UNIVERSITY OF SUSSEX

Adaptive Systems

Assignment 1

2020

Candidate: 213120

Degree: MSc Intelligent & Adaptive Systems

University of Sussex – School of Engineering and Informatics

Contents

1. Introduction

Given a simulated two-wheeled and non-inertial agent initialised in a random orientation at (0,0) on an infinite plane, we explore three potential methodologies to create controllers that are able to navigate it to a bright light located at (10,0), before returning to its origin as quickly as possible. In order to do this, the controller is able to take advantage of one noisy sensor, a forwards-facing photo sensor, as well as two noisy actuators, the left and right motors that form the agents differential drive mechanism.

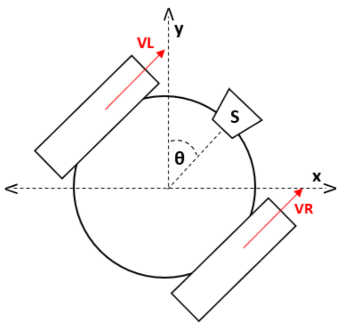


Figure 1 – Structure of simulated agent

We can divide the agent's overall goal in to two distinct parts; navigating to the light being the former and finding its way home being the latter. Neither task is trivial due to the presence of noise in the system, however the latter is particularly challenging due to the agent's inability to sense its home position in the way it can do so with the light.

2. Methods

2.3 Microbial Tumbling

The first steps I took in tackling task 1 (getting to the light) was to create a controller, that instead of returning fixed motor values, it returned a function of the readings given by the light sensor. Even with one motor value being left as a constant, and one being set proportionally to the light sensor readings, this achieved a spinning effect in which the agent would accelerate slightly more-so during the point of each rotation in which it held a bearing towards the light source. By effect, the agent is able to accomplish task 1.

This approach, whilst it served as a good benchmark and starting point, it had many flaws which we will go in to more detail on in section 3.1.2, the most critical of which being its inability to manage task 2. It was also not an effective solution to task 1 once time had been taken into consideration, and this had the knock-on effect of rendering certain 'return home' techniques such as Dead Reckoning obsolete. Due to the noise, techniques relying on an internal representation of the state of the system (e.g the agents co-ordinates/bearing) become increasingly inaccurate with time due to the growing discrepancy between the systems true state, and that which is estimated and stored within the internal representation

2.2 Dead Reckoning

In order for Dead Reckoning to be a viable strategy, our performance on task 1 had to be streamlined significantly. I used a divide and conquer strategy to design a single controller to accomplish both, the problem was divided in to the following 5 distinct stages seen in the high-level pseudocode below.

```
SpinController(Sensors, Info)
Input: Sensors  – A list of integers, each representing the sensor value at that timestep.
       Info     – Internal state representation.
Output: Action  – Left and right motor speeds.
        Info    – Internal state representation.
```

```
If stage = 1:
    Return SpinSearch()
If stage = 2:
    Return LocateMaxIntensity()
If stage = 3:
    Return MoveTowardsTheLight()
If stage = 4:
    Return FindHomeBearing()
If stage = 5:
    Return GoHome()
```

A brief description of each stage is as follows:

SpinSearch(): The agent spins 1 fill rotation on the spot, taking note of the maximum light intensity found.

LocateMaxIntensity(): Rotate until that maximum intensity is found again.

MoveTowardsTheLight(): Move forwards for a specified amount of time steps. Only advance to the next stage if the light is reached, otherwise return to stage one (SpinSearch).

FindHomeBearing(): Calculate the home bearing using the estimated co-ordinates stored within Info. The pseudocode for this is as follows:

```
If $y < 0$:
    $returnBearing = 360 - Tan^{-1}( x / y )$
Else:
    $returnBearing = 180 + Tan^{-1}( x / y )$
```

It's worth noting that this method of dead reckoning works regardless of the agent's initial position, so long as the estimated co-ordinates stored within Info are initialised at (0,0), as all calculations are relative to these initial conditions.

GoHome(): Moves forwards until home is reached, or a calculated distance to origin has been covered.

The *Info* variable, containing the complete internal state representation, is also worthy of some explanation. Due to the nature in which the controller is called upon by the simulation, certain variables must not only be passed between the controller's methods, but also between different iterations of the same methods. These variables have been encapsulated within *info* to allow all 5 of the controller's methods to return complete information describing the state of the system, in a consistent format that allows for a much more succinct implementation. *Info* is a list with contents as follows:

Info[0] = stage – An integer representing one of the 5 stages outlined in the high-level pseudocode.

Info[1] = estimatedPoses – A list containing 3 floats, representing the agents estimated *(x, y)* positions and its bearing respectively.

Info[2] = maxIntensity – A float used to store the maximum light intensity detected during the first stage.

Info[3] = firstPass – A boolean needed by the first stage (*SpinSearch)* to indicate whether it is the methods first execution of the cycle, as when initiating one of these full rotations, *maxIntensity* should be reset, and the *initialBearing* must be taken note of.

Info[4] = initialBearing – A float allowing stage 1 to terminate once one full rotation has been completed by the agent.

Info[5] = stepsTaken – Integer used by stages 3 and 5 as counter to keep track of how long the agent has been moving for.

2.2 CTRNN – NEAT

Although I now had one working methodology, there still seemed to be room for improvement both in terms of the success rate, particularly when returning home, as well as the time required.

The continuous time-recurrent neural network (CTRNN) architecture was first proposed by Beer in his 1990 publication [1], however it wasn't until 1995 that he also demonstrated [2] their potential suitability for our task due to their ability to approximate trajectories of dynamical systems. Whilst feed-forward neural networks can excel at exhibiting reactive behaviours, that is one in which the output is wholly determined by the current input (such as navigating towards the light), our agent must exhibit non-reactive behaviour to find its way home, and Jakobi's [3] review of which network architectures are best suited to this reinforce my decision to use a CTRNN.

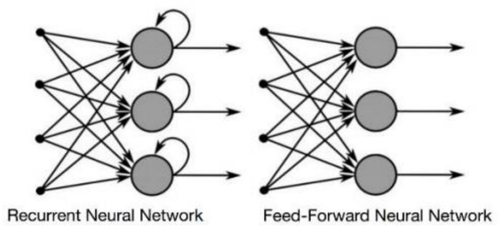


Figure 2 – Neural network architectural comparison [6]

Neural networks that allow recurrence, or feedback loops within its connections, allow internal states of sorts to be represented, with these states being partially dependant on prior outputs. This can function in essence as memory and so it is easy to see why this is ideal for our task, where the output (return home bearing) is a function of past inputs (motor speeds) rather than present ones.

NeuroEvolution of Augmenting Topologies (NEAT), first proposed by Stanley in 2004 [4], is a ground-up approach to evolving arbitrary neural networks. Taking inspiration from the biological world, it starts with the smallest and simplest networks possible (much like a single-celled organism), and through complexification it is able to find highly sophisticated and complex optimised neural networks. Much of the source code for my Python implementation of NEAT can be found in the online documentation [5].

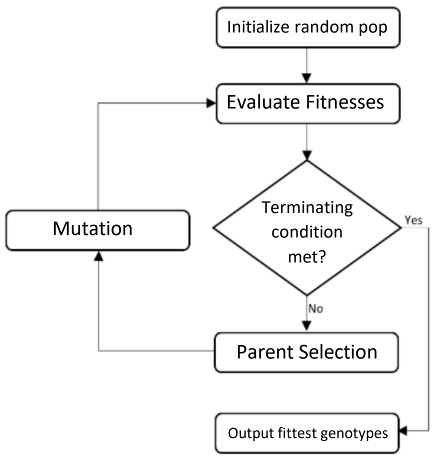


Figure 3 – High-level GA description

As with any genetic algorithm, a fitness function is needed to quantify the success of each genotype. Once again taking a divide-and-conquer approach, I first implemented a fitness function for Task 1 - reaching the light. My initial implementation solely rewarded the genotypes based on how fast they reached the light, giving those that didn't a fitness of 0, however in practice, this didn't provide enough of a fitness gradient to be climbed. My revised implementation is described below:

```
Task1Fitness(Poses)
Input: Poses  – A list containing a genotype's (x,y) position at each timestep
Output: Fitness – A float representing the genotypes fitness

    If reachedLight:
        Fitness = timeLimit – reachedLightAt
    Else:
        Fitness = - distanceToLight$^2$
```

Note the use of a squared error in the case that the agent doesn't reach the light. This is a heavier penalisation that simply being given a fitness of 0, and provides a significantly improved fitness gradient, leaving much less up to pure chance during the evolution stage. My fitness function for Task 2 is much the same as that described above, however in the case that the agent doesn't reach its goal (the origin rather than the light) a lower bound is set on its fitness, equivalent to – *[task1Fitness]*. This is because a total fitness is to be given by totalling the two values, and without this added constraint, it would be very possible for an agent who reached the light to achieve a lower fitness than one that didn't.

Performance	**Theoretical Fitness Range**
Failed Task 1	-infinity : 0
Completed Task 1	0 : 5,000
Completed Task 2	0 : 10,000

In practice, the true possible fitness ranges lie narrower than the theoretical ones outlined above. The upper bounds will be lower as no agent could complete either goal before a single timestep has passed. Equally the lower bound for fitness having completed Task 2 sits higher than 0, as this would require a fitness of 0 to be achieved on both Tasks 1 and 2. This is only achievable if a goal is achieved on the last possible timestep, and it isn't possible to complete both tasks on a single time-step.

In terms of inputs to the network, I shall once again be taking a ground-up approach beginning with just the sensor readings and the goal of completing just Task 1, before investigating the additions of which of the following lead to improvements in performance and the ability to complete Task 2:

- Task indication
- Motor speeds from previous time-step
- Forwards velocity
- Estimated bearing
- Estimated co-ordinates

3. Results and Discussion

3.1 Microbial Tumbling

3.1.1 Success Rate

For the purposes of Task 1, this approach was dynamic in that it was insensitive to the random initial orientation of the agent. I will discuss the effect of the light's position in section 3.1.2, however its success did not depend on it for the most part. Tweaking the specifics of the actuator velocities would lead to some differences in performance, and I found best success when one motor was set to a reciprocal of sorts to that which was directly dependant on the light sensor readings. For example:

*leftMotor = gain * sensorReading*

rightMotor = x – (gain sensorReading)*

With x set as a suitable constant, this can have the effect of allowing the two motor values to converge to form a more direct path to the light once the sensor reading reaches a certain threshold, whilst maintaining the ability to turn sharply enough to avoid getting stuck in an orbital trajectory around the light source, with too great of a radius for it to count as having reached the light.

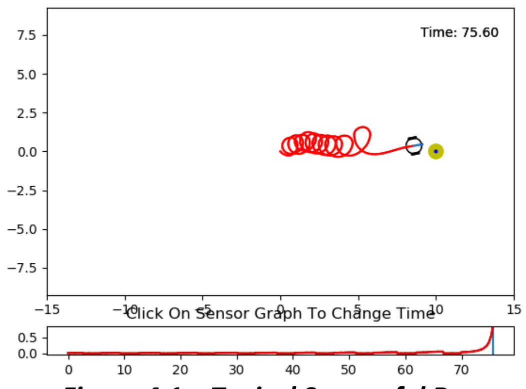


Figure 4.1 – Typical Successful Run

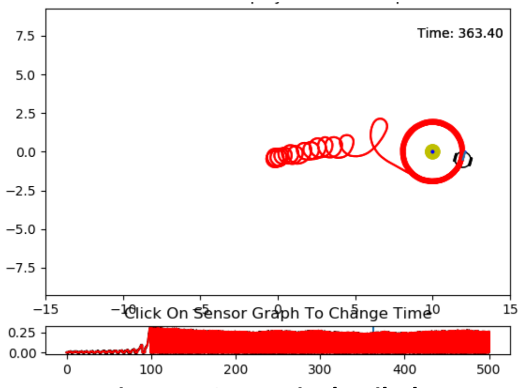
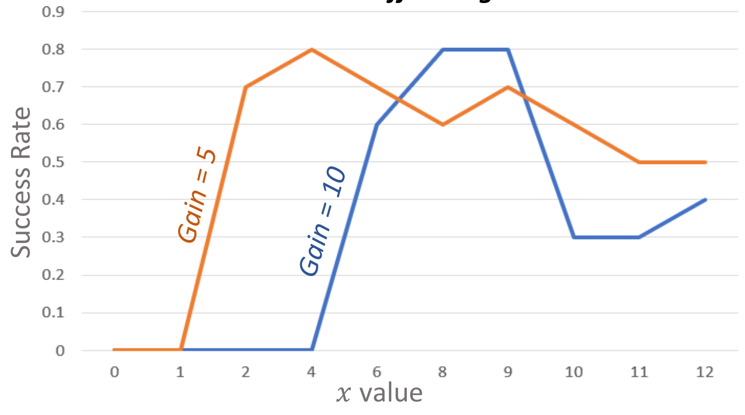


Figure 4.2 – Typical Failed Run

In figure 5 you can see the effect of a varying x value. If x is too low, the success rate is 0, as the agent is not able to reach a velocity sufficient to achieve the displacement needed to get to the light source. As x increases beyond the gain, the influence of the sensor reading in the right motor velocity fades into insignificance, the agent loses the ability to ever veer right, and we see the success rates tend towards 0.4, which is what we found it to be with that motor fixed at a high constant, e.g 1000. This convergence is a result of the tanh function mapping all action (motor) values, regardless of their magnitude, to the (-1, 1) interval. I anticipate this to also be the reason why we see a steeper drop-off in performance with a higher gain – the increased gain allows the performance to peak with a higher x value, but the tanh transformation causes convergence at the same x value regardless, resulting in the steeper decline.

It must also be noted that although the highest achievable success rates seem similar regardless of the gain, those done with a gain of 10 were typically achieved 46% faster.

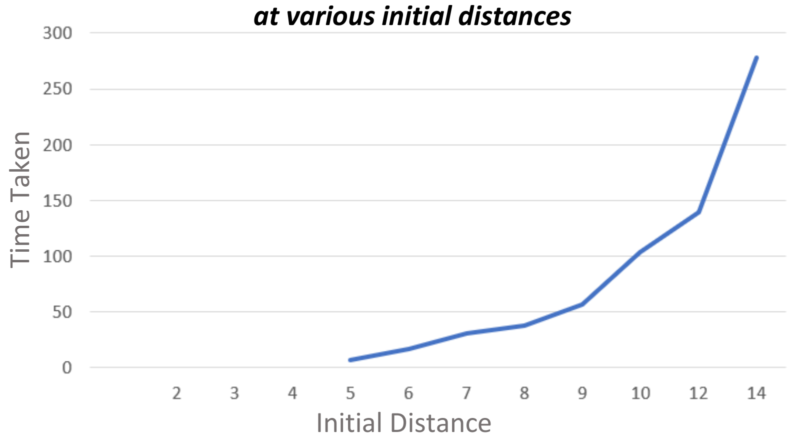
Figure 5 – Graph to show how success rate varies with x value with different gains.



3.1.2 Adaptivity

Since this controller is completely insensitive to orientation, I was able to investigate a large aspect of its adaptivity by simply varying the light source's distance from the agent. My key focus here was the time taken to complete task 1 on those runs which were successful, you can see my results below in figure 6.

Figure 6 – Graph to show time taken to reach the light at various initial distances



Here's where we found some limitations in the controllers adaptivity. It was unable to reach the light at any initial distances less than 5, as the agent would immediately find itself on a wide orbital path. Beyond this, we can see the time taken increase in an exponential fashion as the initial distance between the agent and the light increases, this solution could be described to have a $O(n^2)$ running time.

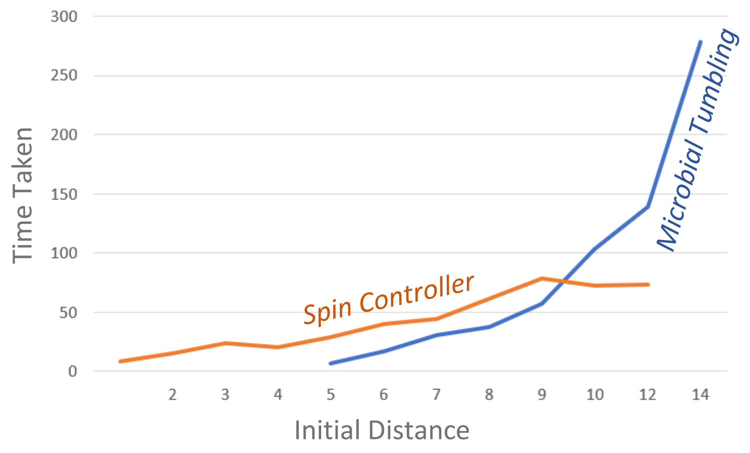
Regardless of how much optimisation and refinement is put into this methodology, it will always maintain its fundamental flaws. Namely its inability to complete task 2, and its poor time complexity resulting in the 'sabotage' of any techniques relying on internal representations estimating the position of the agent, due to the accumulation of noise in the system.

3.2 Dead Reckoning

3.2.1 Task One

My first tests were to see if I had achieved, and if so to what extent, my goal of streamlining performance on Task 1. In figure 7 you can see the results of SpinController, the controller I designed to be able to return home using dead reckoning (outlined in section 2.2), compared to the Microbial Tumbling strategy.

Figure 7 – Graph to show comparative times taken to reach the light at various initial distances



At a glance it's clear that it was a success. Not only did the Spin Controller succeed at Task 1 in situations where the microbial tumbling based strategy couldn't, but it did so with a 97.5% success rate, and reduced the quadratic time complexity by an order of magnitude to one that is linear. In the 2.5% of tested cases that were not successful, the noise perturbed the agent during stage 2 (in which it spins to re-locate the previously detected maximum intensity) to a position further away from the light source such that the maximum detectable intensity from there no longer satisfied the minimum threshold to be equated to the previously detected maximum.

3.2.2 Task Two

SpinController tested to have a Task 2 success rate of 43.6%, giving an overall success rate of 42.5%. All of those that were unsuccessful can be attributed to the noise present in the system - the overwhelming majority of which due to the home bearing being calculated incorrectly as a result; the remainder being the 2.5% that failed at Task 1.

Figure 8.1 – A Typical Successful run

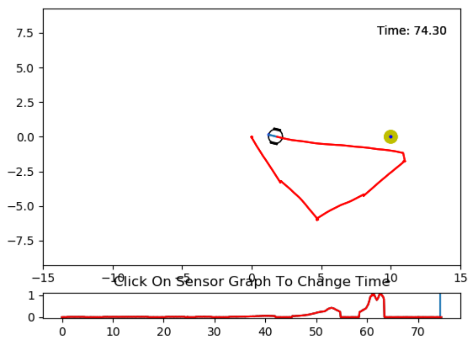


Figure 8.2 – The associated discrepancy between actual and estimated poses

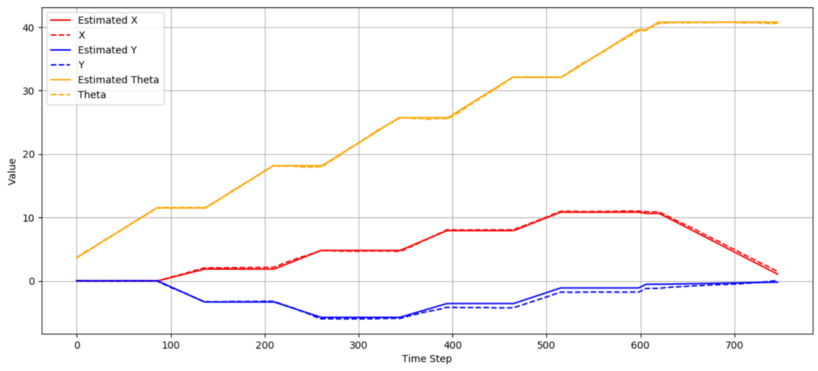


Figure 9.1 – An un-successful run

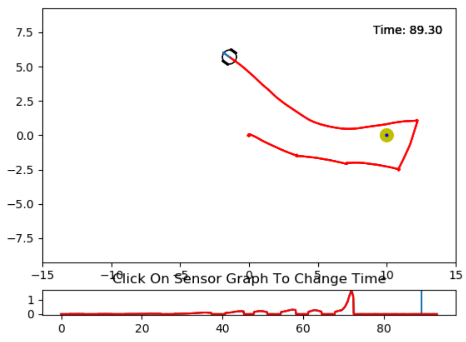
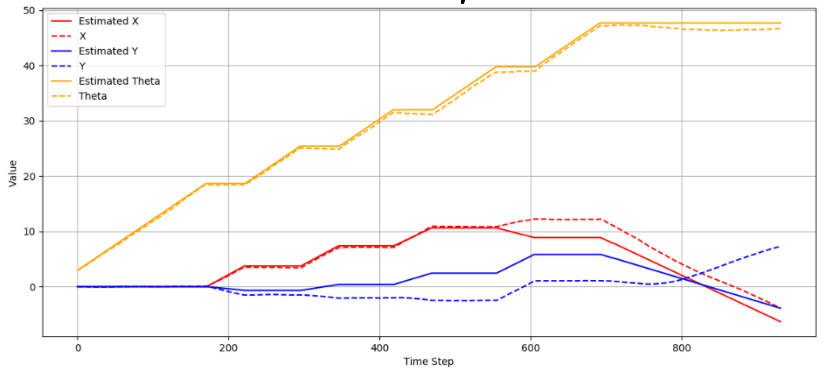
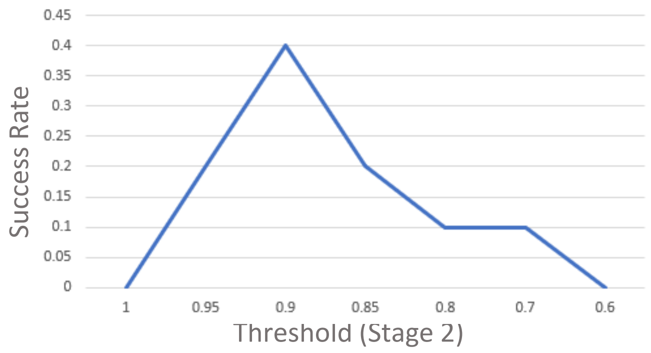


Figure 9.2 – The associated discrepancy between actual and estimated poses



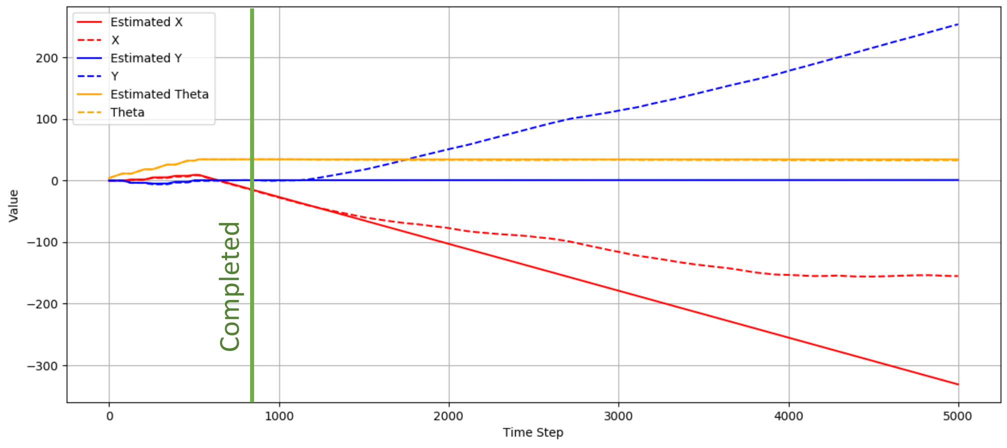
Most typical successful runs resemble that depicted in figure 8.1. Although much more direct runs can and do occur, this trajectory is in part a result of the threshold used by stage 2 to set the lower-bound of what qualifies as the previously detected maximum. A very high threshold raises the probability of successful runs being more direct, however it comes at the cost of a greater chance of failure during Task 1, in the manner described at the end of section 3.2.1. The nature of this relationship between threshold and success rate can be seen below in figure 10.

Figure 10 – Effect of varying threshold on success rate



It's easy to see why the run illustrated in figure 9.1 was unsuccessful after seeing the clear difference in agreement apparent in figure 9.2 versus 8.2. Since the noise is purely probabilistic, the longer a simulation is run for, the greater the detrimental effect of it is likely to be – although visible in figure 9.2, it is far more vivid in figure 10 which depicts a run that was successful but allowed to continue running for the maximum time permitted. This serves to demonstrate the importance of streamlining Task 1 as much as we did.

Figure 11 – A Successful Run not Terminated on Completion



4. Concluding Remarks

Although I was happy with the results being achieved by my implementation of dead reckoning within SpinController, there still seemed to be some room for improvement on Task 2. Sadly I must confess that I personally was not able to achieve any noteworthy results with the CTRNN-NEAT approach beyond agents endlessly circling, or heading off on an infinite straight path. I remain confident that this approach could be used to further optimise performance, and so I attribute this deficiency both to my lack of experience with this complex combination of techniques, as well as the limitations imposed by the processing power of my very dated home computer in combination with the in-place time restrictions.

We must not forget that we haven't fallen short of our goals as a result of this shortcoming; we proposed two techniques for solving Task 1, as well as one to solve Task 2 with a respectable degree of consistency and reliability. After a substantial amount of research into NeuroEvolution of Augmenting Topologies (NEAT) I find myself fascinated and more motivated than ever before to make this a point of future research - so much so that I have now opted to make it the subject of my final-year project.

5. Bibliography

[1] R. D. Beer, Intelligence as Adaptive Behavior: an Experiment in Computational Neuroethology, San Diego, CAUnited States: Academic Press Professional, Inc., 1990.

[2] R. D. Beer, "On the Dynamics of Small Continuous-Time Recurrent Neural Networks," *Adaptive Behaviour,* vol. 3, pp. 469-509, 1995.

[3] N. Jakobi, "Minimal Simulations For Evolutionary Robotics," University of Sussex, 1998.

[4] K. O. Stanaley, "Efficient Evolution of Neural Networks Through Complexification," Department of Computer Sciences, The University of Texas , Austin, 2004.

[5] “ NEAT-Python’s documentation!,” CodeReclaimers, 2015. [Online]. Available: https://neat-python.readthedocs.io/en/latest/. [Accessed 2 February 2020].

[6] N. Donges, “Recurrent Neural Networks and LSTM,” 25 February 2018. [Online]. Available: https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5. [Accessed 20 February 2020].