Zébulon Goriely

Simulating Language Learning and Evolution

Computer Science Tripos – Part II

Queens' College

February 24, 2020

Proforma

Name:	**Zébulon Goriely**
College:	**Queens' College**
Project Title:	**Simulating Language Learning and Evolution**
Examination:	**Computer Science Tripos – Part II, May 2020**
Word Count:	**NULL**[1]
Project Originator:	Zébulon Goriely
Supervisor:	Prof. Paula Buttery and Dr. Andrew Caines

Original Aims of the Project

Language has evolved and therefore probably gave an evolutionary advantage to the individuals that exhibited it. According to Cangelosi and Parisi (1998), it is difficult to investigate the evolutionary origin of language and the selective pressures that may have originated language due to the limited evidence available. They propose using computer simulations of evolutionary scenarios to investigate this. In the paper referenced, they describe a simulated toy world where agents controlled by neural networks interact with an environment of mushrooms that are edible and poisonous. Exploring this simulation is the basis of my project.

Work Completed

I implemented the mushroom-world simulation and explored three populations of feed-forward neural networks; one without language, one with an externally imposed language and one with an evolved language. To simulate evolution, I implemented a genetic algorithm to allow the fittest members of each generation to reproduce. The state of the simulation was saved at each generation in order plot the fitness of the three populations, plot the quality of the language produced and investigate behavioural tests. To conclude the project, I compared these findings with Cangelosi and Parisi (1998).

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

Special Difficulties

None.

Declaration

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

Signed Zébulon Goriely

Date [date]

Contents

1 Introduction **13**

1.1 Motivations 13

1.2 Prior Work 14

1.3 Project Overview 14

2 Preparation **17**

2.1 Mushroom World 17

2.2 Entities 18

2.2.1 Feed-Forward Neural Networks 19

2.2.2 Alternative Structures 19

2.2.3 Genetic Algorithm 20

2.2.4 Population types 21

2.3 Simulation Analysis 22

2.3.1 Generational Fitness 22

2.3.2 Efficiency of Language 23

2.4 Requirements Analysis 24

2.5 Starting Point 25

2.6 Software Engineering 25

2.6.1 Languages and Libraries 25

2.6.2 Project Management 25

2.6.3 Version Control 26

2.6.4 Development tools 26

2.6.5 Development environment 26

2.6.6 Code License 27

2.7 Summary 27

3 Implementation **29**

3.1 High-level Overview 29

3.2 Environment 29

3.2.1 Mushrooms 30

3.2.2 Representation of the World 31

3.2.3 Entity Position 32

3.2.4 World Initialisation 33

3.2.5 Testing 33

3.3 Entities 33

3.3.1 Action 33

3.3.2 Abstract Specification 34

3.3.3 Rule-based Entity 34

3.3.4 Neural Network Entity 35

3.3.5 Reproduction 37

3.3.6 Testing 37

3.4 Simulation 37

3.4.1 Language 37

3.4.2 Initialising the Simulation 38

3.4.3 Single-Entity Simulation Run 38

3.4.4 Genetic Algorithm 39

3.4.5 Running the simulations using HPC 40

3.4.6 Optimisations 41

3.4.7 Interactivity 42

3.4.8 Data Recording 43

3.5 Simulation Analysis 43

3.5.1 Fitness Graphs 44

3.5.2 Language Frequency Distributions 44

3.5.3 QI Score 44

4 Evaluation **45**

5 Conclusion **47**

Bibliography **47**

A Project Proposal **51**

List of Figures

2.1 An organism (▲) in the simulation environment with edible (E) and poisonous (P) mushrooms. 18

2.2 Structure of a fully-connected ANN to control entity behaviour 20

2.3 Differences in simulation structure for the three population types 22

3.1 Core modules for my project 30

3.2 UML Diagram for the Simulating Module 30

3.3 Fully-connected ANN to control entity behaviour 36

Acknowledgements

Paula et al.

Chapter 1

Introduction

Introduction here, start again as if abstract does not exist. Include your contributions in bullet point format. End with paragraph describing structure of the rest of the paper

1.1 Motivations

Language has evolved and therefore probably gave an evolutionary advantage to the individuals that exhibited it. If we go back enough generations, we will find human ancestors that did not possess language. The field of language evolution is diverse and spans many scientific disciplines, from evolutionary biology and neurosciences 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. Language evolution is a once-occurring, long-term and complex system making it very difficult to study, but computer simulations may provide a means by which to examine the theories surrounding it.

Computer simulations are *"theories of the empirical phenomena that are simulated"* (Cangelosi and Parisi, 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 can not be predicted even with complete knowledge of the system. Language is a complex system and it has been shown that the bottom-up approach of simulations can generate key insights (Langton, 1997).

- Main question - how can language evolve when it has a purely informative function and so advantageous to receiver but not to the producer?

- Reference Clark (1993) for what an efficient language is

1.2 Prior Work

State what similar work has been done before and since what I am trying to replicate. Also a continuation of the motivation. Why does this paper in particular interest you. Focus more on their prior work. Talk about work that has continued.'

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) introduces 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. 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).

1.3 Project Overview

In my project I re-implement the "toy mushroom world" simulation described in 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 the following four chapters of this dissertation I will cover all preparations made (Chapter 2) before proceeding to cover my implementation of the simulation (Chapter 3) after which I evaluate my project (Chapter 4) before presenting my closing remarks (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 test these changes. This "mushroom world" was described by Cangelosi and Parisi (1998), inspired by the use of signals to communicate information about food location and quality present in may species.

The organisms will 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 poisonous and 10 of which are poisonous. They will be able to explore this world in 15 epochs of 50 simulation cycles each; in each new epoch the world is 'reset', the position of the entity reset 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 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

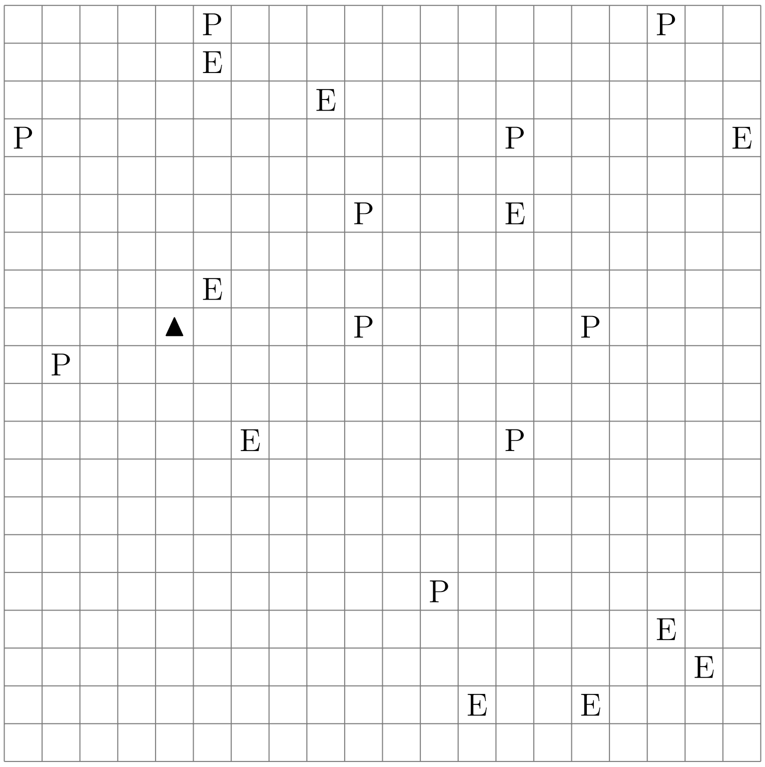

Figure 2.1: An organism (▲) in the simulation environment with edible (E) and poisonous (P) mushrooms.

of the environment ensures that the organism will likely be able to reach at least one edible mushroom but will unlikely be able to eat all edible mushrooms in time. This encourages productive strategies to search for edible mushrooms within the limited time available. 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.

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

The action is composed of two 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 some '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 explore adding 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. From this abstract description of entities, I can consider how this system may be designed.

2.2.1 Feed-Forward Neural Networks

Inspired by the biological neural networks that constitute animal brains, *artificial neural networks* are an appropriate choice for modelling the behaviour of these 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 does not describe which activation function was used in his 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 (INCLUDE REASON WHY HERE). 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, producing different behaviours.

2.2.2 Alternative Structures

- talk about linear regression

[1]ReLU stands for Rectified Linear Unit

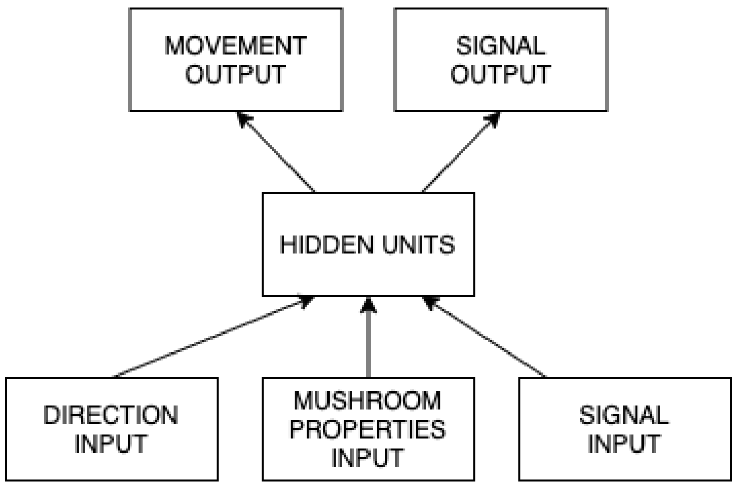

Figure 2.2: Structure of a fully-connected ANN to control entity behaviour

- talk about svms

- talk about more complex neural networks

2.2.3 Genetic Algorithm

Similarly to how an artificial neural network is a natural implementation for simulating the behaviour of an entity, the *genetic algorithm* is a natural implementation for simulating the population dynamics of a population of such entities. Just as artificial neural networks are inspired by animal brains, the genetic algorithm is inspired by the process of natural selection which is 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 a population of randomly generated individuals and proceeds iteratively. For each *generation*, the *fitness* of each individual is calculated. The more fit 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. These introduce two new constants, as described in Table 2.1.

2.2.4 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
- External Language
- Evolved Language

The population with no language acts as a baseline. The audio input to the neural network controlling the entity is set to a constant and the audio 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 an externally provided 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 audio 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 primary entity with another randomly chosen entity at each simulation cycle. This second entity will be given same inputs as the primary entity as well as the properties of the closest mushroom, no matter the distance. This entity labels the mushroom for the primary entity. The audio output of this second entity will be used as the audio input to the primary entity; simulating a one-word utterance.

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

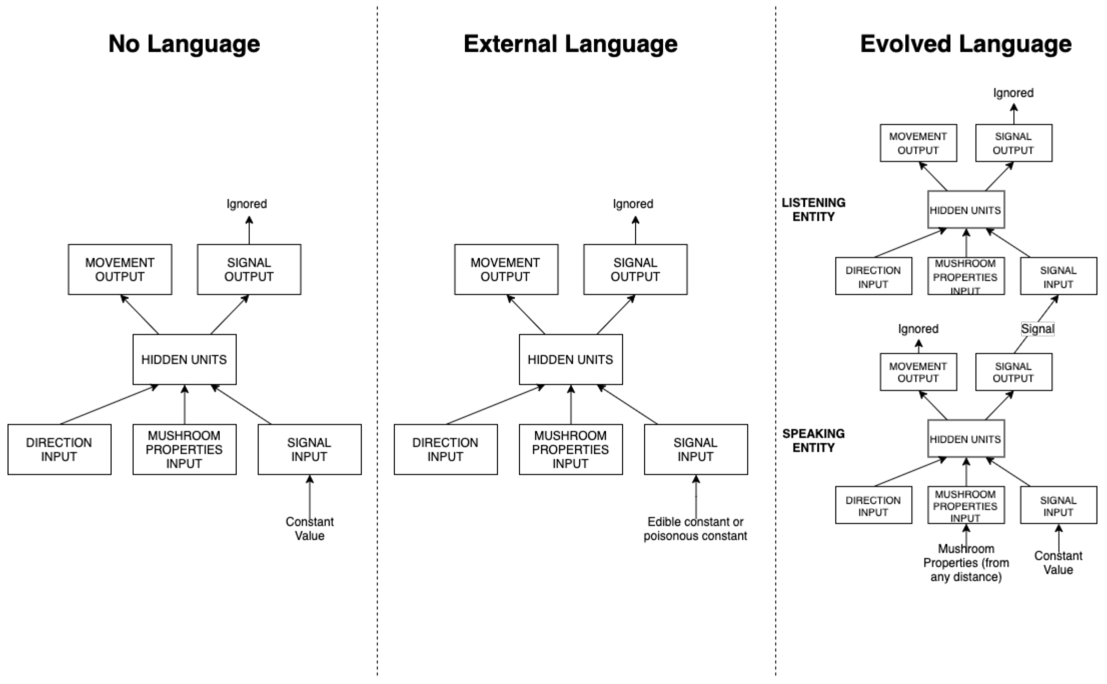

Figure 2.3: Differences in simulation structure for the three population types

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	1000
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 2.2.3, the fitness score for each entity describes its success in distinguishing between edible and poisonous mushrooms and 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 2.2.4. 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. To investigate the language produced by different populations, Cangelosi 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 adding(?) this to 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. In particular, it will be interesting to see if a correlation exists in the No Language population, despite not using any form of communication, as this would imply the parallel development of cognitive ability with linguistic ability.

LOOK INTO THIS EQUATION - A PLUS MAKES A LOT MORE SENSE

2.4 Requirements Analysis

My project involves reimplementing 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 sections; implementing the simulation and constructing the means of analysis 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 feedforward neural networks; with the structure described in section 2.2.1.

3. Have a genetic algorithm that runs after all agents complete the simulation, as described in section 2.2.3.

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 2.2.4.

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 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 preporatory 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 form 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 dates of the project.

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

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

tasks were compiled into a Kanban board, with sections for Todo, 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 completed 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 was met very early on in the timeline, allowing me to focus on refining the project and work on my extensions.

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 service for running the simulations, as discussed below. In addition, I also performed 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].
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.

[5]https://github.com/
[6]https://travis-ci.org/
[7]https://www.pylint.org/
[8]http://google.github.io/styleguide/pyguide.html
[9]https://github.com/google/yapf
[10]https://code.visualstudio.com/

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 filesystem over 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 interesting 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, I will be able to investigate the parallel development of the ability of the entities to categorise mushrooms and their ability to name them.

Chapter 3

Implementation

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. Finally, the Experiments module contains a series of functions used for quickly running different variants of the simulation.

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 subclasses 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 (2.2.3) and the different populations (2.2.4).

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.

3.2 Environment

The Environment modules 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.

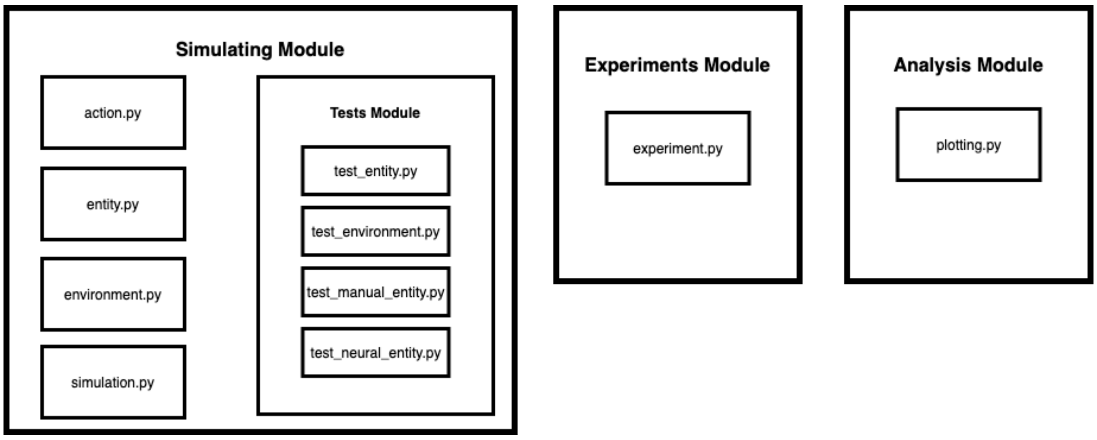

Figure 3.1: Core modules for my project

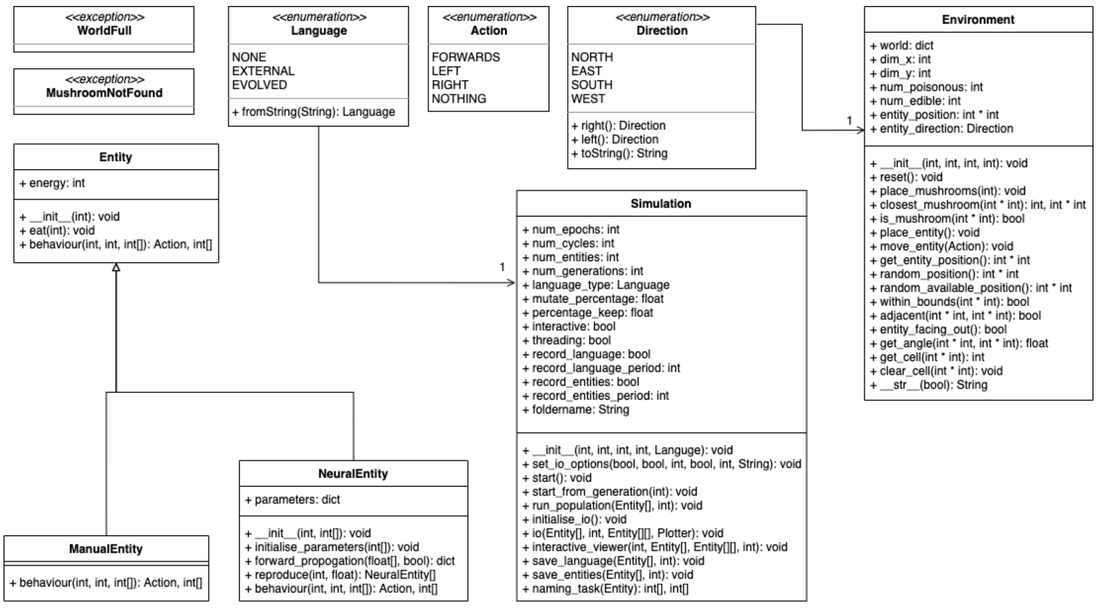
<<exception>>
WorldFull
<<exception>>
MushroomNotFound
<<enumeration>>
Language
NONE
EXTERNAL
EVOLVED
+ fromString(String): Language
<<enumeration>>
Action
FORWARDS
LEFT
RIGHT
NOTHING
<<enumeration>>
Direction
NORTH
EAST
SOUTH
WEST
+ right(): Direction
+ left(): Direction
+ toString(): String
Environment
+ world: dict
+ dim_x: int
+ dim_y: int
+ num_poisonous: int
+ num_edible: int
+ entity_position: int * int
+ entity_direction: Direction
+ __init__(int, int, int, int): void
+ reset(): void
+ place_mushrooms(int): void
+ closest_mushroom(int * int): int, int * int
+ is_mushroom(int * int): bool
+ place_entity(): void
+ move_entity(Action): void
+ get_entity_position(): int * int
+ random_position(): int * int
+ random_available_position(): int * int
+ within_bounds(int * int): bool
+ adjacent(int * int, int * int): bool
+ entity_facing_out(): bool
+ get_angle(int * int, int * int): float
+ get_cell(int * int): int
+ clear_cell(int * int): void
+ __str__(bool): String
1
Entity
+ energy: int
+ __init__(int): void
+ eat(int): void
+ behaviour(int, int, int[]): Action, int[]
Simulation
+ num_epochs: int
+ num_cycles: int
+ num_entities: int
+ num_generations: int
+ language_type: Language
+ mutate_percentage: float
+ percentage_keep: float
+ interactive: bool
+ threading: bool
+ record_language: bool
+ record_language_period: int
+ record_entities: bool
+ record_entities_period: int
+ foldername: String
+ __init__(int, int, int, int, Languge): void
+ set_io_options(bool, bool, int, bool, int, String): void
+ start(): void
+ start_from_generation(int): void
+ run_population(Entity[], int): void
+ initialise_io(): void
+ io(Entity[], int, Entity[][], Plotter): void
+ interactive_viewer(int, Entity[], Entity[][], int): void
+ save_language(Entity[], int): void
+ save_entities(Entity[], int): void
+ naming_task(Entity): int[], int[]
1
NeuralEntity
+ parameters: dict
+ __init__(int, int[]): void
+ initialise_parameters(int[]): void
+ forward_propogation(float[], bool): dict
+ reproduce(int, float): NeuralEntity[]
+ behaviour(int, int, int[]): Action, int[]
ManualEntity
+ behaviour(int, int, int[]): Action, int[]

Figure 3.2: UML Diagram for the Simulating Module

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 edible and poisonous mushrooms may share two or zero bits in the same position.

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 at most 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 runtime 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 the sake of 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
- `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 turning 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 in the world

- `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 that an entity experiences, 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. This was ensured using Continuous Integration; after every Git commit a Travis Build would runt he test suite and report if any errors occurred.

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 90º left, `RIGHT` to turn 90º right and `NONE` to do nothing.

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 audio signal (heard)	Input
vocal	`int[]`	A 3-bit audio signal (spoken)	Output
action	`Action`	The action taken by the Entity	Output

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

3.3.2 Abstract Specification

The Entity class acts as an empty parent class. As state, it has an `energy` value which is used to determine the fitness of the entity. The `eat(int)` method increases this value by 10 if passed an edible mushroom and decreases this value by 11 if passes 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 audio signals have different datatypes. All audio signals 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 audio signal `[0.5, 0.5, 0.5]`, requiring three floats. In the other two populations, the inputted audio signals are always comprised of three 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

Once I had a functional simulation using rule-based entities, I implemented the feed-forward neural network used to control entities and allow their behaviour to evolve over many generations. In section 2.2.1 I gave the general structure for the fully-connected neural network I wanted to use to control the behaviour of the entities. Now that I have defined the datatypes for these inputs and outputs, I can produce a more concrete description of this neural network.

The neural network has fourteen input units. The first unit is the float value `location` which describes the angle to the closest mushroom. The next ten units describe the bit-string representation of the mushroom the adjacent to the entity. Note that all 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 audio 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). The paper 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). This is the pseucodode for my implementation of forward propagation:

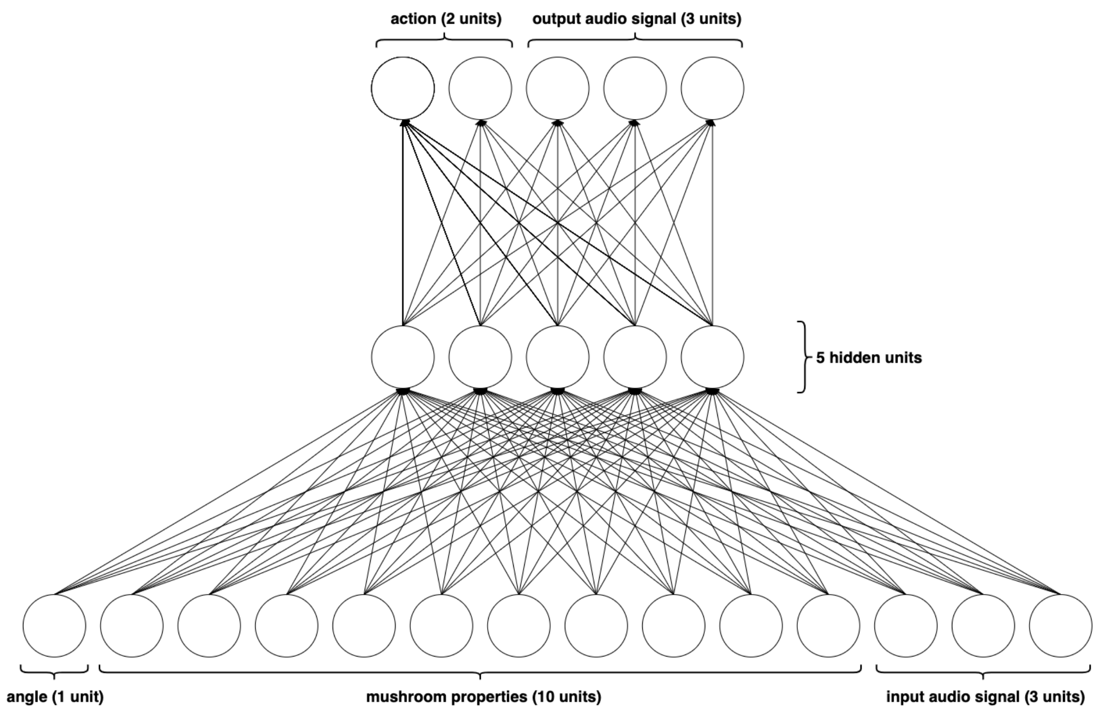
action (2 units)
output audio signal (3 units)
5 hidden units
angle (1 unit)
mushroom properties (10 units)
input audio signal (3 units)

Figure 3.3: Fully-connected ANN to control entity behaviour

```
def forward_propagation(inputs):

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

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

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

  # Round final layer to 0 or 1 and return
  outputs <- round(activations[final], 0, 1)
  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 **??**, 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.

3.3.6 Testing

Similarly to with the Environment class, I used pytest to produced a series of unit tests to assert the behaviour of the Entity class and subclasses. Tests were created to assert the behaviour of every method implemented, such as ensuring that the pointers in the children entities generated by the `reproduce()` method are distinct from the pointers in the parent entity.

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 2.2.4. `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 (100 by default). These defaults are set fairly arbitrarily by Cangelosi and Parisi (1998) and were discussed section 2.1.

Further specified are two constants 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 2.2.3 and are listed with the other arbitrary constants 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 <- audio 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.

```
def run_population(entities):

  for generation in num_generations:

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

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

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

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 scheduling 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 then 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 computations.

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

3.4.6 Optimisations

Although simulation speed is not the primary goal for this project (if it was I would not have used Python), it was still useful to consider some optimisations in order to increase the rate at which I could run experiments.

To find areas of the code to optimise, I analysed which method were called the most often using some basic analysis on the loop structures. If we consider a run of the genetic algorithm operating on a population of 100 entities for 1000 generations, we can already see that the `run_single()` method is called 100000 times. If we consider that each single simulation is made up of 15 epochs of 50 cycles each then the perception-action loop described in section 3.4.3 occurs 750 times for each of these 100000 calls. This gives us a total of 75 million repeats of any methods called in this perception-action loop.

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 only getting and setting certain properties of the environment and entity. Furthermore, the `behaviour()` method is called twice for the population with the evolved language as it is used for the 'naming' task to produce the input signal for the entity. By examining the difference in runtime between the No Language population and the Evolved Language population, 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 reducing the number of times this method was called and increasing the efficiency of the method for when 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 first 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 our simulation depending on the number of processes available.

Secondly, I optimised the texttrun_single() method by reducing the number of simulation loops required in the early stages of the genetic algorithm. A simple optimisation could be made by noticing that the `behaviour()` method is entirely deterministic and that the environment that the entity interacts with is static. Thus, if the method returns the `NOTHING` action then the entity will not move, the inputs to `behaviour()` will not change and the method will continue to return `NOTHING` for the rest of the epoch. Thus, we can simply break from the current epoch and skip to the next one. 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 of 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 probably 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, False by default which allows this behaviour to occur.

An `initialise_io` method is called at the start of the genetic algorithm to produce a live fitness over time graph for the genetic algorithm. In the primary loop for the genetic algorithm, an `io()` function is called to update this graph with the latest average fitness for the population. This allows the fitness of the population to be tracked live. An example of this is seen in Figure **??**.

The `io()` function also outputs other useful information and pauses the genetic algorithm to allow for interactivity. At each generation, the current generation, current list of fitness values and average energy for the current population is printed. The program then waits for user input before continuing. An inputted number will run the genetic algorithm for that many generations. Alternatively, by entering "watch" followed by a number, the program will allow the user to watch the behaviour of the entity indexed by the inputted number within the simulation. This is done by calling `run_single()` again on the selected entity with an additional parameter, `viewer`, set to True.

The `run_single()` method also has interactive behaviour when `viewer` is set to True. At each simulation step, the current world is displayed as a grid with emojis used to indicate the positions of edible and poisonous mushrooms. The position and direction of the entity is indicated using a triangle. This is done by converting the `Environment` object to a string; I had provided an implementation of the `__str__` method in the Environment class. Further information is also displayed, such as the current inputs and outputs to the entity at each step. This method is also paused for interactivity, the user presses ENTER

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

to move to the next step or ESC to leave the viewer. This interactivity was crucial in early testing to locate complicated bugs and perform behavioural tests fo the system.

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 simply plotted by reading the average fitness values recorded from the simulation, plotting fitness over generation number. Typically I ran 10 simulations for each language type at once, so 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.

PUT INITIAL RESULTS HERE

Interpreting this equation further we can see that their equation does not match their 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, $\min(d_{\text{poisonous}}$ 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, then these two values should be *added* instead of subtracted. I thus implemented the equation as the following:

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

This gave far better correlations, as discussed in the next chapter.

Chapter 4

Evaluation

Show graphs, run by run etc.

Variance is interesting to discuss

Could also be interesting to query what happens in an elbow in the graph - visual and a bit subjective. Three categories - those that do not take off, those that suddenly take off and those that gradually take off.

Variance in each run, consequences of initialisation/activation, bigger questions about how reliable neural networks are, whether they are a good model for language evolution.

Chapter 5

Conclusion

Bibliography

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

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

Cangelosi, A. and Parisi, D. (2002). *Simulating the Evolution of Language.*

Cangelosi, A. and Parisi, D. (2012). *Simulating the evolution of language.* Springer Science & Business Media.

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

Clark, E. (1995). *The lexicon in acquisition.*

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

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

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

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

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Simulating Language Learning and Evolution

Zébulon Goriely — Queens' — zg258

Originator: Zébulon Goriely

18 October 2019

Project Supervisor: Prof. Paula Buttery and Dr. Andrew Caines

Director of Studies: Prof. Alastair Beresford

Project Overseers: Prof. Pietro Lio & Dr. Robert Mullins

Introduction

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

In the paper, Cangelosi and Parisi use small feed-forward neural networks to control the behaviour of each agent. The weights are initially random; a genetic algorithm is used

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

to improve the fitness of the species over many generations. The agents are also given linguistic abilities; input and output nodes of the neural networks produce signals that allow for communication.

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

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

Starting Point

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

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

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

Work to be Done

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

Implementing the Simulation

1. Set up the simulation environment by creating the world grid and implementing the properties of poisonous and edible mushrooms. Create the simulation loop divided into regular 'epochs'.

2. Create the agents for the simulation, giving them position and energy properties.

3. Implement feedforward neural networks to control the behaviour of the agents; input units to identify the location of the nearest mushroom, visual perception units to observe mushroom properties (only when close enough) and signal perception units for when language is implemented. The output units control the movement of the agent and production of signals. There will also be hidden units.

4. Implement the genetic algorithm that runs after all agents complete the simulation. The fittest agents are determined by the energy level (based on eating edible mushrooms and avoiding poisonous mushrooms). The fittest agents are then chosen for asexual reproduction, producing offspring that have genetic mutations in the form selecting a percentage of the weights to change by a random amount.

5. Create two different populations, one without language (where the signal perception units are always set to the same, constant value) and one with an externally imposed language (where the signal perception units are set to one of two signals depending on the type of the nearest mushroom).

6. Create a third population with an evolved language. Instead of externally imposed signals, in each simulation cycle one of the other agents is randomly selected as the 'speaker' and its output is connected to the input signal perception units of the 'listener'.

Analysis

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

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

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

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

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

Testing

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

Success Criterion

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

Timetable and Milestones

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

25th October – 10th November

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

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

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

11th November – 24th November

Middle of Michaelmas Term.

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

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

25th November – 8th December

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

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

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

9th December – 22nd December

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

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

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

23rd December – 12th January

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

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

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

13th January – 2nd February

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

Progress Report Deadline: 31st January

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

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

3rd February – 23rd February

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

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

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

24th February – 15th March

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

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

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

16th March – 5th April

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

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

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

6th April – 19th April

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

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

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

20th April – 8th May

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

Final Deadline: 8th May

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

Milestone: Submission of Dissertation

Possible Extensions

Graphic Visualisation

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

Symbolic Theft vs. Sensorimotor Toil

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

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

Investigating the Evolution of Syntax

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

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

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

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

Resources Declaration

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

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