Zébulon Goriely

April 8, 2020

Simulating Language Learning and Evolution

Computer Science Tripos – Part II

Queens' College

Chapter 1

Introduction

1.1 Motivations

Humans evolved to produce and understand 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. Language evolution is a once-occurring[1], 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, 2012). 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.

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 project, I will use a simulation to explore the genetic advantage of populations of entities who exhibit language over those who do not.

[1] (to our knowledge)

1.2 Prior Work

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.

In further papers, Cangelosi explores how this simulation can be made more complex by introducing cultural transmission of language and finds that in compositional languages with a verb-noun structure, verbs have a larger positive effect on the performance of the species (Cangelosi, 2001). As an extension to my core project I will investigate the effect of incorporating cultural transmission to my simulation. [TODO: DO THIS EXTENSION]

1.3 Project Overview

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

- Implementing the environment populated with edible and poisonous 'mushrooms'
- Implementing feed-forward neural networks from scratch to simulate the behaviour of the agents
- Implementing the genetic algorithm used to evolve the population over many generations
- Implementing three populations types categorised by their use of 'language'
- Implementing a suite of data gathering tools, interactivity features and analysis methods to examine the behaviour of the simulation

In this chapter I have introduced why it can be useful to use computer simulations to help research the evolution of language. In Chapter 2, I cover all of the preparations made for this project, including introducing the key concepts and discussing project management and software tools. In Chapter 3, I present my implementation of the simulation. In Chapter 4, I evaluate my project, comparing my results to Cangelosi and Parisi (1998) before presenting my concluding remarks in 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.

The organisms live in an environment populated by two different types of mushroom; edible and poisonous. The organisms will reproduce based on their ability to eat edible mushrooms and avoid the poisonous ones. They will 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 organisms, the mushrooms will have different properties. Edible mushrooms will resemble each other but will not be identical and likewise for poisonous mushrooms.

Each organism 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 will be 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.

These constants are fairly arbitrarily chosen by Cangelosi & Parisi and 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 are used to average out the randomly generated positions of mushrooms so that it is really the behaviour of the organism that is being tested, not the random choice of environment. The 50 simulation cycles along with the specific dimensions of the environment ensure that the organism will likely be able to reach at least one edible mushroom but will likely not be able to eat all edible mushrooms in time. This encourages productive strategies

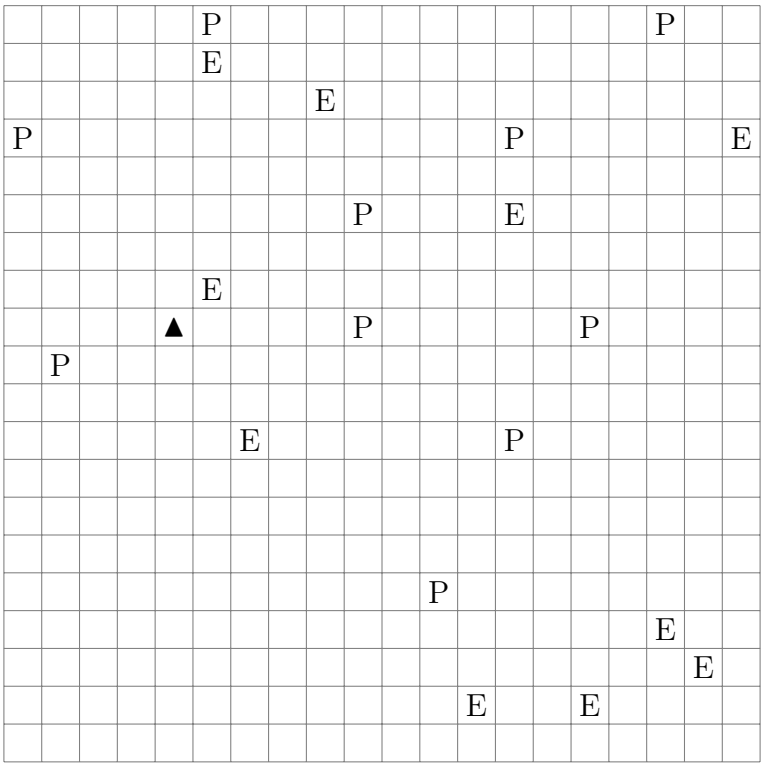

Figure 2.1: An organism (▲) in the simulation environment with edible (E) and poisonous (P) mushrooms.

to search for edible mushrooms within the limited number of simulation cycles. Intuitively, this reflects the limited time that living organisms have to search for food in their lifetimes. A full list of constants can be seen in Table 2.1.

2.2 Entities

In the simulation, organisms are represented by *entities*. Entities can *perceive* their surroundings and *act* accordingly.

Here 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. Instead of incorporating an 'eat' action, a mushroom will be considered eaten when the entity moves into the cell that the mushroom occupies. 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.

With this abstract definition of entities in place, I describe the design of my system in the next section.

2.2.1 Feed-Forward Neural Networks

Consistent with Cangelosi and Parisi (1998), I will 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 a signal by computing a non-linear function of the sum of the 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 there are no loops present to allow a signal to traverse a layer multiple times. 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 computes an input using a *propagation function* as a weighted sum of the outputs of predecessor nodes, incorporating a *bias* added to the result of the propagation. 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, I will explore the use of three common functions:

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

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). This structure can be seen in Figure 2.2. By altering the weights and biases used by the network, the entity will respond differently to the perceptual inputs and will produce 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 (**?**). 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, a *genetic representation* which can be mutated and altered. The evolution starts from

[1]ReLU stands for Rectified Linear Unit (Nair and Hinton, 2010)

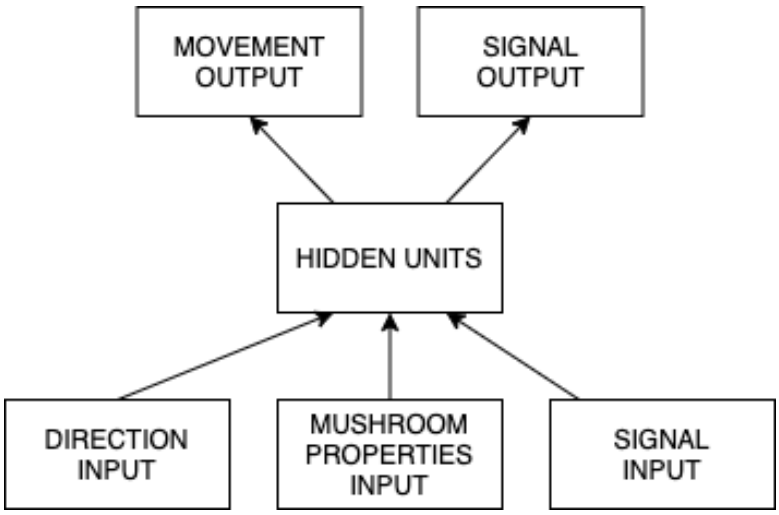

Figure 2.2: Structure of a fully-connected ANN to control entity behaviour

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 will be calculated according to the number of edible and poisonous mushrooms eaten by the entity during the simulation, E and P respectively:

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

This 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.

Finally, the 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 by randomly altering a percentage of the weights and biases in the neural network. In Cangelosi and Parisi (1998), the top 20% of entities are selected from a population of 100. Each of these entities produces five entities by randomly mutating 10% of their weights and biases, producing a new generation of 100 entities. 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 will 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

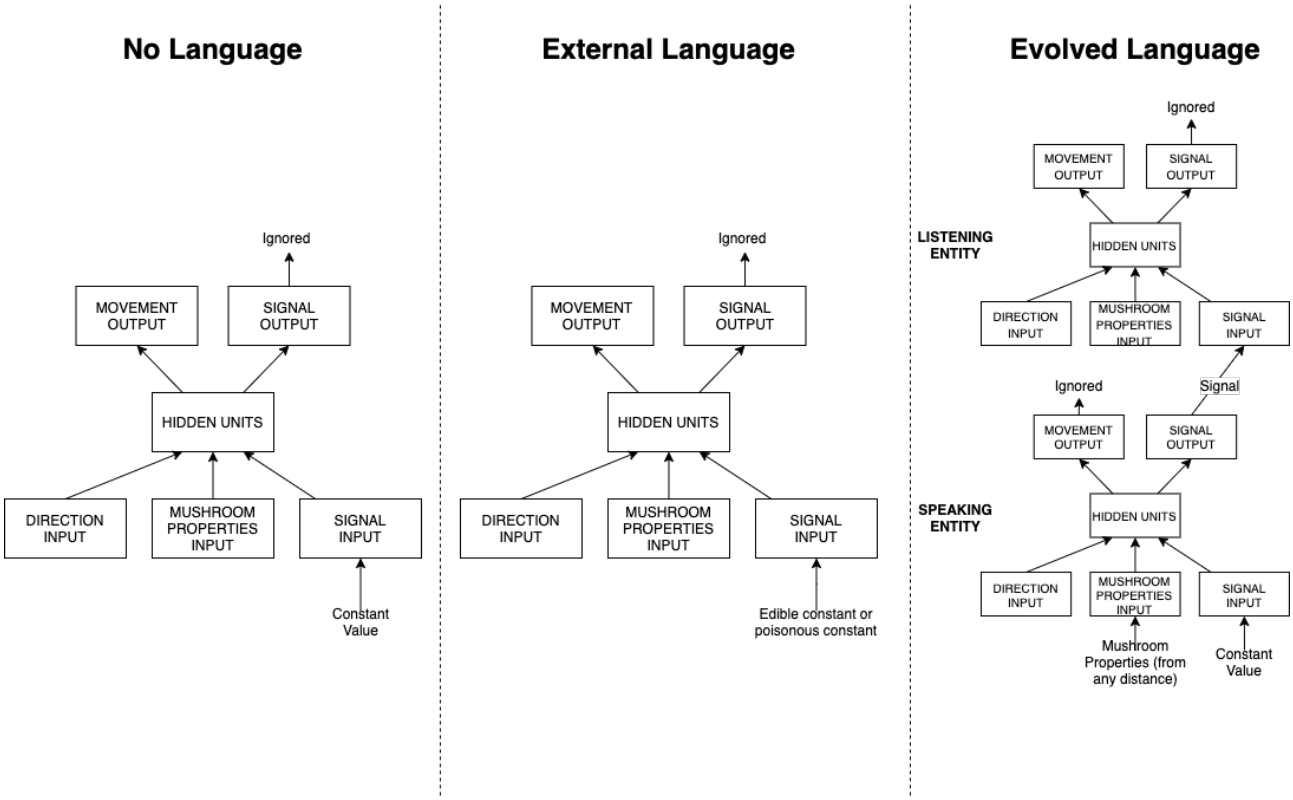

Figure 2.3: Differences in simulation structure for the three population types

- 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 some linguistic signal.

The population with the **external language** is slightly different. Here, I imagine that at the closest mushroom to the entity, another entity is present and can see the properties of the mushroom, generating an appropriate linguistic signal for the primary entity. The language is "externally provided" because I will enforce that 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 I will not enforce the use of particular signals. Instead, I will allow the population to derive its own signals. This will be done by pairing the entity with another randomly chosen entity from the population at each simulation cycle (a 'speaking' entity). This second entity has the role of labelling the mushroom for the listening entity. Both entities are given the same inputs but the speaking entity additionally receives the properties of the closest mushroom, no matter the distance. The linguistic output of this second entity will be used as the linguistic input to the primary entity; simulating a one-word utterance.

The differences between these populations can be seen in Figure 2.3.

Constant	Description	Default Value
dim_x	The width of the simulation environment	20
dim_y	The height of the simulation environment	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 organisms 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 organisms chosen to reproduce	20

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

2.3 Simulation Analysis

2.3.1 Generational Fitness

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) plot the average fitness across 1000 generations to compare the three population types discussed in Section 2.2.3. This will be the primary way that I compare different runs of the simulation.

2.3.2 Efficiency of Language

Cangelosi and Parisi (1998) were also interested in the *efficiency* of the language produced by these organisms. They gave three requirements for a population having an efficient language:

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

These requirements were based on principles that Clark (1995) argues govern a child's acquisition of a lexicon. 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). This produces 80 signals per entity. The frequency distribution for the signals produced for edible and poisonous mushrooms by all entities can be plotted and this will be the second way I analyse the simulations.

Cangelosi and Parisi (1998) also described the calculation of a "Quality Index" (QI) to describe the efficiency of a language. The QI is 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 and y_i are the frequencies of signals used for poisonous and edible mushrooms respectively, as calculated from the "naming task" and x_e and y_e are the expected percentages in the case of a flat distribution. 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 when only one signal is used for the category.

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.

This will be the third way that I analyse the simulations; in particular I will examine the correlation between language efficiency and population fitness.

2.4 Requirements Analysis

My project involves re-implementing the simulation described in Sections 2.1 - 2.2 and analysing this simulation in regards to the three 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).

Implementing the Simulation

1. Have a 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. Have a genetic algorithm that executes after all agents complete the simulation, as described in Section 2.2.2.

4. Have 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.

Analysis of the Simulation

1. Have plots of the average fitness over the number of generations to compare between the three populations.
2. Have behavioural tests to investigate the behaviour of random individual organisms at specific generations.
3. Have plots of 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.
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 entities affects the linguistic ability.

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 and 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].

2.6.2 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, evaluate my simulation in detail and work on my extension.

2.6.3 Version Control

For version control, I used Git to track change with a remote repository on GitHub[5]. This 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, as discussed below. In addition, I also performed a hardware backup to a hard drive twice a week.

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.
2. I used pylint[7] to lint my code. This allowed me to ensure my code was readable and that I was complying with Google's Python style guide[8].

[2]https://www.scipy.org/
[3]https://matplotlib.org/
[4]https://docs.pytest.org/
[5]https://github.com/
[6]https://travis-ci.org/
[7]https://www.pylint.org/
[8]http://google.github.io/styleguide/pyguide.html

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 my Travis script.

2.6.5 Development Environment

I used Visual Studio Code[10] as a development environment, due to the friendly interface and useful set of plugins that integrated nicely with git, pylint and pytest.

To run my simulations, I used the University's High Performance Computing service. I created a set of Slurm scripts with a bash script that would allow me to schedule 30 simulations to run at once and independently. The results of these simulations were saved to files that I could copy to my local file system using SSH.

2.6.6 Code License

My source code is publicly available on GitHub with a README to explain how to use it. The project is licensed under the MIT license, for free modification and reuse while limiting my liability. It also preserves the copyright notice.

2.7 Summary

The evolution of language is an challenging problem that can be explored using simulations. In this project I will implement the "mushroom world" environment populated by entities controlled by neural networks with a genetic algorithm to model evolution. Three populations will be implemented; 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, I will be able to investigate the parallel development of the ability of the entities to categorise mushrooms and their ability to name them.

[9] https://github.com/google/yapf
[10] https://code.visualstudio.com/

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 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 2.3. It also contains another Python file that 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.

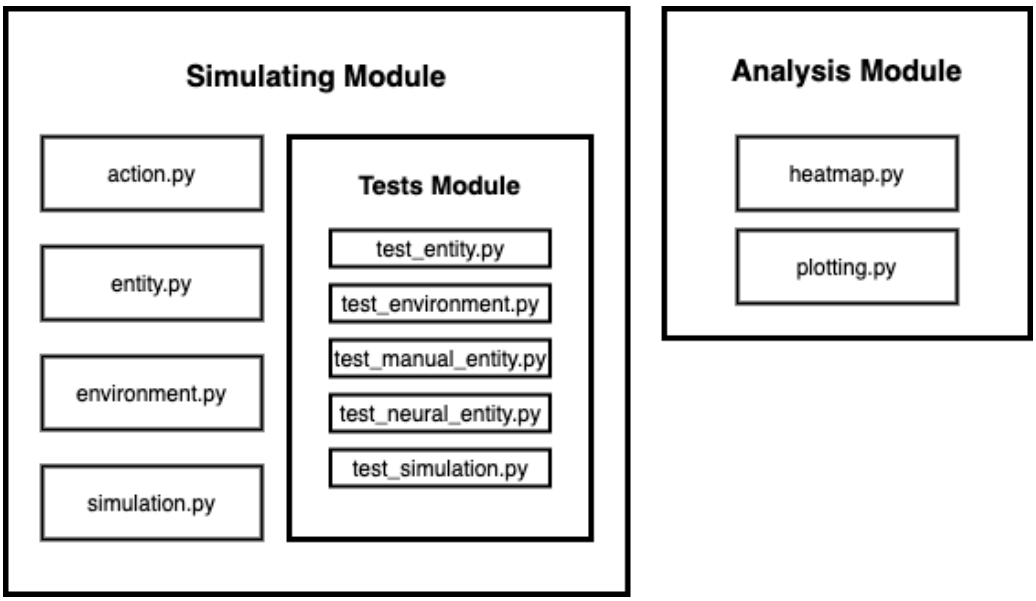

Figure 3.1: Core modules for my project

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 with some variations to simulate visual inconsistencies.

To quickly generate mushrooms and test if mushrooms are edible or poisonous, I implemented functions `make_edible()`, `make_poisonous()`, `is_edible()` and `is_poisonous()`. For efficiency, these are implemented using bit-wise arithmetic.

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, we want this to be efficient and it is faster to iterate through 20 keys in a dictionary than through 400 cells in an array.

Two run-time exceptions are implemented, `WorldFull` and `MushroomNotFound`. The methods that throw them are described below. Positions in the world are passed between methods as integer pairs (x, y). For clarity, the type `pos` will be used when such a pair is used.

The environment object has the following utility methods for manipulating and querying the status of the world:

- `random_position(): pos` — Returns a random position within the dimensions of the world

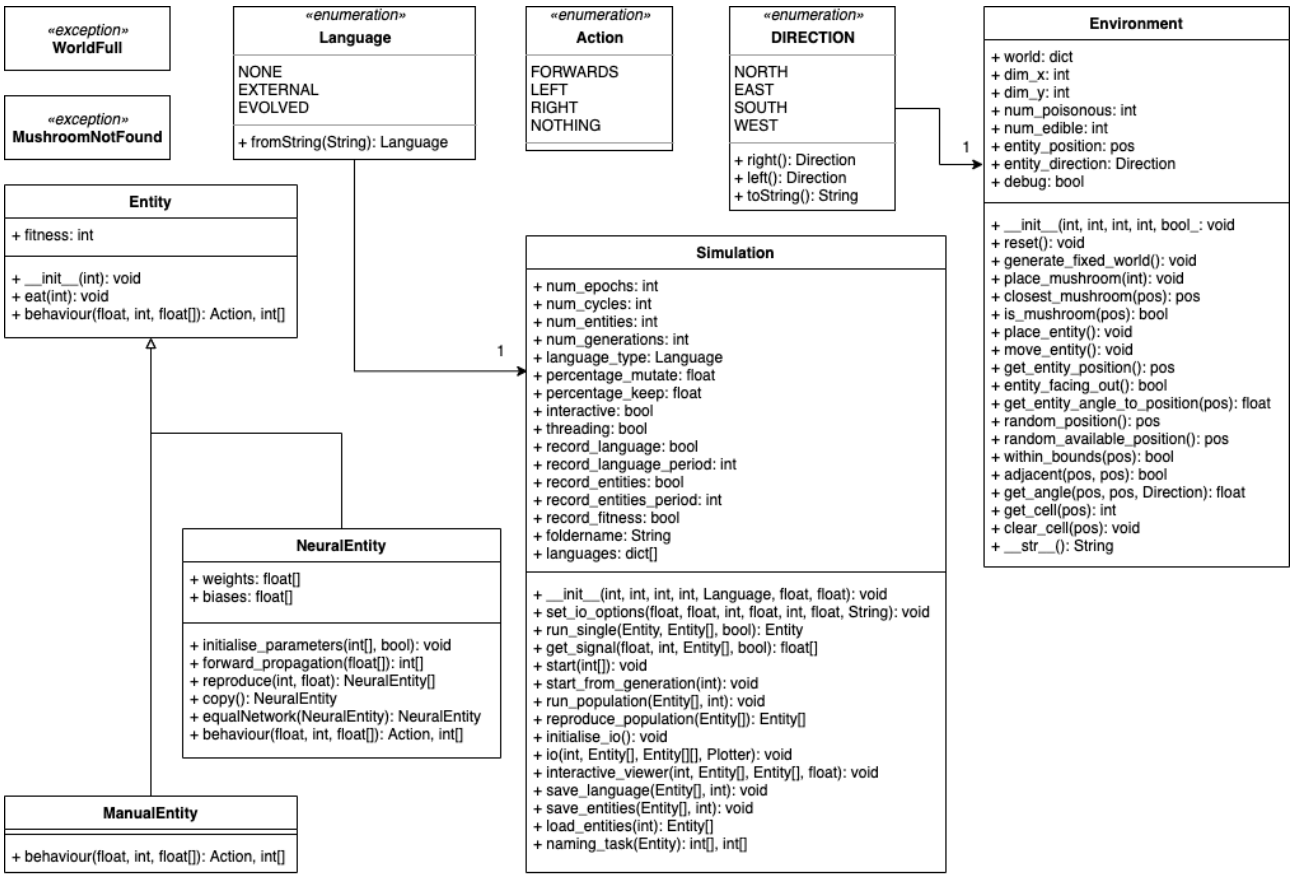

Figure 3.2: UML Diagram for the Simulating Module

- `random_available_position(): pos` — Calls `random_position()` iteratively until an unoccupied position is found. Throws a `WorldFull` exception if no cell is available
- `within_bounds(pos): bool` — Returns true if the position is within the world
- `adjacent(pos, pos): bool` — Returns true if the two passed positions are adjacent
- `get_angle(pos, pos, Direction): float` — Returns the angle from one position to another, measured from 0 to 1 clockwise from the direction given
- `get_cell(pos): int` — Given a position, returns the mushroom at that position or 0 if the cell is empty
- `clear_cell(pos): void` — Removes a mushroom at the position specified and has no effect if the cell is empty

I then implemented the following methods for manipulating mushrooms in the world:

- `place_mushroom(int): void` — Given a mushroom, places it in the world by calling `random_available_position()` and throws WorldFull if there is no space left
- `closest_mushroom(int): pos` — Returns the position of the closest mushroom in the world from a given position, using the Manhattan distance
- `is_mushroom(pos): bool` — Returns true if there is a mushroom at the position specified

The `closest_mushroom()` method loops through the position keys in the dictionary and for each of these calculates the Manhattan distance. The Manhattan distance is used because it is faster to calculate, it is always an overestimate (or equal to) the Euclidean distance and because the Entity can only move rectilinearly.

```
def closest_mushroom(self, pos):
    """ Returns the position to the closest mushroom in the world.

    Args:
        pos (int, int): Position searching from.
    Raises:
        MushroomNotFound: No mushrooms found.

    """

    if len(self.world) == 0:
        raise MushroomNotFound("No Mushrooms in World")

    dist = self.dim_x + self.dim_y + 1
    mush_pos = (-1, -1)
    for (i, j) in self.world:
        # Use manhattan distance
        dist_to_mushroom = abs(pos[X] - i) + abs(pos[Y] - j)
        if dist_to_mushroom < dist:
            dist = dist_to_mushroom
            mush_pos = (i, j)

    return mush_pos
```

3.2.3 Entity Position

The world also 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.

To represent the direction that the entity is facing, I implemented the enumeration `Direction` that stores the four values `NORTH, EAST, SOUTH, WEST` along with helper functions `right()` and `left()` to return the direction faced when turning right or left respectively.

I implemented the following methods for representing the entity in the world:

- `place_entity(): void` — Gives the entity a random position in the world and throws `WorldFull` if there is no space for the entity
- `move_entity(Action): void` — Given an action, moves or rotates the entity accordingly
- `get_entity_position(): pos` — Returns the position of the entity
- `entity_facing_out(): bool` — Returns true if the entity is at the edge of the world and facing out
- `get_entity_angle_to_position(pos): float` — Returns the angle from the entity's position and uses the direction it is facing to return the angle to a given position by calling `get_angle`.

When moving the entity, the `within_bounds()` method is used to ensure that the entity does not move out of the world. The `entity_facing_out()` method is used for an optimisation discussed in Section 3.4.6.

3.2.4 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.

After creating a new object, the initialisation method calls the `reset()` method which places mushrooms in the world. This is separated from the initialisation because the world is reset at the start of each of the epochs, removing any currently placed mushrooms.

3.2.5 Testing

Using pytest, I produced a series of unit tests to assert the behaviour of the Environment class. This was done early to ensure that any subsequent refactoring of the code would retain the correct behaviour of the program. Using Continuous Integration, a Travis build ran the test suite after every Git commit and reported if any errors occurred.

Upon discovering bugs during the development process, I produced regression tests. These tests failed according to the bug found and passed once the bug was fixed; ensuring that the bug wouldn't appear later in development.

Unit tests were also produced for the Entities and Simulation module.

3.3 Entities

The Entities module 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. The full UML diagram can be seen in Figure 3.2.

3.3.1 Action

The entity can take four possible actions at each simulation step which are encapsulated in an enumeration; `FORWARDS` to move once cell forwards, `LEFT` to turn 90^{o} left, `RIGHT` to turn 90^{o} right and `NONE` to do nothing.

3.3.2 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. The `eat(int)` method increases this value by 10 if passed

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

an edible mushroom and decreases this value by 11 if passed a poisonous mushroom, according to equation 2.1.

The Entity class also defines the method `behaviour(float, int, float[]): Action, int[]`. This 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, the `behaviour` method always returns `Action.NOTHING` and the vocal signal `[0,0,0]`.

Note that the inputted and outputted utterances have different data types. All utterances are three-bit strings (encoding 8 possible signals) so outputted signals are always mapped to three bits. However, for the population without language, we input a constant signal `[0.5, 0.5, 0.5]`, requiring three floats. In the other two populations, the inputted signals are always comprised of three integer bits.

3.3.3 Rule-based Entity

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.

3.3.4 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 rep-

resentation 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 inputted linguistic signal. Five hidden units are used, as in Cangelosi and Parisi (1998). As discussed in Chapter 2, the paper does not specify the activation functions used in the neural network so I implemented three possible candidates:

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

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). 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 (giving a float between 0 and 1) then rounding the result. The full neural network structure can be seen in Figure 3.3.

The `NeuralEntity` class introduces a few additional methods and some extra state. The `parameters` dictionary stores the weights and biases of the nodes in the neural network; mapping each layer number to two numpy matrices; the weights matrix corresponds to the weights associated with each edge in Figure 3.3 and the biases matrix corresponding to the biases added to the weighted sum at each node. Creating a new `NeuralEntity` calls the `initialise_parameters()` method to set the weights and biases to random values chosen from the rectangular distribution specified by [-1, 1].

The `forward_propagation()` method carries out the forward propagation algorithm. At each layer, the weighted sum is calculated using the dot product of the weight matrix with the output of the previous layer, then the biases matrix is added before an activation function is performed to produce the outputs for that layer. For the final layer, a sigmoid activation is used as the activation function. Finally, the method rounds the outputs of the final layer and returns these as integers (0 or 1).

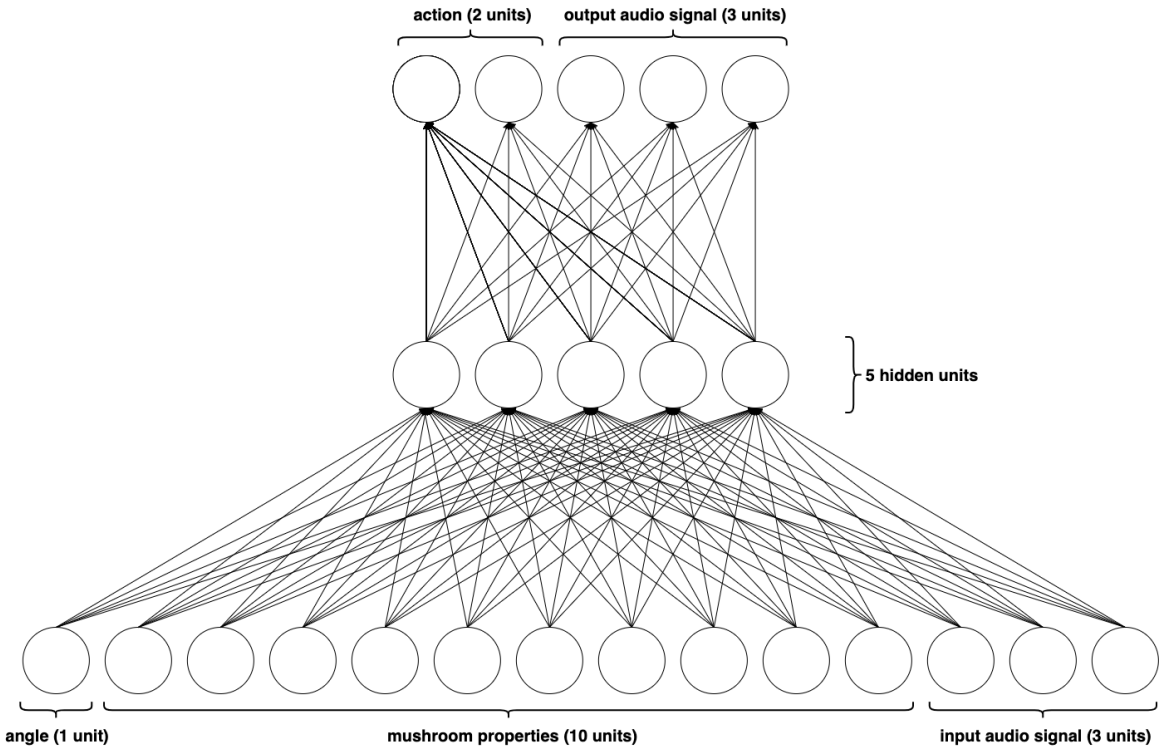

Figure 3.3: Fully-connected ANN to control entity behaviour

This is the pseudocode for my implementation of forward propagation:

```
1.  def forward_propagation(inputs):

2.    # Initialise the cache used to store activations and weighted sums
3.    activations <- []
4.    activations[0] = inputs
5.    final<- number of layers in the neural network

6.    # Calculate weighted sum (Z) and activation (A) at each layer
7.    for layer in (1, final_layer):
8.      Z <- dot(weights[layer], activations[layer-1]) + biases[layer]
9.      activations <- activation_function(Z)

10.   # Calculate activation on the final layer
11.   Z <- dot(weights[final], activations[final-1]) + biases[final]
12.   activations[final] = sigmoid(Z)

13.   # Round final layer to 0 or 1 and return
14.   outputs <- round(activations[final], 0, 1)
15.   return outputs
```

`NeuralEntity` also 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.

3.3.5 Reproduction

An important aspect of these neural entities is their ability to asexually reproduce to produce children. This is a vital part of the genetic algorithm and is implemented as a method `reproduce(int, float): Entity[]` in `NeuralEntity`.

The 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 starts by producing n deep copies of itself. For each of these "children", it iterates through the weights and biases of their neural networks and uses p to decide whether to mutate it. Mutation occurs by adding a random value in the rectangular distribution specified by [-1, 1] to the current value.

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.

3.4.1 Language

The Language enumeration describes the differences in language between the populations as described in Section 2.2.3. `NONE` is used for no language, `EXTERNAL` for the external language and `EVOLVED` for the evolved language. This enumeration is used in the Simulation class and to distinguish between different behaviours.

3.4.2 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 (15 by default), the number of cycles per epoch (50 by default), the population size (100 by default) and the number of generations to run the simulation for (2000 by default). These defaults are set fairly arbitrarily by Cangelosi and Parisi (1998) and were discussed 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.

3.4.3 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 10 to 14). The outputs of the `behaviour()` method are interpreted accordingly, moving the entity in the environment (line 15). 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 17 to 19).

```
1. def run_single(entity):

2.   # Create a random environment
3.   env <- new Environment

4.   for epoch in num_epochs:

5.     # Reset mushroom positions and entity position between each epoch
6.     env.reset()
7.     env.place_entity()

8.     for cycle in num_cycles:
9.       # Gather inputs for the entity
10.      location <- angle from entity to closest mushroom
11.      perception <- bit string of mushroom if entity adjacent to it
12.      listening <- input signal according to language type

13.      # Get behaviour of the entity and move accordingly
14.      action <- entity.behaviour(location, perception, listening)
15.      env.move_entity(action)

16.      # Entity eats a mushroom if on top of one
17.      if env.entity_position is a mushroom:
18.        entity.eat(mushroom)
19.        env.remove(mushroom)
```

Line 12 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.

3.4.4 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 3-6 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 3.4.3.

Lines 7-9 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 10-14 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.

```
1.  def run_population(entities):

2.    for generation in num_generations:

3.      # Step 1: Run the simulation for each entity
4.      # Pass the remaining population for the evolved language
4.      for entity in entities:
5.         population <- entities - {entity}
6.         run_single(entity, population)

7.      # Step 2: Sort the entities by their fitness and select the best
8.      sort(entities, entity.fitness)
9.      best_entities <- top percentage_keep of entities

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

Language Type	Average Runtime
NONE	53:20
EXTERNAL	51:15
EVOLVED	1:36:00

Table 3.2: Run times for the genetic algorithm according to language type, averaged across ten runs

```
14.     population <- children
```

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 3.4.2 above.

3.4.5 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 SLURM. 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).

Thus I could quickly make changes to my program, sync my code with my HPC environment using git then schedule a series of jobs using SLURM. These jobs would then eventually be scheduled, would run on different nodes in the HPC and write data to files in my login space. This data could be copied to my local computer using the `rsync` command. 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 specified in Tables 2.1 and 3.3. Examples of running my system from the command line can be seen in Appendix A.

3.4.6 Optimisations

Although simulation speed was not the primary goal for this project, it was still useful to consider some optimisations in order to increase the rate at which I could run experiments.

Considering a run of the genetic algorithm gives 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 involves 15 epochs of 50 cycles, or 750 iterations of the perception-action loop described in Section 3.4.3. 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). By examining the difference in runtime between the **no language** and **evolved language** populations, we can estimate that calls to the the `behaviour()` method are responsible for approximately 80% of the runtime for when the language type is `NONE` and 89% when the language type is `EVOLVED`. This is calculated by comparing the average runtime of ten runs of the genetic algorithm for 1000 simulations for each language type, seen in Table 3.2. My optimisations were thus focused on making this call efficient and reducing the number of times it was called.

The first optimisation was to use numpy arrays to represent the neural network weights and to implement the matrix multiplications in the forward propagation algorithm.

The second optimisation was made by taking advantage of the fact that the 100 calls to `run_single()` at each generation can be made independently. Using the `multiprocessing` library for Python, 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 processes available. Using the `Pool`, I call the `starmap` method to dynamically offload the 100 calls to `run_single()` to these processes. This provides a significant speedup for my simulations depending on the number of processes available.

Thirdly, noting that (1) the `behaviour()` method is deterministic and (2) that 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. Similarly, if the entity is at the edge of the world and attempts to move `FORWARD`, it will not move and will return the same response in the next cycle so we can again skip the epoch. Note that we cannot skip the entire simulation because at the end of each epoch the position of the mushrooms and the entity are reset, which may change the inputs to the function.

More complicated optimisations of this form could be considered, such as detecting looping behaviours (spinning forever, moving backwards and forwards forever and so on). However, detecting these would increase the time and memory requirements for the program and would not yield a huge increase in performance for epochs of only 50 cycles.

3.4.7 Interactivity

In order to be able to watch the simulation and genetic algorithm occur live, I implemented interactive behaviour into the Simulation class. One of the parameters set by the `set_io_options()` method is the `interactive` property, set to False by default, which allows this behaviour to occur.

An `initialise_io()` method is called at the start of the genetic algorithm. This produces an average fitness graph which is updated continuously with the latest average fitness of the population through a call to an `io()` function in the primary loop of the genetic algorithm. Running the simulation in this interactive mode also displays useful information at each generation and

I/O Parameter	Description	Default Value
interactive	Sets interactive behaviour on	False
record_language	Whether to save the language	True
record_language_period	How often the language is saved	1
record_entities	Whether to save the population of entities	True
record_entities_period	How often the population is saved	1
record_fitness	Whether to save the average fitness	True
foldername	Where the data is saved in the local filesystem	"folder"

Table 3.3: Parameters used for controlling the Input and Output of the system

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 A.

3.4.8 Data Recording

For the analysis of the simulation, I needed a few utility methods to collect data as it ran. The `initialise_io` method is used to set a group of simulation parameters concerning how data is saved throughout the running of the simulation. These parameters are detailed in Table 3.3.

When the `io()` method is called, these parameters are used to control how often and whether data is recorded. The main information recorded is just the average fitness at each generation, used to produce the fitness graphs. This is done just by appending the current average fitness of the population to a file.

Language is recorded by performing a 'naming task', as was described in Section 2.3.2. This is implemented by calling the `naming_task()` method for each entity in the population, which calls `behaviour()` for 80 different possible inputs and recording the outputted signals. These signals are then written to a file by the `save_language()` method which is called by `io()`.

The entities are saved by calling the `save_entities()` method in `io()`. This simply uses the `pickle` library to save the list of entities to a binary file; easily loaded again with the `load_entities()` method.

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 above in Section 3.4.7.

3.5.1 Fitness Graphs

The fitness graphs were plotted by reading the average fitness values recorded from the simulation, plotting fitness over generation number. Typically I ran 10 simulations for each language

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 3.4: 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.

type at once: another method plotted the average of ten of these fitness graphs.

3.5.2 Language Frequency Distributions

Similarly to the fitness graphs, I implemented a method for plotting the frequency distribution of the language. It plots a bar graph with frequency on the y-axis and each of the eight possible signals on the x-axis. The signals used for edible and poisonous mushrooms were separated by colour and the method allowed for the frequency distribution of several different generations to be compared by stacking them above each other; this allows for observing the convergence of signals to a productive language.

3.5.3 QI Score

I then implemented methods for calculating the QI score of the language produced by the population as given in equation 2.4 in Section 2.3.2. 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 3.4.

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.

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:

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

As well as having logical justification, the preliminary results in Table 3.4 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 equation that Cangelosi and Parisi gave. However, having achieved similar correlations to them using my revised equation, I have also

assumed that their mistake was just in the paper and that the equation they implemented was the same as equation 3.1. I will further discuss this correlation in the next chapter.

3.6 Summary

In this chapter I covered my implementation of the "mushroom world" simulation. I detailed the Environment, Entity and Simulation classes and gave details of how I implemented the genetic algorithm and the optimisation choices I made. I discussed my testing framework, how I made the simulation interactive and how I recorded data for analysing the simulation.

Chapter 4

Evaluation

4.1 Overview

In Section 4.2.1, I begin by examining the average fitness of different populations with default simulation parameters. In Section 4.2.2 I then examine the language produced by the **evolved language** population and investigate the quality index (QI) of this language as defined in Section 2.3.2.

In Section 4.3, 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 4.4, and discuss whether the conclusions of Cangelosi and Parisi (1998) still hold.

4.2 Initial Analysis

4.2.1 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 try to find 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[1].

[1] I was also able to run these experiments due to my access to the HPC facility; computing power that Cangelosi and Parisi (1998) may not have had.

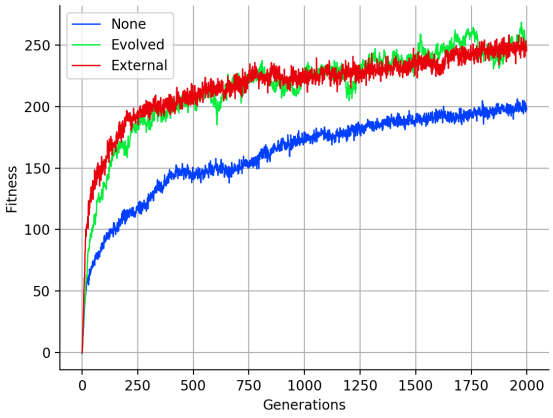

Figure 4.1: Average fitness of each population, averaged over 10 replications, with a **ReLU** hidden layer activation.

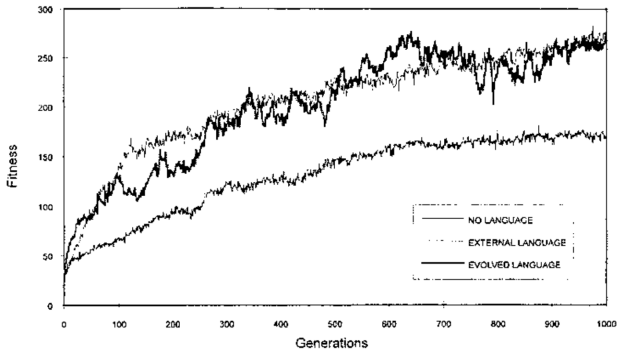

Figure 4.2: Average fitness of each population, averaged over 5 replications. Source: Cangelosi and Parisi (1998).

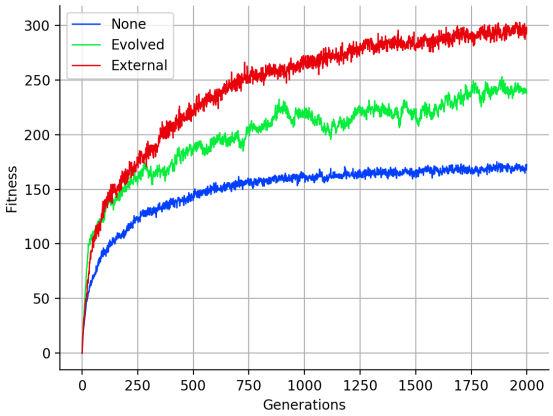

Figure 4.3: Average fitness of each population, averaged over 10 replications, with an **identity** hidden layer activation.

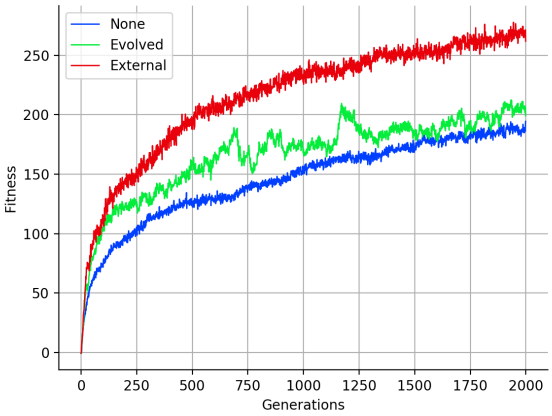

Figure 4.4: Average fitness of each population, averaged over 10 replications, with a **sigmoid** hidden layer activation.

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.

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 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 4.4, I will further explore robustness to other changes in simulation parameters.

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 2.3.2.

4.2.2 Language Analysis

In Section 2.3.2, 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 replicas from the experiment. The frequency distribution is calculated using the 'naming task' described in Section 2.3.2. 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

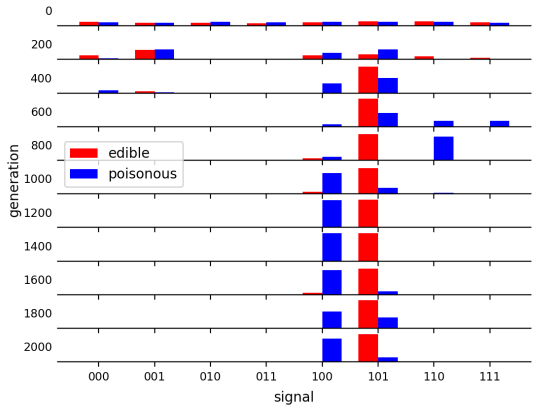

Figure 4.5: Frequency distribution of the possible signals produced by all individuals in 10 generations in one replica of a simulation with an **evolved language.**

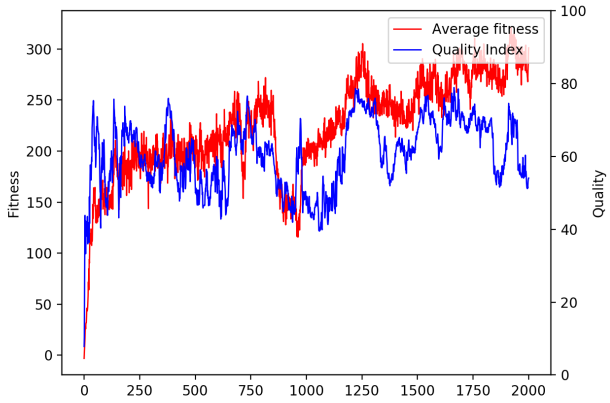

Figure 4.6: Average fitness for one replica of a population with an **evolved language**. Also shown is the Quality Index score of the language produced by this population.

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 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 correlation between fitness and quality was 0.568, calculated using Pearson r. 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.

4.3 Behavioural Analysis

In Section 4.2.1 I presented the average fitness of each population averaged 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 under-

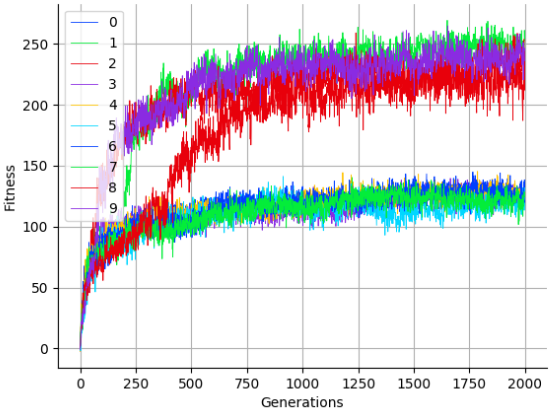

Figure 4.7: Average fitness of 10 replications of the **no language** population, with an **identity** hidden layer activation.

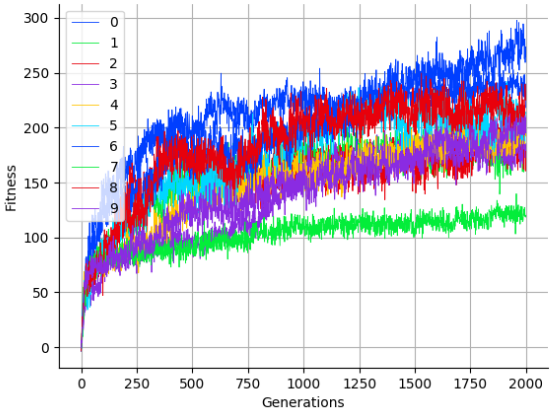

Figure 4.8: Average fitness of 10 replications of the **no language** population, with a **ReLU** hidden layer activation.

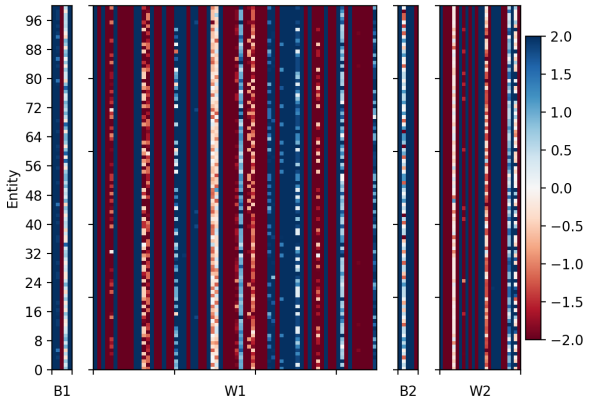

Figure 4.9: Heatmap of the weights and biases of 100 entities taken from generation 2000.

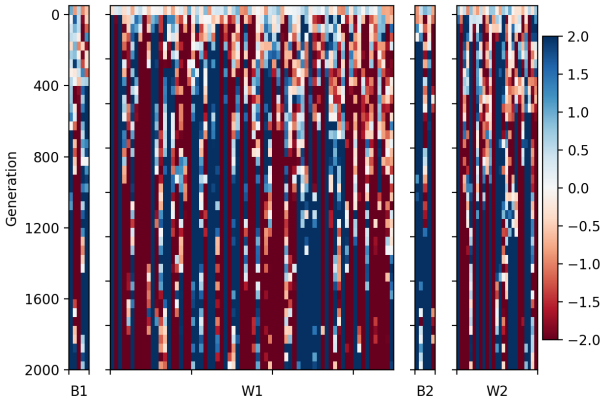

Figure 4.10: Heatmap of a randomly chosen entity taken from increasing generations.

standing. 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 begin 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 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:

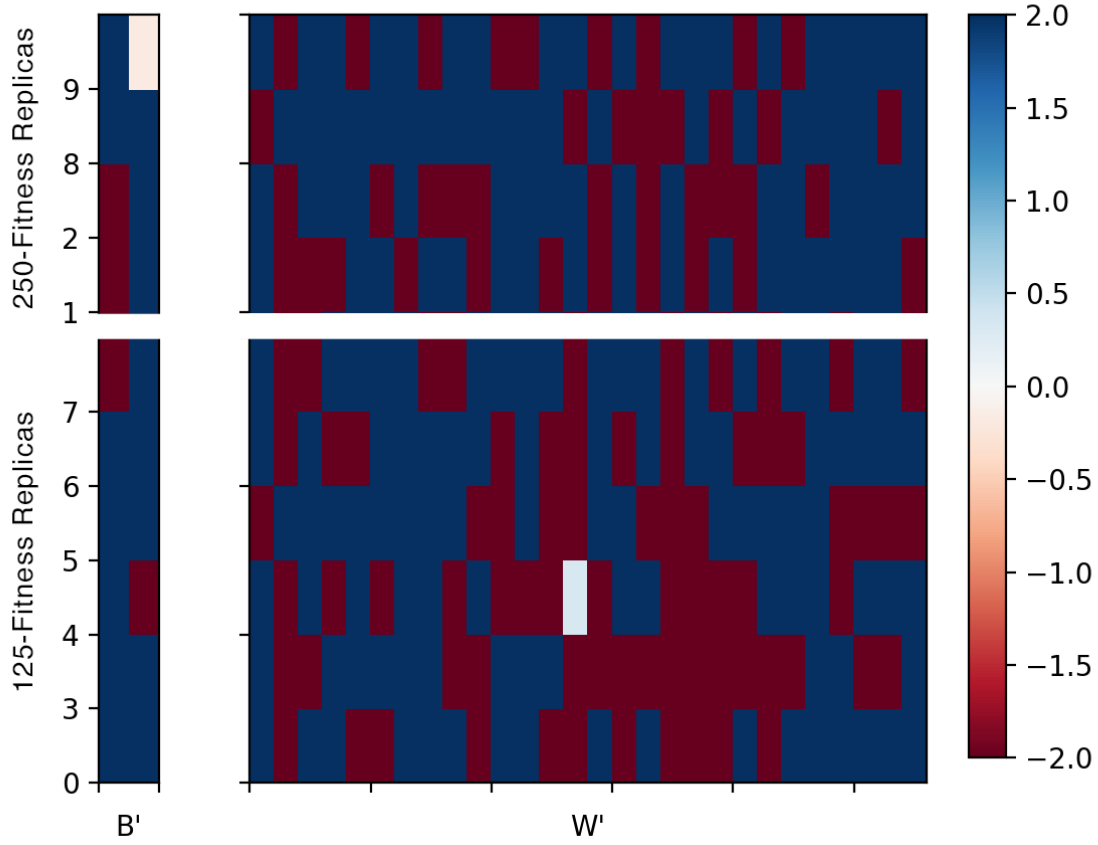

Figure 4.11: Heatmaps of the flattened neural networks of a randomly chosen entity from generation 2000 of ten replicas.

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

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. Taking a random entity from generation 2000 of each population, I placed a poisonous mushroom two cells in front of them. 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. These are seen in Figure 4.12.

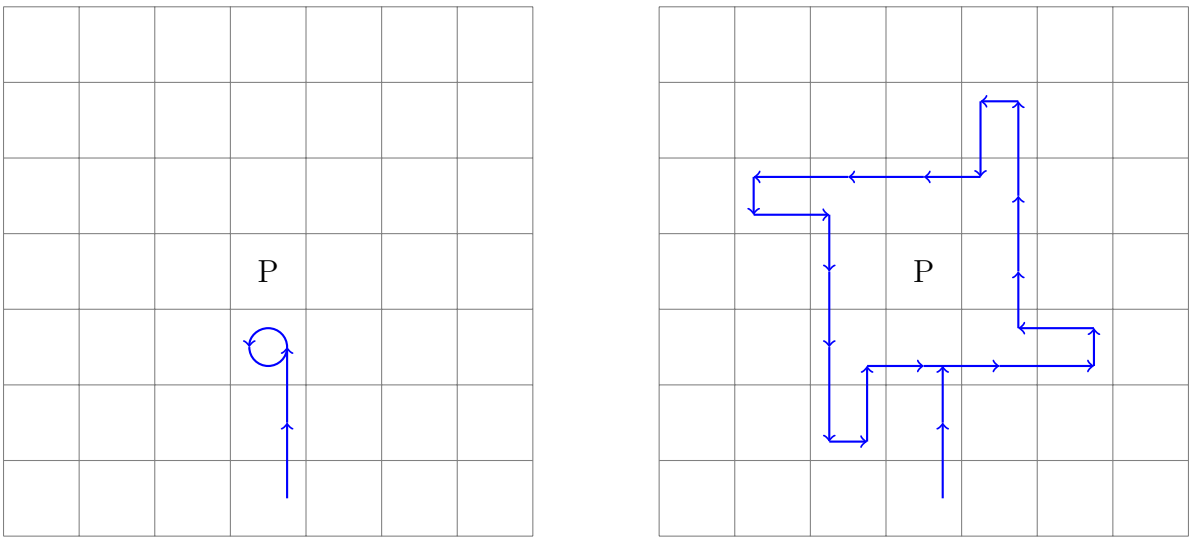

Figure 4.12: The behaviour of two entities approaching a poisonous mushroom (P). The entities are randomly chosen from generation 2000 of two **no language** populations with an average fitness of 140 (left) and 250 (right).

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.

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.

4.4 Exploring Simulation Parameters

In Section 4.2.1 I demonstrated that the simulation was robust to which activation function was used in the hidden layer of the neural networks. In all cases, the populations with language

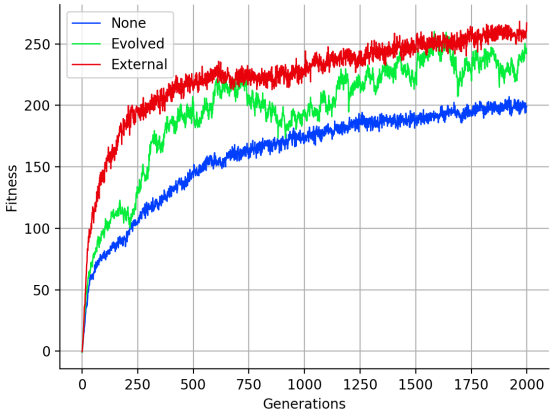

Figure 4.13: Average fitness of each population, averaged over 10 replications, with two hidden layers of five units each.

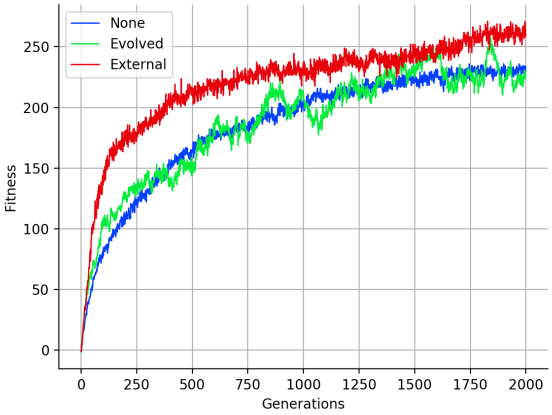

Figure 4.14: Average fitness of each population, averaged over 10 replications, with three hidden layers of ten units, five units and ten units.

performed better than those without. In this section, I will explore the effect of changing neural network depth, adjusting the length of each epoch and altering the genetic algorithm parameters.

4.4.1 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 4.3, 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.

4.4.2 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

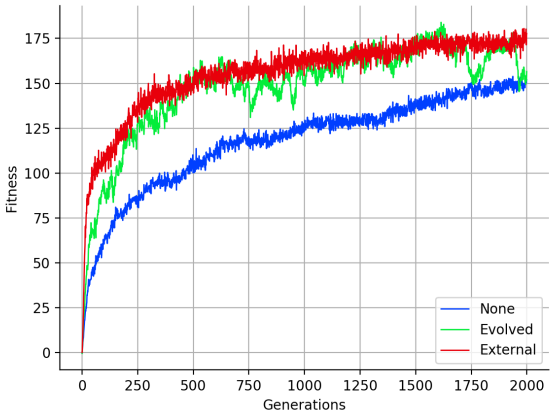

Figure 4.15: Average fitness of each population, averaged over 10 replications, with simulations consisting of 30 epochs of length 15.

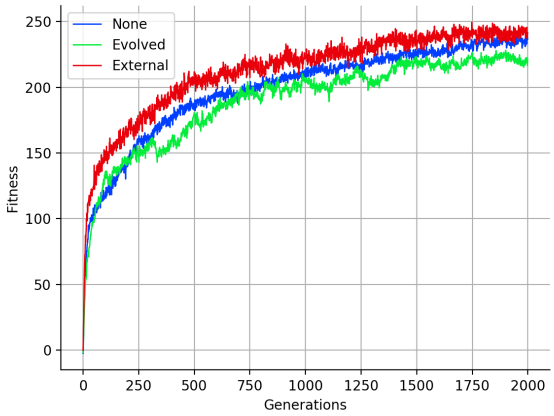

Figure 4.16: Average fitness of each population, averaged over 10 replications, with simulations consisting of 10 epochs of length 75.

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.

I first experimented with 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 this prevents the entities from reaching more than one edible mushroom in each cycle. 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.

4.4.3 Genetic Parameters

Following the behavioural analysis in Section 4.3, 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 have an effect on the population without language; resulting in the **evolved language** population performing worse than the population with **no language**. In my opinion, this is because lowering this parameter prevents the emergence of new behaviours

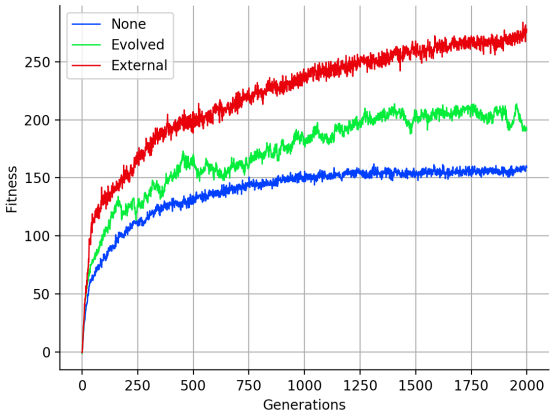

Figure 4.17: Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 20%.

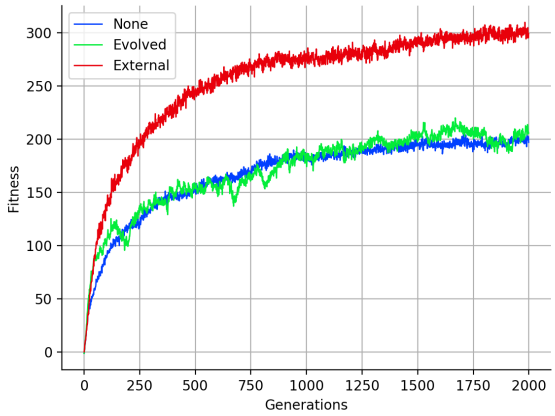

Figure 4.18: Average fitness of each population, averaged over 10 replications, with the mutation percentage set to 50%.

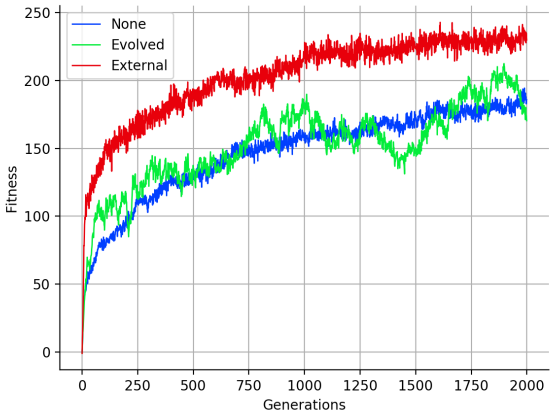

Figure 4.19: Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 10%.

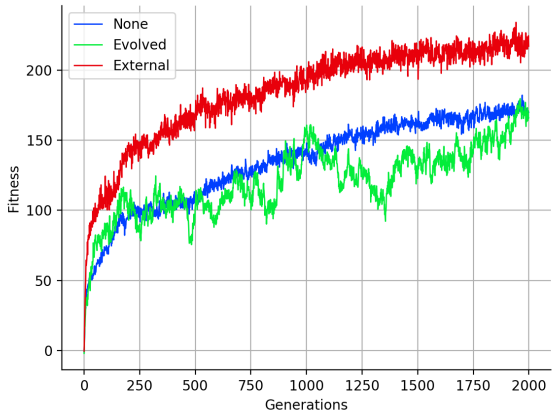

Figure 4.20: Average fitness of each population, averaged over 10 replications, with the percentage of entities selected to reproduce set to 5%.

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

4.5 Summary

In this chapter I analysed and evaluated my implementation of the "mushroom world" simulation. I performed initial analysis of the average fitness achieved by the three populations and analysed the language produced by the **evolved language** populations, mirroring the analysis made by Cangelosi and Parisi (1998). I then conducted my own, deeper analysis of the simulation, exploring different behaviours exhibited by the **no language** population and finally exploring robustness to changes made to the simulation parameters.

Chapter 5

Conclusion

5.1 Achievements

This project was a success. All points listed in the success criteria and all items in the requirements analysis (Section 2.4) 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. 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.

5.2 Lessons Learnt

The main issue I faced was the emergence of strange behaviours at the population level of the simulation due to small underlying bugs in the core implementation of the entities and the environment objects. This was despite having implemented as thorough a suite of unit tests as possible; the problem was unforeseen. Once discovered, I was able to create regression tests to prevent their re-emergence.

Another useful change was the implementation of a command-line interface for my system. Earlier in development, I created hard-coded methods for each experiment that required code changes between each run on the HPC. This slowed the process 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.

Finally, during the course of the project I found that Python's dynamic typing introduced very subtle bugs that could not be discovered before compilation. This led me to conclude that Python would not be suitable if porting this as 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 duck typing was actually useful in many occasions.

5.3 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. 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.