarios to show that each part of the simulation is fully functional.

Success Criterion

The project will be deemed a success if I can implement the simulation as described in the tasks above (evaluated by my unit tests) and if I can implement the analysis tools to compare the findings of my implementation to the findings of the original paper.

Timetable and Milestones

I've broken down this timetable into two and three week intervals. At the end of December, I will be writing my Progress Report and simultaneously making adjustments to the timetable as needed.

25th October – 10th November

Middle of Michaelmas Term. Includes first deadline for NLP coursework.

Task: Create a high-level design of the system. Set up the project files with a version-control system. Experiment with creating small simulations in python and do suitable research into Neural Network libraries.

Milestones: Have a git repository with project files. Have a design plan with specific details of the simulation confirmed.

11th November – 24th November

Middle of Michaelmas Term.

Task: Complete implementation tasks 1 and 2 as described above. Experiment with adding Neural Networks to control the behaviour of the agents.

Milestones: Have a working simulation environment with poisonous and edible mushrooms. Have agents with positions and energy values but no functional neural networks yet.

25th November – 8th December

End of Michaelmas Term. Includes second and third deadline for NLP coursework.

Task: Complete implementation tasks (3). Start working on implementation task (4).

Milestones: Have the agents successfully controlled by neural networks. Be able to run an entire lifespan of one agent within the simulated world.

9th December – 22nd December

Christmas holiday. Will likely be in Cambridge to help with Queens' interviews.

Task: Complete implementation tasks (4), (5) and (6). Also aim to complete analysis tasks (1) and (3).

Milestones: Have a fully functional simulation that allows for running a thousand generations of a population of agents. Have three different populations to compare; one without language, one with an externally imposed language and one with an evolved language. Have a tool to graph the average fitness of each population over the number of generations and another tool to view the probability distribution of the signals chosen for the evolved language over the number of generations.

23rd December – 12th January

Christmas holiday. Will take a break to revise Michaelmas courses and to spend time with family.

Task: Complete analysis task (2). Write the Preparation chapter of the Dissertation. Review the timetable for the remainder of the project and adjust in light of experience so far. If ahead of schedule, plan time for extensions. Start to plan tests cases.

Milestones: An outline of the dissertation document with a completed Preparation section.

13th January – 2nd February

Start of Lent term. Will have regular labs for Mobile Robot Systems.

Progress Report Deadline: 31st January

Task: Write the Progress Report. Start to fill out the Implementation chapter of the Dissertation. Complete analysis tasks (4) and (5).

Milestones: Progress report submitted and entire project reviewed both personally and with overseers. Have tools to plot the Quality Index of the evolved language against the fitness of the population. At this point, all the tasks in the **Work to Do** section will have been completed, satisfying the **Success Criteria**.

3rd February – 23rd February

Middle of Lent term. Both deadlines for the Mobile Robot Systems assignments.

Task: Begin analysis of the simulation. Begin to work on extensions to the project, keeping in mind time needed to write the Dissertation.

Milestones: Have the start of a test suite with a series of diagrams to use to evaluate my implementation.

24th February – 15th March

End of Lent term. Deadline for the Mobile Robot Systems mini-project report.

Task: Complete testing. Evaluate the outcomes of the tests against the findings in the original paper. At this point, the second half of the **Success Criteria** will have been achieved. If needed, revise the implementation to be clean, documented and consise. Work on other extensions.

Milestones: Examples and test cases run with results collected. Code should perform a variety of interesting tasks and should be in a state that in the worst case it would satisfy the examiners with at most cosmetic adjustment

16th March – 5th April

Start of Easter holiday. Might stay in Cambridge for part of it to work. Will balance revision and work on the project.

Task: Complete work on any extensions. Draft the Evaluations and Conclusions chapters of the Dissertation.

Milestones: Extensions almost complete. Skeleton of entire Dissertation in place.

6th April – 19th April

End of Easter holiday. Might get back to Cambridge early to work. Will balance revision and work on the project.

Task: Complete the Implementation and Introduction chapters of the Dissertation. Send the full draft to Director of Studies and Supervisors by 21st of April.

Milestones: Dissertation essentially complete, with large sections of it proof-read by Supervisors and possibly friends and/or Director of Studies.

20th April – 8th May

Start of Easter Term. Will be balancing revision, lectures and final work on the project.

Final Deadline: 8th May

Task: Finish Dissertation, preparing diagrams for insertion. Review the whole project, checking the Dissertation and spending the final few days on whatever is in greatest need of attention. Aim to submit the dissertation at least a week before the deadline.

Milestone: Submission of Dissertation

Possible Extensions

Graphic Visualisation

As a side extension, I could implement a visual interface to watch the life of one agent within the simulation. This would involve rendering a simple 2D world with textures for the agents and mushrooms. This could be expanded further by adding a User Interface for setting up the simulation and having windows showing the progress as it occurs live.

Symbolic Theft vs. Sensorimotor Toil

In a 2000 paper[2], Cangelosi and Harnad use a similar same toy world of mushrooms and foragers to place two ways of acquiring categories in direct competition with each other. They compare "sensorimotor toil" (where categories are acquired through real-time, feedback-correct, trial and error experience) to "symbolic theft" (where new categories are acquired by hearsay from boolean combinations of symbols *describing* them). They find that the origins of natural language could be explained by the apparent infinitely superiority of a hybrid symbolic/sensorimotor combination compared to purely sensorimotor precursors.

As an extension, I could expand my simulation to investigate the findings of this paper. This involves implementing a more complicated neural network, adding supervised learning through back-propagation, implementing more sophisticated mushroom features and expanding the simulation to host multiple populations at once.

Investigating the Evolution of Syntax

In a 1999 paper[3], Cangelosi expands the toy mushroom world simulation further to investigate how languages that use combinations of words (such as the "verb-object" rule) can emerge by auto-organisation and cultural transmission. Mushrooms are either edible or poisonous but also have one of three colours - the edible mushrooms of a particular colour correspond to a particular action in response.

[2] `http://cogprints.org/2036/`
[3] `https://link.springer.com/chapter/10.1007/3-540-48304-7_86`

In this extended simulation, after the first 300 generations parents and children co-exist within the simulated world. Parents teach the evolved language to their children. Children undergo a Listening Task (where parents describe the closest mushroom) and a Naming Task (where the mushroom name is used for supervised learning through backpropagation).

This is a substantial increase in complexity but allows would allow me to investigate the evolution of a more complex language.

Resources Declaration

For this project, I plan to use my computer, (2.8 GHz CPU, 16 GB RAM, 750GB Flash Storage, macOS Mojave). The code will be regularly pushed to a GitHub repository to be able to recover from failure or loss on my local machine. I will also create weekly backups on an external hard-drive to provide another source of recovery. Should my machine fail, I will be able to continue working on an MCS machine. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will also need to use the high powered computer when running large simulations to save processing time.